

Fluidity Manual

Applied Modelling and Computation Group (AMCG)

<http://amcg.ese.ic.ac.uk>

Department of Earth Science and Engineering,
South Kensington Campus,
Imperial College London,
London, SW7 2AZ, UK

Version 4.1.1 Release

Overview

Fluidity is an open source, general purpose, multi-phase CFD code capable of solving numerically the Navier-Stokes and accompanying field equations on arbitrary unstructured finite element meshes in one, two and three dimensions. It uses a moving finite element/control volume method which allows arbitrary movement of the mesh with time dependent problems. It has a wide range of finite element/control volume element choices including mixed formulations. Fluidity is coupled to a mesh optimisation library allowing for dynamic mesh adaptivity and is parallelised using MPI.

Chapter 1 of this manual gives details on how prospective users can obtain and set up Fluidity for use on a personal computer or laptop. The fluid and accompanying field equations solved by Fluidity and details of the numerical discretisations available are discussed in chapters 2 and 3 respectively. When discretising fluid domains in order to perform a numerical simulations it is inevitable that at certain scales the dynamics will not be resolved. These sub-grid scale dynamics can however play an important in the large scale dynamics the user wishes to resolve. It is therefore necessary to parameterise these sub-grid scale dynamics and details of the parameterisations available within Fluidity are given in chapter 4. Fluidity also contains embedded models for the modelling of non-fluid processes. Currently, a simple biology (capable of simulating plankton ecosystems) and a sediments model are available and these models are detailed in chapter 5.

As mentioned above, one of the key features of Fluidity is its ability to adaptively re-mesh to various fields so that resolution can be concentrated in regions where the user wishes to accurately resolve the dynamics. Details regarding the adaptive re-meshing and the manner in which Fluidity deals with meshes are given in chapters 7 and 6.

Fluidity has its own specifically designed options tree to make configuring simulations as painless as possible. This options tree can be viewed and edited using the diamond GUI. Details on how to configure the options tree are given in 8. Output from simulations is in the VTK format and details regarding viewing and manipulating output files are given in chapter 9. Finally, in order to introduce users to a range of common configurations and to the functionality available within Fluidity, chapter 10 presents examples covering a range of fluid dynamics problems. For information regarding the style and scope of this manual, please refer to Appendix A.

Fluidity Primer

Please check the Fluidity webpage at <http://amcg.es.ic.ac.uk/Fluidity> to ensure you are reading the most recent version of this manual. Methods for installing Fluidity may sometimes change, and instructions may be updated!

This is a one-page primer for obtaining Fluidity and running a simple example. It assumes that:

- You are running Ubuntu Linux, release 10.10 (Maverick) or newer
- You have administrative rights on your computer
- You know how to run a terminal with a command prompt
- You have a directory in which you can create files

Set up your computer to access the Fluidity repository by typing:

```
sudo apt-add-repository -y ppa:fluidity-core/ppa
```

Type your password when prompted.

Once this completes, update your system and install Fluidity along with its supporting software by typing:

```
sudo apt-get update
sudo apt-get -y install fluidity
```

Now uncompress the packaged examples to a directory in which you can create files (in this example, /tmp) by typing:

```
cd /tmp
tar -zxvf /usr/share/doc/fluidity/examples.tar.gz
```

Change into an examples directory (top_hat is suggested as a straightforward starter) and run the example:

```
cd examples/top_hat/
make preprocess
make run
make postprocess
```

You have now run your first Fluidity model. Chapter 10 describes this and the other examples provided with Fluidity.

Contents

1	Getting started	1
1.1	Introduction	1
1.2	Obtaining Fluidity	1
1.2.1	Overview	1
1.2.2	Fluidity binary packages for Ubuntu Linux	1
1.2.3	Fluidity source packages	2
1.2.4	Bazaar	2
1.3	Building Fluidity	5
1.3.1	Setting up the build environment	5
1.3.2	Configuring the build process	5
1.3.3	Compiling Fluidity	7
1.3.4	Installing Fluidity and diamond	8
1.4	Running Fluidity	9
1.4.1	Running Fluidity in serial	9
1.4.2	Running Fluidity in parallel	10
1.5	Running diamond	10
1.6	Working with the output	11
2	Model equations	13
2.1	How to navigate this chapter	13
2.2	Advection–Diffusion equation	13
2.2.1	General equation	13
2.2.2	Scalar boundary conditions	14
2.3	Fluid equations	15
2.3.1	Mass conservation	15
2.3.2	Momentum conservation	15
2.3.3	Equations of state & constitutive relations	17
2.3.4	Momentum boundary conditions	19
2.4	Extensions, assumptions and derived equation sets	20
2.4.1	Equations in a moving reference frame	20
2.4.2	Linear Momentum	22
2.4.3	Buoyancy and Hydrostacy	22
2.4.4	The Boussinesq approximation	23
2.4.5	Supplementary boundary conditions and body forces	24
2.4.6	Multi-material simulations	25
2.4.7	Multi-phase simulations	26
3	Numerical discretisation	29
3.1	Introduction & some definitions	29
3.2	Spatial discretisation of the advection-diffusion equation	29
3.2.1	Continuous Galerkin discretisation	29
3.2.2	Boundary conditions	35
3.2.3	Discontinuous Galerkin discretisation	36

3.2.4	Control volume discretisation	43
3.3	The time loop	53
3.3.1	Time notation	54
3.3.2	Nonlinear relaxation	54
3.3.3	The θ scheme	55
3.4	Time discretisation of the advection-diffusion equation	55
3.4.1	Discontinuous Galerkin	55
3.4.2	Control Volumes	56
3.5	Momentum equation	56
3.5.1	Boussinesq approximation	57
3.6	Pressure equation for incompressible flow	57
3.6.1	Pressure correction	58
3.7	Velocity and pressure element pairs	60
3.7.1	Continuous Galerkin pressure with control volume tested continuity	61
3.8	Balance pressure	61
3.9	Free surface	63
3.10	Wetting and drying	63
3.11	Linear solvers	63
3.11.1	Iterative solvers	64
3.11.2	Preconditioned Krylov subspace methods	65
3.11.3	Convergence criteria	66
3.12	Algorithm for detectors (Lagrangian trajectories)	67
4	Parameterisations	69
4.1	Turbulent flow modelling and simulation	69
4.1.1	Reynolds Averaged Navier Stokes Modelling	69
4.1.2	Large-Eddy Simulation (LES)	74
4.2	Ice shelf parameterisation	78
4.2.1	Boundary condition at ice surface	79
5	Embedded models	81
5.1	Biology	81
5.1.1	Four component model	81
5.1.2	Six-component model	83
5.1.3	Photosynthetically active radiation (PAR)	84
5.1.4	Detritus falling velocity	85
5.2	Sediments	85
5.2.1	Deposition and erosion	85
6	Meshes in Fluidity	89
6.1	Supported mesh formats	89
6.2	Surface and regions ids	89
6.3	Meshes and function spaces	90
6.4	Extruded meshes	90
6.5	Periodic meshes	91
6.6	Meshing tools	91
6.6.1	Mesh Verification	91
6.6.2	Mesh creation	91
6.6.3	Mesh conversion	91
6.6.4	Decomposing meshes for parallel	92
6.6.5	Decomposing a periodic mesh	93
6.7	Non-Fluidity tools	93
6.7.1	Terreno	93
6.7.2	Gmsh	93

6.7.3	Importing contours from bathymetric data into Gmsh	94
7	Adaptive remeshing	95
7.1	Motivation	95
7.2	A typical adaptive loop	96
7.3	Representing meshes as metric tensors	97
7.4	Adaptive remeshing technology	98
7.5	Using mesh adaptivity	99
7.5.1	Choice of norm	100
7.5.2	Absolute, relative and p - metrics	101
7.5.3	Weights	101
7.5.4	Gradation parameter	101
7.5.5	Maximum and minimum edge length tensors	102
7.5.6	Maximum and minimum numbers of nodes	102
7.5.7	Metric advection	103
7.6	Interpolation	103
7.7	Parallel adaptivity	105
7.8	The cost of adaptivity	105
8	Configuring Fluidity	107
8.1	Overview	107
8.2	Options syntax	107
8.2.1	Allowed Characters	107
8.2.2	Named options	108
8.3	The options tree	108
8.3.1	Simulation Name	109
8.3.2	Problem Type	109
8.3.3	Geometry	109
8.3.4	IO	111
8.3.5	Timestepping	114
8.3.6	Physical parameters	116
8.4	Meshes	117
8.4.1	Reading meshes from file	117
8.4.2	Deriving meshes from other meshes	118
8.5	Material/Phase	120
8.6	Fields	120
8.6.1	Types of field	120
8.6.2	Setting field values	121
8.6.3	Region IDs	123
8.6.4	Mathematical constraints on initial conditions	123
8.7	Advected quantities: momentum and tracers	123
8.7.1	Spatial discretisations	123
8.7.2	Temporal discretisations	126
8.7.3	Source and absorption terms	126
8.7.4	Sponge regions	126
8.8	Solving for pressure	127
8.8.1	Geostrophic pressure solvers	127
8.8.2	First guess for poisson pressure equation	127
8.8.3	Removing the null space of the pressure gradient operator	127
8.8.4	Continuous Galerkin pressure with control volume tested continuity	127
8.9	Solution of linear systems	128
8.9.1	Iterative Method	128
8.9.2	Preconditioner	128
8.9.3	Relative Error	128

8.9.4	Absolute Error	128
8.9.5	Max Iterations	128
8.9.6	Start from Zero	129
8.9.7	Remove Null Space	129
8.9.8	Solver Failures	129
8.9.9	Reordering RCM	129
8.9.10	Solver Diagnostics	129
8.10	Equation of State (EoS)	130
8.11	Sub-grid Scale Parameterisations	131
8.11.1	GLS	131
8.11.2	k- ϵ Turbulence Model	132
8.11.3	Large Eddy Simulation Models	133
8.12	Boundary conditions	134
8.12.1	Adding a boundary condition	134
8.12.2	Selecting surfaces	135
8.12.3	Boundary condition types	135
8.12.4	Special input date for boundary conditions	137
8.12.5	Special cases	138
8.13	Astronomical tidal forcing	139
8.14	Ocean biology	139
8.15	Sediment model	140
8.16	Large scale low aspect ratio ocean simulations	141
8.16.1	Options that must be switched on	141
8.16.2	Recommended or optional settings	142
8.17	Geophysical fluid dynamics problems	143
8.17.1	Problem type	143
8.17.2	Geometry	143
8.17.3	Timestepping	144
8.17.4	Material/phase	144
8.18	Mesh adaptivity	147
8.18.1	Field settings	147
8.18.2	General adaptivity options	148
8.19	Multiple material/phase models	151
8.19.1	Multiple material models	152
8.19.2	Multiple phase models	154
8.20	Compressible fluid model	155
8.20.1	Pressure options	155
8.20.2	Density options	155
8.20.3	Velocity options	156
8.20.4	Restrictions: discretisation options and element pairs	156
9	Visualisation and Diagnostics	157
9.1	Visualisation	157
9.2	Online diagnostics	157
9.2.1	Fields	157
9.3	Offline diagnostics	165
9.3.1	vtktools	166
9.3.2	Diagnostic output	171
9.3.3	fltools	171
9.4	The stat file	188
9.4.1	File format	189
9.4.2	Reading .stat files in python	191
9.4.3	Stat file diagnostics	191
9.4.4	Detectors	196

10 Examples	199
10.1 Introduction	199
10.2 One dimensional advection	199
10.2.1 Overview	199
10.2.2 Configuration	200
10.2.3 Results	202
10.2.4 Exercises	203
10.3 The lock-exchange	203
10.3.1 Overview	203
10.3.2 Configuration	204
10.3.3 Results	204
10.3.4 Exercises	206
10.4 Lid-driven cavity	208
10.4.1 Overview	208
10.4.2 Configuration	208
10.4.3 Results	208
10.4.4 Exercises	209
10.5 2D Backward facing step	210
10.5.1 Overview	210
10.5.2 Geometry	210
10.5.3 Initial and boundary conditions	211
10.5.4 Results	211
10.6 3D Backward facing step	211
10.6.1 Configuration	211
10.6.2 Geometry	213
10.6.3 Initial and boundary conditions	213
10.6.4 Results	214
10.7 Flow past a sphere: drag calculation	216
10.7.1 Overview	216
10.7.2 Configuration	216
10.7.3 Results	216
10.7.4 Exercises	218
10.8 Rotating periodic channel	218
10.8.1 Overview	218
10.8.2 Results	219
10.9 Water column collapse	220
10.9.1 Overview	220
10.9.2 Problem specification	221
10.9.3 Results	221
10.9.4 Exercises	226
10.10 The restratification following open ocean deep convection	226
10.10.1 Overview	226
10.10.2 Configuration	227
10.10.3 Results	228
10.11 Tides in the Mediterranean Sea	228
10.11.1 Overview	228
10.11.2 Configuration	229
10.11.3 Results	229
10.12 Hokkaido-Nansei-Okai tsunami	230
10.12.1 Overview	230
10.12.2 Configuration	232
10.12.3 Results	234
10.12.4 Exercises	236

10.13	Tephra settling	236
10.13.1	Overview	236
10.13.2	Problem specification	237
10.13.3	Results	237
10.13.4	Exercises	239
Bibliography		241
A	About this manual	253
A.1	Introduction	253
A.2	Audience and Scope	253
A.3	Style guide	254
A.3.1	Headings	254
A.3.2	Language	254
A.3.3	Labelling	254
A.3.4	Images	254
A.3.5	fml options	255
A.3.6	Generating pdf and html output	255
A.3.7	Representing source code	255
A.3.8	Bibliography	256
A.3.9	Mathematical notation conventions	256
B	The Fluidity Python state interface	259
B.1	System requirements	259
B.2	Data types	259
B.2.1	Field objects	260
B.2.2	State objects	260
B.3	Predefined data	261
B.4	Importing modules and accessing external data	261
B.5	The persistent dictionary	261
B.6	Debugging with an interactive Python session	261
B.7	Limitations	262
C	External libraries	263
C.1	Introduction	263
C.2	List of external libraries and software	263
C.3	Installing required libraries on Debian or Ubuntu	264
C.4	Manual install of external libraries and software	264
C.4.1	Build environment	265
C.4.2	Compilers	265
C.4.3	Python	266
C.4.4	Numerical Libraries	267
C.4.5	VTK and supporting software	269
C.4.6	Supporting Libraries	270
D	Troubleshooting	273
E	Mesh formats	275
E.1	Mesh data	275
E.1.1	Node location	275
E.1.2	Element topology	275
E.1.3	Facets	275
E.1.4	Surface IDs	276
E.1.5	Region IDs	276
E.2	The triangle format	276

E.3	The Gmsh format	278
Index		281

List of Figures

2.1	Schematic of coordinates in a frame rotating with a sphere. The rotation is about a vector pointing from South to North pole. A point on the surface of the sphere x and its perpendicular distance from the axes of rotation x_{\perp} are shown. The latitude of x is given by φ and the unit vectors i, j and k represent a local coordinate axes at a point x in the rotating frame: i points Eastwards, j points Northwards and k points in the radial outwards direction.	21
3.1	One-dimensional (a, b) and two-dimensional (c, d) schematics of piecewise linear (a, c) and piecewise quadratic (b, d) continuous shape functions. The shape function has value 1 at node A descending to 0 at all surrounding nodes. The number of nodes per element, e , depends on the polynomial order while the support, s , extends to all the elements surrounding node A	31
3.2	Pure advection of a 1D top hat function in a periodic domain at CFL number 1/8 after 80 timesteps using a continuous Galerkin discretisation.	34
3.3	Pure advection of a 1D top hat function in a periodic domain at CFL number 1/8 after 80 timesteps using a continuous Galerkin discretisation with streamline-upwind stabilisation.	35
3.4	Pure advection of a 1D top hat function in a periodic domain at CFL number 1/8 after 80 timesteps using a continuous Galerkin discretisation with streamline-upwind Petrov-Galerkin stabilisation.	35
3.5	One-dimensional (a, b) and two-dimensional (c, d) schematics of piecewise linear (a, c) and piecewise quadratic (b, d) discontinuous shape functions. The shape function has value 1 at node A descending to 0 at all surrounding nodes. The number of nodes per element, e , depends on the polynomial order while the support, s , covers the same area as the element, e	37
3.6	One-dimensional (a) and two-dimensional (b) schematics of piecewise constant, element centred shape functions. The shape function has value 1 at node A and across the element, e , descending to 0 at the element boundaries. As with other discontinuous shape functions, the support, s , coincides with the element, e	44
3.7	Comparison between (a) a two-dimensional finite volume simplex mesh and (b) the equivalent control volume dual mesh (solid lines) constructed around a piecewise linear continuous finite element parent mesh (dashed lines). In the finite volume mesh the nodes (e.g. A) are element centred whereas in the control volume dual mesh the nodes are vertex based. In 2D the control volumes are constructed around A by connecting the centroids of the neighbouring triangles to the edge midpoints. See Figure 3.8 for the equivalent three-dimensional construction.	44
3.8	The six dual control volume mesh faces within a piecewise linear tetrahedral parent mesh element. Each face is constructed by connecting the element centroid, the face centroids and the edge midpoint.	45

3.9	One-dimensional (a, b) and two-dimensional (c, d) schematics of piecewise constant control volume shape functions and dual meshes based on the parent (dashed lines) linear (a, c) and quadratic (b, d) finite element meshes. The shape function has value 1 at node A descending to 0 at the control volume boundaries. The support, s , coincides with the volume, v .	46
3.10	Calculation of the upwind value, c_{u_k} , on an unstructured simplex mesh (a) internally and (b) on a boundary. The control volume mesh is shown (solid lines) around the nodes (black circles) which are co-located with the solution nodes of the parent piecewise linear continuous finite element mesh (dashed lines). An initial estimate of the upwind value, $c_{u_k}^*$ is found by interpolation within the upwind parent element. The point-wise gradient between this estimate and the donor node, c_{c_k} , is then extrapolated the same distance between the donor and downwind, c_{d_k} , to the upwind value, c_{u_k} .	48
3.11	The Sweby (a) and ULTIMATE (b) limiters (shaded regions) represented on a normalised variable diagram (NVD). For comparison the trapezoidal face value scheme is plotted as a dashed line in both diagrams. γ is the Courant number at the control volume face.	48
3.12	The HyperC face value scheme represented on a normalised variable diagram (NVD).	49
3.13	The UltraC face value scheme represented on (a) a normalised variable diagram (NVD) and (b) a modified normalised variable diagram. For comparison (a) also shows the HyperC face value scheme as a dotted line.	50
3.14	The coupled limiter for the field c^I represented by the grey shaded area on a normalised variable diagram (NVD). Labels in the upper left blue region refer to the case when the difference between the parent sum downwind and upwind values has the same sign as the limited field, $\text{sign}\left(c_{d_k}^{\sum I} - c_{u_k}^{\sum I}\right) = \text{sign}\left(c_{d_k}^I - c_{u_k}^I\right)$. Similarly, labels in the lower right yellow region refer to the case when the signs of the slopes are opposite, $\text{sign}\left(c_{d_k}^{\sum I} - c_{u_k}^{\sum I}\right) \neq \text{sign}\left(c_{d_k}^I - c_{u_k}^I\right)$. The regions are separated by the upwinding line, $\bar{c}_f^I = \bar{c}_{c_k}^I + \bar{c}_{c_k}^{\sum I-1} - \bar{c}_f^{\sum I-1}$.	52
3.15	Outline of the principal steps in the nonlinear iteration sequence.	54
3.16	A sketch representing the Guided Search method used in combination with an explicit Runge-Kutta algorithm to advect the Lagrangian detectors with the flow.	68
5.1	The fluxes between the biological tracers. Grazing refers to the feeding activity of zooplankton on phytoplankton and detritus.	86
5.2	Six-component biology model.	86
6.1	An unstructured mesh around a NACA0025 aerofoil, the coloured patches show a possible decomposition into 4 partitions.	92
7.1	Example of mesh modification operations.	99
8.1	A Diamond screenshot showing the <code>/geometry/dimension</code> option. Note that the option path is displayed at the bottom of the diamond window	108
8.2	A Diamond screenshot showing the <code>/geometry/mesh::Coordinate</code> option. Note that the name is shown in brackets in the main Diamond window but after double colons in the path in the bottom bar.	109
8.3	Periodic unit square with surface IDs 1-4 shown.	119
9.1	Configuration of a diagnostic field using a diagnostic algorithm in Diamond. Here a pressure gradient diagnostic is defined.	163
9.2	Visualisation of a heat flux diagnostic in a 2D cavity convection simulation using <code>statplot</code> .	183
9.3	Visualisation of the six basis functions of the quadratic triangle generated by <code>visualise_elements</code> .	185
9.4	Example configuration of the stat file for <code>.../vector_field(Velocity)</code> .	189

10.1	Initial condition and numerical solutions after 100 s , for the 1D top hat tracer advection problem.	201
10.2	Conservation and bounds checking using statplot.	203
10.3	Lock-exchange temperature distribution (colour) with meshes, over time (t)	206
10.4	Distance along the domain (X) and Froude number (Fr) for the no-slip and free-slip fronts in the lock-exchange. The values of Härtel et al. [2000] and Simpson and Britter [1979] are included for reference.	206
10.5	Mixing diagnostics for the lock-exchange.	207
10.6	Diagnostic fields from the lid-driven cavity problem at steady state at 1/128 resolution. Left: the streamfunction. Right: the vorticity. The contour levels are taken from those given by Botella and Peyret [1998] in their tables 7 and 8.	209
10.7	Convergence of the eight error metrics computed for the lid-driven cavity problem with mesh spacing. The eight metrics are described in the text.	210
10.8	Schematic of the domain for the two-dimensional flow past a backward facing step.	211
10.9	Snapshots of the velocity magnitude from the 2D run at times 5, 10 and 50 time units (top to bottom) from the k-epsilon run. The evolution of the dynamics to steady state can be seen, in particular the downstream movement of the streamline reattachment point (where zero-magnitude contour touches bottom).	212
10.10	Streamwise velocity profiles from the 2D run at $x/h = 1.33, 2.66, 5.33, 8.0$ and 16.0 downstream of the step, where $h = 1$ is the step height. The converged solution is in blue. Ilinca's numerical and Kim's experimental data [Ilinca and Pelletier, 1997] are in red and black respectively. The recirculation region is indicated by negative velocities.	212
10.11	Evolution of reattachment length in k-epsilon simulation.	213
10.12	Schematic of the domain for the three-dimensional flow past a backward facing step problem.	214
10.13	From top to bottom: vertical plane cuts through the 3D domain showing the velocity magnitude at times 5, 25 and 50 time units. The evolution of the dynamics to steady state can be seen, in particular the downstream movement of the streamline reattachment point (indicated by contours of $U = 0$).	215
10.14	Streamwise velocity profiles from the 3d run at $x/h = 4, 6, 10$ and 19 downstream of the step, where $h = 1$ is the step height, at $t = 5$ seconds.	215
10.15	Streamlines and surface mesh in the flow past the sphere example. Top-left to bottom-right show results from Reynolds numbers $Re = 1, 10, 100, 1000$	217
10.16	Details of the mesh and flow at $Re = 1000$	217
10.17	Comparison between the numerically calculated drag coefficients (C_D , circles) and the correlation (10.3) (solid line) for $Re = 1, 10, 100, 1000$	218
10.18	Velocity forcing term and analytic solutions for velocity and pressure for the rotating periodic channel test case. Note that each of these quantities is constant in the x direction.	219
10.19	Error in the pressure and velocity solutions for the rotating channel as a function of resolution.	220
10.20	(a) Initial set-up of the water volume fraction, α_1 , and the velocity and pressure boundary conditions for the two-dimensional water column collapse validation problem [Zhou et al., 1999]. The presence of water is indicated as a blue region and the interface to air is delineated by contours of the volume fraction at 0.025, 0.5 and 0.975. The locations of the pressure (P2) and water depth gauges (H1, H2) are also indicated. (b) The adapted mesh used to represent the initial conditions.	222
10.21	The evolution of the water volume fraction, α_1 , over several timesteps. The presence of water, $\alpha_1 = 1$, is indicated as a blue region and the interface to air, $\alpha_1 = 0$, is delineated by contours at $\alpha_1 = 0.025, 0.5$ and 0.975	223
10.22	The evolution of the adaptive mesh over the same timesteps displayed in Figure 10.21. The mesh can be seen to closely follow the interface between the water and air.	224

10.23	Comparison between the experimental (circles) and numerical water gauge data at H1 (i) and H2 (ii), $x_1 = 2.725m$ and $2.228m$ respectively. Experimental data taken from Zhou et al. [1999] through Park et al. [2009].	225
10.24	Comparison between the experimental (circles) and numerical pressure gauge data at P2, $x_1 = 3.22m$, $x_2 = 0.16m$. Experimental data taken from Zhou et al. [1999] through Park et al. [2009].	226
10.25	A vertical slice through the domain showing the initial temperature stratification. The domain is a cylinder of radius 250 km and height 1 km.	227
10.26	The temperature cross-section at a depth of 40m.	228
10.27	Results of the mixing stats diagnostic, showing how the temperature is mixed during the simulation. There is initially most water with a temperature of 0.7-0.8. This mixes during the course of the simulation.	229
10.28	Plots of the tidal harmonic amplitudes in the Mediterranean Sea from ICOM and the high resolution model of Tsimplis et al. [1995].	231
10.29	Plots of the tidal harmonic phases in the Mediterranean Sea from ICOM and the high resolution model of Tsimplis et al. [1995].	232
10.30	Locations of 62 tide gauges in the Mediterranean Sea. Modified from Wells [2008] with data originally taken from Tsimplis et al. [1995].	233
10.31	Scatter diagrams plotting harmonic amplitudes from ICOM at each gauge location against tide gauge data.	234
10.32	The bathymetry and the three gauge stations used for the Hokkaido-Nansei-Oki tsunami example.	235
10.33	The input wave elevation of the Okushiri tsunami test case (a) and the numerical and experimental results at "Gauge 1" (top), "Gauge 2" (middle) and "Gauge 3" (bottom) (b).	235
10.34	Tephra (particle) phase volume fraction at time $t = 10, 30, 50, 80, 110$ seconds. These images are from a lower-resolution version of the tephra settling example problem, with a characteristic element size of $\Delta x = 0.01$ m.	237
10.35	Tephra (particle) phase volume fraction at time $t = 10, 30, 50, 80, 110$ seconds. The onset of plumes is between 10 and 30 seconds. Note that these images are from a high-resolution version of the tephra settling example problem, with a characteristic element size of $\Delta x = 0.0025$ m.	238
10.36	Maximum velocity of the tephra phase against time.	238

Chapter 1

Getting started

1.1 Introduction

This first chapter gives a brief guide to setting-up, running and visualising simulations. It is assumed that you have access to a system running Linux or UNIX with all the Fluidity supporting libraries installed. If not, you may wish to reference appendix C which contains detailed information for your systems administrator to help setting up a Linux workstation.

1.2 Obtaining Fluidity

1.2.1 Overview

Fluidity is available both as precompiled binaries for Ubuntu Linux version 10.10 (Maverick) and newer, and as source code available via bazaar or as gzipped tarballs. Which method you use to obtain Fluidity depends primarily on whether you use Ubuntu Linux and whether you wish to modify the Fluidity source code.

Fluidity generally attempts to support any given release of Ubuntu for 18 months from the point of release.

Users who run Ubuntu Linux and who have no need to change the source code will probably wish to use binary packaged Fluidity. Developers and users on non-Ubuntu platforms will probably wish to build Fluidity from source.

1.2.2 Fluidity binary packages for Ubuntu Linux

Fluidity is distributed via the `fluidity-core` launchpad package archive. If you run Ubuntu Linux version 10.10 (Maverick) or newer (11.04 or newer after April 2012) and have administrative privileges on your computer, you can add the package archive to your system by typing:

```
sudo apt-add-repository -y ppa:fluidity-core/ppa
```

Type your password if prompted. Once the repository has been added, update your system and install Fluidity along with the required supporting software by typing:

```
sudo apt-get update
sudo apt-get -y install fluidity
```

You now have Fluidity installed on your computer. Examples, as referred to in chapter 10, are available as a compressed tarball and can be expanded into a writeable directory with:

```
tar -zxvf /usr/share/doc/fluidity/examples.tar.gz
```

1.2.3 Fluidity source packages

Fluidity is released as compressed source archives which can be downloaded from links on the Fluidity homepage, <http://amcg.ese.ic.ac.uk/Fluidity>.

You can download just the source, or you can download extra archives containing examples, standard tests, and long tests.

To uncompress the Fluidity 4.1-release source from an archive file in Linux, run:

```
tar -zxvf fluidity-4.1.tar.gz
```

This should create a directory, `fluidity-4.1/`, containing your copy of the source code. To the standard tests (sufficient to run 'make mediumtest') or long tests (extended testing), download the relevant archive into the `fluidity-4.1/` directory, and in that directory run (in the case of tests):

```
tar -zxvf fluidity-tests-4.1.tar.gz
```

or in the case of longtests:

```
tar -zxvf fluidity-longtests-4.1.tar.gz
```

The files will appear in a `tests/` directory.

1.2.4 Bazaar

The Fluidity source code is hosted on launchpad. Experienced bazaar users can obtain the latest development version of Fluidity from `lp:fluidity`. For inexperienced users, or those wanting a more stable version than the development trunk, please read on to learn how to obtain a copy of Fluidity.

As a Fluidity user you need only be aware of two modes of operation in bazaar. The first is checking out a copy of the source code from the central repository, the second is updating your copy of the code to reflect changes that have been made to the central repository. Think of the repository as a central archive of source code held on the Launchpad platform.

There are a few other bazaar features useful for users which will be mentioned in passing.

Details of how to commit changes to Fluidity back into the central repository are outside the scope of this user manual; if you are transitioning to become a developer who commits changes back into Fluidity please contact an existing Fluidity developer to find out how to get commit privileges.

1.2.4.1 Checking out a current copy of Fluidity

Fluidity can be obtained in three forms: release, stable, and trunk.

The first, release, is aimed at general users of Fluidity. Releases are static versions of the code, and an individual release with a specific version number will never change content. New versions will be produced whenever there is sufficient demand for new features to be included. Release are very well tested and do not contain any code known to have bugs. As such, releases are infrequent so will generally not contain the latest changes. The manual and other resources are written to support releases.

The second, *stable* series, is aimed at users who require access to development code but do not wish to be exposed to the frequent but transitory issues that tend to occur in bleeding-edge Fluidity. *Stable* will generally not contain the very latest code changes, as it is only updated when Fluidity passes all tests in the automated testing framework. It will generally be updated on a weekly to fortnightly basis, but may go longer between updates during periods of particularly intense development. There are no guarantees that subtle bugs will not make it through to *stable*. The manual should, but may not be fully up to date for *stable*, and examples may not be well tested.

The third, *trunk*, is the day to day development version of Fluidity. It is primarily used by developers and is not recommended for everyday users. Updates appear many times each day, and the *trunk* will frequently have periods where not all tests are passing. Use at your own risk.

To check out a copy of Fluidity you can use one of the following three commands for the *release*, *stable* and *trunk* versions respectively, which are given here in entirety but will be broken down in the subsequent discussion so you can better understand them.

Release:

```
bzr co lp:fluidity/4.1 fluidity/
```

Stable:

```
bzr co lp:fluidity/stable fluidity/
```

Trunk:

```
bzr co lp:fluidity fluidity/
```

The first part of this command calls the `bzr` program:

```
bzr
```

`bzr` takes a first argument which describes the mode in which it will run. In this case, it is going to check out a copy of the code, which abbreviates to `co`:

```
bzr co
```

The second argument to `bzr` describes the location used to check out Fluidity from, in this case using the Launchpad site (abbreviated to `lp`) and the series required (ie, `fluidity/stable`):

```
bzr co lp:fluidity/stable
```

This is now a valid command line, and if you ran it would check out a copy of the *stable* series of Fluidity into a directory `stable/` in your current directory. If you prefer to have the code checked out into a directory called `fluidity/`, append that directory name to the end of the command line so `bzr` checks out the code where you want it to:

```
bzr co lp:fluidity/stable fluidity/
```

Bear in mind that the Fluidity check out is of the order of many hundreds of megabytes and should be checked out onto a file system which has corresponding amounts of free space. As it checks out, `bzr` should print progress information, and when it is complete should clear this information and return to a command prompt.

If you are asked to check out a different series or branch of Fluidity you will simply need to change the name after `lp:` and should be given the relevant information by whoever has requested the checkout.

1.2.4.2 Updating your copy of Fluidity

This section does not apply to users having checked out a Fluidity release.

If you are interested in keeping up to date with the latest developments in Fluidity you will probably want to update your local copy of the Fluidity code to reflect changes made to the central source code repository. To do this, change directory so that you are in the directory which you checked out Fluidity to (in the above case, a directory called `fluidity/`) and then run `bzr`'s update command, which is abbreviated to `up` as per:

```
bzr up
```

You will probably see a number of lines with a leading `M`, denoting that a file in your local copy of Fluidity is being modified, such as:

```
M    preprocessor/Populate_State.F90
```

This manual assumes that you are not modifying any of the files in your copy of Fluidity. If you do so, and your changes clash with changes from the central repository, you may end up with `bzr` reporting conflicts when you update. This is worth being aware of, but brings you into the realms of Fluidity development, outside the scope of this manual.

1.2.4.3 Other useful Bazaar commands

If you can check out and update your copy of Fluidity then you are equipped to be an active Fluidity user. However, you may wish to know about a few more useful subversion commands, including getting information about your local version of the code using:

```
bzr info
```

You may well want to do this if someone asks you which branch of Fluidity you are using, at which point you're looking for a line such as:

```
checkout of branch: bzr+ssh://bazaar.launchpad.net/%2Bbranch/fluidity/
```

with the relevant information. If you are asked for the revision of Fluidity you're using, a shortcut command to give this is `bzr revision-info`.

To get information on what files have been modified in your local copy use the the following:

```
bzr status
```

Note that if you have compiled Fluidity you will see a large number of files in the 'unknown' category, which have been generated by the compilation process. To suppress those supply the parameter `-V`:

```
bzr status -V
```

To get specific information on the difference between a local file and the central repository use the following:

```
bzr diff <filename>
```

You can get help on the `bzr` command `<command>` with:

```
bzr help <command>
```

and list available commands with:

```
bzr help commands
```

replacing `<filename>` with the file you want differences reported for.

If you wish to delve deeper into Bazaar operation, an excellent source of detail is <http://doc.bazaar.canonical.com/bzr.2.4/en/user-guide/index.html> which is the official Bazaar User Guide.

1.3 Building Fluidity

The build process for Fluidity comprises a configuration stage and a compile stage. In the simplest form, this can be completed with two commands, run in the directory containing your local source code check out, which is denoted by `<<fluidity_source_path>>` in this manual:

```
cd <<fluidity_source_path>>
./configure
make
```

You may often only wish to perform this basic build, but frequently you will want more fine-grained control over the configuration procedure or you will want to perform non-default compilation steps. The following section describes these procedures.

Note that at this point configuration refers to the build-time configuration options which define how the Fluidity program will be compiled, and do not refer to configuration of the options that you will run Fluidity with once it has built. However, presence or lack of features configured at the build stage may change what is available to you at run time.

It is assumed throughout this section that you are in the top-level directory of your local copy of the Fluidity code for the purposes of describing configuration and compilation commands.

1.3.1 Setting up the build environment

Depending on the computer you are building Fluidity on, you may need to set up the build environment before configuring Fluidity. On the standard Fluidity workstation built around Ubuntu Linux, you will need to initialise the PETSc package by running:

```
module load petsc-gcc4
```

This is very specific to the standard Fluidity workstation, and other systems may well have detailed requirements to be carried out before building Fluidity. If you are in any doubt, stop now, and consult with your local systems administrator who should be able to help you resolve any problems ahead of time. If you are not on a standard Fluidity workstation you may want to refer to (or refer your systems administrator to) appendix C of this manual before proceeding.

The build environment is discussed in more detail in sections [1.3.2.2](#) and [1.3.2.3](#).

1.3.2 Configuring the build process

For a full list of the build-time configuration options available in fluidity, run:

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

Key configuration options are described here, but you are advised to check output from the above command for any changed or new options which may have been introduced since the last update of this manual.

Where you wish to specify multiple configuration options at once, supply them all on the same configuration command line, separated by spaces. For example:

```
./configure --prefix=/data/fluidity --enable-debugging
```

Note that there is one key option NOT enabled by default which users running the Fluidity examples will need to enable to make all examples work. This is `--enable-2d-adaptivity`, which cannot be turned on as a default as it makes use of external code that is not license-compatible with all other Fluidity codes. Hence, it is a reasonable default state for you to run:

```
./configure --enable-2d-adaptivity
```

One additional consideration is worth making at this point; do you want to put any components of the Fluidity build outside the build directory for later use? An automatic installation for Fluidity (and also for diamond) is available, and defaults to using a directory which can (normally) only be written to with superuser privileges. However, you can tell the install process to put files elsewhere with the use of the 'prefix' flag to configure. A typical use of this might be to tell the install process to put files in your home directory, using the 'HOME' environment variable which should be set by the system automatically. To do this in addition to enabling 2D adaptivity, you would type:

```
./configure --enable-2d-adaptivity --prefix=$HOME
```

We'll go on now to discussing enabling and disabling features.

1.3.2.1 Enabling and disabling features

Fluidity has a number of optional features which may be enabled or disabled at build time. For a list of all these features see the output of configuring with the `--help` argument. This list should indicate which options are enabled by default by appending `(default)` to the option description. An example of enabling Fluidity's debugging feature at build time would be:

```
./configure --enable-debugging
```

Whilst a number of options have the facility to be enabled and disabled, doing so may be prejudicial to the expected normal running of Fluidity so unless you are fully aware of the consequences of enabling or disabling features it is recommended that you do not do so.

1.3.2.2 Specifying locations of supporting libraries

Fluidity requires many supporting libraries, which can either be provided via environment variables (see later discussion for how to do this) or, in specific cases, provided via options during configuration. This uses the `--with` option, for example specifying the directory containing BLAS using:

```
./configure --with-blas-dir=/data/libraries/netlib/BLAS
```

Whilst there is also the option to supply `--without` arguments, this is likely to be highly prejudicial to the normal running of Fluidity or, in many cases, be incompatible with building Fluidity such that the configuration exits with an error.

1.3.2.3 Environment variables set for configuration

A description of environment variables is outside the scope of this manual, and Fluidity users are encouraged to find an experienced UNIX user who can explain the rudiments of environment variables to them if they are not already familiar with how to set and use them. Influential environment

variables are listed towards the end of the help output from Fluidity's configuration. Particularly notable are:

`LIBS`, which allows passing a series of linker arguments such as `"-L/data/software/libs"` describing how to access libraries at link-time. This will often be set or appended to by loading modules on modern UNIX systems.

`FCFLAGS` and `FFLAGS` which describe flags passed to the Fortran and Fortran77 compilers respectively and allow you broad control of the overall the build process. This will also often be set or appended to by loading modules on modern UNIX systems.

`PETSC_DIR` defines the base directory into which your PETSc install has been placed. This will often be set by loading an environment module specific to your system, but if you have a local build of PETSc you may need to set this variable yourself. Note that for most Fluidity users, having PETSc to provide solver functionality is unavoidable so setting this variable by some means is necessary in almost all cases. The standard Fluidity workstation allows you to do this by running `module load petsc-gcc4`.

`VTK_INCLUDE` and `VTK_LIBS` may be important to set if VTK is not installed at a system level. VTK is critical to Fluidity for writing output files, and many UNIX systems lack some VTK components so it frequently ends up installed in a nonstandard location.

1.3.3 Compiling Fluidity

Once you have successfully configured Fluidity, you need to compile the source code into binary files including programs and libraries. In the simplest form you can do this by running:

```
make
```

which will generate Fluidity and Diamond programs in the `bin/` directory and a set of libraries in the `lib/` directory.

If you have a modern, powerful computer then you can speed this process up significantly by parallelising the make, running:

```
make -jN
```

where `N` is the number of CPU cores of your system.

Note that this is very resource-intensive and whilst most modern workstations should be equal to the task it should not be run on shared machines (ie, login nodes of compute clusters) or systems with smaller quantities of memory.

If you want to build the extended set of Fluidity tools which supplement the main Fluidity program, section 9.3.3, run:

```
make fltools
```

If this is the first time you have run Fluidity on your computer and you want to check that it has built correctly, you can run the included suite of test problems at three levels. The shortest tests test individual units of code for correctness and can be run with:

```
make unittest
```

Please note that, due to compiler bugs which Intel have shown no inclination to fix, the 'make unittest' suite is known to fail with all releases of the Intel compiler. Thus, it is advised that you not run 'make unittest' if building Fluidity with any version of the Intel compiler.

To run the suite of short test cases which more extensively test the functionality of your Fluidity build, run:

```
make test
```

For the most comprehensive set of tests included in your checked out copy of Fluidity, run:

```
make mediumtest
```

If you have obtained Fluidity through source archives rather than through subversion you will need to have also downloaded and unpacked the standard tests archive to be able to run `make test` and `make mediumtest`.

Note that even on the most modern systems `make mediumtest` may take on the order of an hour and on slower systems may take on the order of many hours to complete.

1.3.4 Installing Fluidity and diamond

It is perfectly possible to run Fluidity and diamond from inside the build tree, and many users do just that with appropriate setting of variables. However, you may want to install at least diamond if not the rest of the Fluidity package into the directory you had the option of specifying during the configure stage.

There are three key commands to be aware of here.

The first is:

```
make install
```

This will install Fluidity and a full set of Fluidity tools into the directory you specified with 'prefix' during configuration. If you didn't specify a directory there, it will try to install into the `/usr` directory, which generally needs superuser privileges. If you get a lot of 'permission denied' messages, you probably forgot to specify a prefix (or specified an invalid prefix) at configure time, and the install is trying to write to a superuser-only or nonexistant directory.

Assuming you specified a prefix of `\$HOME`, Fluidity will install into directories such as `bin/`, `lib/`, and `share/` in your home directory.

Fluidity does not by default install diamond, the graphical configuration tool used to set up Fluidity. If you do not already have a central installation of diamond on your computer, you may want to install a copy of it into the directory you specified with 'prefix' at configure time. To do this, run:

```
make install-diamond
```

which will install diamond into a `bin/` directory inside your 'prefix' directory. In the case we've been working through, this would be your home directory. It will also install supporting python files into a subdirectory of `lib/`.

To set up your environment for automatic running of diamond, you probably need to do three things. For this process, we will assume you specified a prefix of `\$HOME` to configure.

First, set your path to automatically find the diamond program:

```
export PATH=$PATH:$HOME/bin
```

Next, run:

```
ls -d $HOME/lib/python*/site-packages/diamond
```

This should return something like:

```
/home/fred/lib/python2.6/site-packages/diamond
```

Copy all of the above apart from the trailing `/diamond` into the following command, so that you type (for the above example):

```
export PYTHONPATH=$PYTHONPATH:/home/fred/lib/python2.6/site-packages
```

Finally, you should register Fluidity with diamond for your account by telling it where to find the Fluidity schema, by typing:

```
make install-user-schemata
```

Note that this points to the current build tree that you are in; if you change the name of this build tree, or move it, you will need to un the above command again to register the new location.

1.4 Running Fluidity

1.4.1 Running Fluidity in serial

To run Fluidity in serial use the following command:

```
<<fluidity_source_path>>/bin/fluidity foo.flml
```

Here, `foo.flml` is a Fluidity configuration file which defines the options for the model. These files are covered in detail in Chapter 8 and are set up using a simple Graphical User Interface, Diamond. See section 1.5 for more information on how to run Diamond.

There are other options that can be passed to the Fluidity executable. A full list can be obtained by running:

```
<<fluidity_source_path>>/bin/fluidity
```

This will produce the following output:

```
Revision: 3575 lawrence.mitchell@ed.ac.uk-20110907125710-d44hcr4no0ev8icc
Compile date: Sep  7 2011 13:59:34
OpenMP Support                no
Adaptivity support            yes
2D adaptivity support         yes
3D MBA support                no
CGAL support                  no
MPI support                    yes
Double precision              yes
CGNS support                  no
NetCDF support                yes
Signal handling support       yes
Stream I/O support            yes
PETSc support                  yes
Hypre support                  yes
ARPACK support                yes
Python support                yes
Numpy support                  yes
VTK support                    yes
Zoltan support                 yes
Memory diagnostics            no
FEMDEM support                 no
Hyperlight support            no
```

Usage: fluidity [options ...] [simulation-file]

Options:

```
-h, --help
    Help! Prints this message.
-l, --log
    Create log file for each process (useful for non-interactive testing).
    Sets default value for -v to 2.
-v <level>, --verbose
    Verbose output to stdout, default level 0
-p, --profile
    Print profiling data at end of run
    This provides aggregated elapsed time for coarse-level computation
    (Turned on automatically if verbosity is at level 2 or above)
-V, --version
    Version
```

Note that this also gives information on which version is being used, the build-time configuration options used and a list of command-line options for Fluidity.

Running Fluidity will produce several output files. See Chapter 9 for more information.

1.4.2 Running Fluidity in parallel

Fluidity is a fully-parallel program, capable of running on thousands of processors. It uses the Message Passing Interface (MPI) library to communicate information between processors. Running Fluidity in parallel requires that MPI is available on your system and is properly configured. The terminology for parallel processing can sometime be confusing. Here, we use the term *processor* to mean one physical CPU core and *process* is one part of a parallel instance of Fluidity. Normally, the number of processors used matches the number of processes, i.e. if Fluidity is split into 256 parts, it is run on 256 processors.

To run in parallel one must first decompose the mesh. See section 6.6.4 for more information on how to do this. Fluidity must then be run within the mpirun or mpiexec framework. Simply prepend the normal command line with mpiexec:

```
mpiexec -n [PROCESSES] <<fluidity_source_path>>/bin/fluidity foo.flml
```

1.5 Running diamond

Diamond is the Graphical User Interface (GUI) by which a Fluidity simulation is configured. The flml file that Fluidity uses is an XML file, which defines the meshes, fields and options for Fluidity. Chapter 8 covers the options that are currently available. In addition, Diamond will display inline help on options where available.

If the `make` command successfully completed, there will be an executable program `diamond` in the `bin/` directory. Running diamond is now as simple as typing:

```
<<fluidity_source_path>>/bin/diamond
```

If the `make install` command completed and a directory which executable programs were installed into to the `PATH` environment variable, one may also be able to run diamond simply by typing

`diamond` at the command line from any directory on your system. This may also be possible if the Diamond package for Ubuntu or Debian is installed by running:

```
sudo apt-get install diamond
```

Note that installing Diamond on a system-wide basis will require superuser privileges on the system.

1.6 Working with the output

Once Fluidity has been built and run, result files will be generated, that will need visualising and post-processing. These data are stored in VTK files, one per output dump. How these files can be visualised and post-processed is covered in detail in chapter 9.

As well as visualisation, another important output resource is the stat file, described in detail in section 9.4, which contains data from the model collected at run time.

See the [AMCG website](#) for more information.

Chapter 2

Model equations

2.1 How to navigate this chapter

This chapter covers the fluid and associated field equations modelled by Fluidity.

problem	underlying equations	boundary conditions	other considerations
scalar advection	2.2.1, 2.2.1.1	2.2.2.1, 2.2.2.2	
scalar advection-diffusion	2.2.1, 2.2.1.2		
momentum equation	2.3	2.3.4.1, 2.3.4.2, 2.3.4.3, 2.3.4.4	2.4

The material covered in this chapter is dealt with in great detail in [Batchelor \[1967\]](#) and [Landau and Lifshitz \[1987\]](#). [Cushman-Roisin \[1994\]](#) is also a useful reference.

2.2 Advection–Diffusion equation

2.2.1 General equation

The general form the equation that governs the evolution of a scalar field c (e.g., passive tracer, species concentration, temperature, salinity) is

$$\frac{\partial c}{\partial t} + \nabla \cdot (\mathbf{u}c) = \nabla \cdot (\bar{\kappa} \nabla c) - \sigma c + F, \quad (2.1)$$

where $\mathbf{u} = (u, v, w)^T$ is the velocity vector, $\bar{\kappa}$ is the diffusivity (tensor), σ is an absorption coefficient ($-\sigma c$ is sometimes termed Rayleigh or linear friction) and F represents any source or reaction terms.

2.2.1.1 Advection

The advection term in (2.1), given by

$$\nabla \cdot (\mathbf{u}c) = \mathbf{u} \cdot \nabla c + (\nabla \cdot \mathbf{u})c, \quad (2.2)$$

expresses the transport of the scalar quantity c in the flow field \mathbf{u} . Note that for an incompressible flow $\nabla \cdot \mathbf{u} = 0$ (see section 2.3.3) resulting in the second term on the right hand side of (2.2) dropping out. However, there may be numerical reasons why the discrete velocity field is not exactly

divergence free, in which case this term may be included in the discretisation. Fluidity deals with the advection term in the form

$$\nabla \cdot (\mathbf{u}c) + (\beta - 1)(\nabla \cdot \mathbf{u})c, \quad (2.3)$$

so that $\beta = 1$ corresponds to the conservative form of the equation and $\beta = 0$ to the non-conservative.

2.2.1.2 Diffusion

The diffusion term in (2.1), given by

$$\nabla \cdot (\bar{\kappa} \nabla c), \quad (2.4)$$

represents the mixing of c and may be due to molecular mixing of individual particles via Brownian motion, or mixing via large scale (in comparison to the molecular scale) motion in the flow. For many applications (2.4) can be written in simpler forms. Often, diffusion is isotropic giving $\bar{\kappa} = \text{diag}(\kappa, \kappa, \kappa)$ and thus the diffusion term may be written as

$$\nabla \cdot (\bar{\kappa} \nabla c) = \kappa \nabla \cdot \nabla c = \kappa \nabla^2 c = \kappa \Delta c. \quad (2.5)$$

In domains with high aspect ratio dynamics one often uses a smaller value of diffusivity in the 'thin' direction. For example, in the atmosphere or ocean we may choose a horizontal (eddy) diffusivity κ_H and a vertical (eddy) diffusivity κ_V so that $\bar{\kappa} = \text{diag}(\kappa_H, \kappa_H, \kappa_V)$ with $\kappa_V < \kappa_H$. In this case the diffusion term may be written as

$$\nabla \cdot (\bar{\kappa} \nabla c) = \kappa_H \left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} \right) + \kappa_V \frac{\partial^2 c}{\partial z^2}. \quad (2.6)$$

Note that the second-order terms defined above are often termed Laplacian diffusion.

2.2.1.3 Absorption, reaction and source terms

The absorption term in (2.1)

$$-\sigma c, \quad (2.7)$$

has the effect of decreasing the magnitude of c (note the minus sign and the fact that σ would typically be positive). It is sometimes termed Rayleigh friction.

The remaining term in (2.1)

$$F = \sum_i F_i, \quad (2.8)$$

can encompass a number of source and reaction terms. Those terms where F_i are a given function of time, location or a-priori known fields are termed sources (and sometime sinks if they are negative). Those terms which are also functions of other prognostic fields are termed reactions and are common when dealing with chemistry or biology.

2.2.2 Scalar boundary conditions

To form a well-posed system upon which to attempt a numerical discretisation, the set of equations (discussed above) describing the behaviour of the system must be supplemented with appropriate boundary conditions.

2.2.2.1 Dirichlet condition for a scalar field

For a scalar field, c say, a Dirichlet condition on the boundary $\partial\Omega$ takes the form

$$c = \tilde{c}, \quad \text{on } \partial\Omega.$$

2.2.2.2 Neumann condition for a scalar field

Taking the weak form (applying Green's theorem to the diffusion term) of the advection-diffusion equation (2.1) leads to a surface integral of the form

$$\int_{\partial\Omega} \varphi(\bar{\kappa}\nabla c) \cdot \mathbf{n} \, d\Gamma,$$

where φ is a test function (see section 3). The Neumann condition is specified by assigning a value to $(\bar{\kappa}\nabla c) \cdot \mathbf{n}$, e.g.,

$$(\bar{\kappa}\nabla c) \cdot \mathbf{n} = q, \quad \text{on } \partial\Omega,$$

and substituting in this surface integral to the discretised equation. Note that q is often termed a flux.

2.3 Fluid equations

A starting point for describing the physics of a continuum are the conservation equations. Fluid volumes deform in time as the fluid moves. If $\theta(\mathbf{x}, t)$ is the density of some quantity (e.g., Temperature) associated with the fluid, the time evolution of that quantity in a fluid volume $V(t)$ is

$$\frac{d}{dt} \left[\int_{V(t)} \theta(\mathbf{x}, t) \right] = \int_{V(t)} \left(\frac{D\theta}{Dt} + \theta \nabla \cdot \mathbf{u} \right), \quad (2.9)$$

which is the Reynolds' Transport theorem. In (2.9) $\mathbf{x} = (x, y, z)^T$ and $\mathbf{u} = (u, v, w)^T$ are three-dimensional position and velocity vectors respectively and

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla, \quad (2.10)$$

is the *material derivative* (NB. it has many other commonly-used names including the total and Lagrangian derivative).

2.3.1 Mass conservation

Substituting $\theta = \rho$ in to (2.9) and noting that matter is neither created nor destroyed gives that the left hand side of (2.9) is zero. Then, as the volume $V(t)$ is arbitrary, it is seen that the mass density satisfies

$$\frac{D\rho}{Dt} = -\rho \nabla \cdot \mathbf{u}, \quad (2.11)$$

or equivalently

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0. \quad (2.12)$$

The quantity $\rho \mathbf{u}$ is called the *mass flux* or *momentum* and (2.12) is termed the *equation of continuity*.

2.3.2 Momentum conservation

The momentum associated with a unit volume of fluid is given by $\rho \mathbf{u}$. Initially, consider that the fluid is *ideal*, that is, viscosity and conductivity are assumed to be unimportant. Then, the rate of change of momentum is given by

$$\frac{\partial}{\partial t} (\rho \mathbf{u}) = \rho \frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \rho}{\partial t} \mathbf{u}. \quad (2.13)$$

Using the equation of continuity (2.12) and Euler's equation [Batchelor, 1967] (the force equation for an inviscid fluid) in the form

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla \mathbf{u} - \frac{1}{\rho} \nabla p, \quad (2.14)$$

gives

$$\frac{\partial}{\partial t}(\rho \mathbf{u}) = -\nabla p - \nabla \cdot (\rho \mathbf{u} \mathbf{u}), \quad (2.15)$$

where $\mathbf{u} \mathbf{u}$ is a tensor which represents the dyadic product of vectors which, using index notation, can be written as $u_i u_j$. Writing $\bar{\bar{\Pi}} = p \mathbf{I} + \rho \mathbf{u} \mathbf{u}$, (2.15) can be written as

$$\frac{\partial}{\partial t}(\rho \mathbf{u}) + \nabla \cdot \bar{\bar{\Pi}} = 0, \quad (2.16)$$

where $\bar{\bar{\Pi}}$ is clearly a symmetric tensor and is termed the *momentum flux density tensor*.

The effects of viscosity on the motion of a fluid are now considered. To express the equations of motion governing a viscous fluid, some additional terms are required. The equation of continuity (conservation of mass) is equally valid for any viscous as well as inviscid fluid. However, Euler's equation (2.14) and hence (2.16) require modification.

By adding $-\bar{\bar{\tau}}$ to the previously introduced *momentum flux density tensor*, $\bar{\bar{\Pi}}$, so that

$$\bar{\bar{\Pi}} = p \mathbf{I} + \rho \mathbf{u} \mathbf{u} - \bar{\bar{\tau}} = -\bar{\bar{\sigma}} + \rho \mathbf{u} \mathbf{u}, \quad (2.17)$$

where $\bar{\bar{\sigma}} = -p \mathbf{I} + \bar{\bar{\tau}}$, the viscous transfer of momentum in the fluid can be taken into account. $\bar{\bar{\sigma}}$ is called the stress tensor and gives the part of the momentum flux which is not due to direct transfer of momentum with the mass of the fluid. $\bar{\bar{\tau}}$ is named the deviatoric or viscous stress tensor. These tensors and the forms which they may take are discussed in more detail in section 2.3.3. Thus, the most general form of the momentum equation of a compressible viscous fluid may be written as

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla \cdot \bar{\bar{\sigma}} + \rho \mathbf{F}, \quad (2.18)$$

where \mathbf{F} is a volume force per unit mass (e.g., gravity, astronomical forcing).

2.3.2.1 Compressible equations in conservative form

Using the conservation laws outlined above the following point-wise PDE system governing the motion of a compressible fluid is obtained

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \quad (2.19a)$$

$$\frac{\partial}{\partial t}(\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u} \mathbf{u} - \bar{\bar{\sigma}}) = \rho \mathbf{F}, \quad (2.19b)$$

$$\frac{\partial}{\partial t}(\rho E) + \nabla \cdot (\rho E \mathbf{u} - \bar{\bar{\sigma}} \mathbf{u} + \mathbf{q}) = \rho \mathbf{F} \cdot \mathbf{u}, \quad (2.19c)$$

where $E \equiv e + |\mathbf{u}|^2/2$ is the total specific energy (in which e is the internal energy). (2.19a) is exactly the conservative form of the continuity equation given in (2.12), (2.19b) is equivalent to equation (2.18) and (2.19c) is obtained from making the substitutions $w = e + p/\rho$ and $E \equiv e + |\mathbf{u}|^2/2$ in (2.18).

2.3.2.2 Compressible equations in non-conservative form

Expanding terms in (2.19) yields the non-conservative form of the compressible equations¹

$$\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{u} = 0, \quad (2.20a)$$

$$\rho \frac{D\mathbf{u}}{Dt} - \nabla \cdot \bar{\bar{\sigma}} = \rho \mathbf{F}, \quad (2.20b)$$

$$\rho \frac{De}{Dt} - \bar{\bar{\sigma}} \cdot \nabla \mathbf{u} + \nabla \cdot \mathbf{q} = 0. \quad (2.20c)$$

Note that, provided the fields (*e.g.*, density and pressure) vary smoothly, that is, the fields are differentiable functions, equations (2.19) and (2.20) are identical.

2.3.3 Equations of state & constitutive relations

Closure of the conservation equations requires an additional equation describing how the stress tensor is related to density, temperature and any other state variables of relevance. In general, this relationship is dependent of the physical and chemical properties of the material in the domain; hence, this relationship is known as the material model (note that in multi-material simulations a different material model can and must be specified for each material).

It is convenient to separate the full stress tensor $\bar{\bar{\sigma}}$ into an isotropic (hydrostatic) part, the pressure p , and a deviatoric part $\bar{\bar{\tau}}$. With the convention that compressive stress is negative, the stress tensor is given by $\bar{\bar{\sigma}} = -p\mathbf{I} + \bar{\bar{\tau}}$, where \mathbf{I} is the identity matrix. If the stress tensor is separated in this way, the material model comprises two parts: an equation of state² $f(p, \rho, T, \dots) = 0$ relating density to pressure, temperature, etc., and a constitutive relationship³ $g(\bar{\bar{\tau}}, \mathbf{u}) = 0$.

2.3.3.1 Newtonian fluids

Two important classes of fluids are: (i) Newtonian fluids, where deviatoric strain rate ($\bar{\bar{\varepsilon}}$) is linearly proportional to deviatoric stress ($\bar{\bar{\tau}}$); and (ii) non-Newtonian fluids, where deviatoric strain rate is non-linearly proportional to the deviatoric stress. At present Fluidity is only configured to deal with Newtonian fluids.

For a Newtonian fluid the relation between deviatoric stress and deviatoric strain rate can be expressed as:

$$\bar{\bar{\tau}} = 2\mu\bar{\bar{\varepsilon}} + \lambda(\nabla \cdot \mathbf{u})\mathbf{I}, \quad (2.21)$$

where

$$\bar{\bar{\varepsilon}} = \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T), \quad (2.22)$$

is the deviatoric strain rate tensor, and \mathbf{I} is the identity matrix. μ and λ are the two coefficients of viscosity. Physical arguments yield the so-called Stokes' relationship $3\lambda + 2\mu = 0$, and hence:

$$\bar{\bar{\tau}} = 2\mu(\bar{\bar{\varepsilon}} - (\nabla \cdot \mathbf{u})\mathbf{I}/3), \quad (2.23)$$

where μ is the molecular viscosity. See [Batchelor \[1967\]](#) for further details.

¹(2.20a) is trivial to obtain. (2.20b) makes use of (2.19a) and the divergence of the dyadic product, given by

$$\nabla \cdot (\mathbf{u}\mathbf{u}) = \mathbf{u} \cdot \nabla \mathbf{u} + \mathbf{u} \nabla \cdot \mathbf{u},$$

along with (2.10). (2.20c) makes use of both (2.20a) and (2.20b) and note that substituting for $E \equiv e + |\mathbf{u}|^2/2$ results in the cancellation of kinetic energy terms.

²Equation of state settings in Fluidity are described in section 8.10

³constitutive relationship settings in Fluidity are described in section (to be written)

2.3.3.2 Equation of state for incompressible flow

If a material is *perfectly* incompressible its density cannot change; in other words, the material density is independent of pressure and temperature, giving $\rho = \rho_0$, where ρ_0 is the reference density. Note that a flow may contain multiple incompressible materials of different densities, in which case $\rho^k = \rho_0^k$ applies for each individual material (indexed with the superscript k).

All real materials are compressible to some extent so that changes in pressure and temperature cause changes in density. However, in many physical circumstances such changes in material density are sufficiently small that the assumption of incompressible flow is still valid. If U/L is the order of magnitude of the spatial variation in the velocity field, then the flow field can be considered incompressible if the relative rate of change of density with time is much less than the spatial variation in velocity; *i.e.*, if $\frac{1}{\rho} \frac{D\rho}{Dt} \ll U/L$ then [Batchelor, 1967, p.167].

$$\nabla \cdot \mathbf{u} \approx 0. \quad (2.24)$$

The term incompressible flow is used to describe any such situation where changes in the density of a parcel of material are negligible. Not all parcels in the flow need have the same density; the only requirement is that the density of each parcel remains unchanged. For example, in the ocean where salt content and temperature change with depth, the density of adjacent parcels changes but any one parcel has a constant density Panton [2006]. In such cases it is often important to account for changes in buoyancy caused by the dependence of density on pressure, temperature and composition C (see, for example, section 2.4.4). If the change in ρ, p, T, C about a reference state ρ_0, p_0, T_0, C_0 is small, the dependence of density on each state variable can be assumed to be linear. In this case, a general equation of state takes the form

$$\rho = \rho_0(1 - \alpha(T - T_0) + \beta(C - C_0) + \gamma(p - p_0)), \quad (2.25)$$

where α is the thermal expansion coefficient:

$$\alpha = -\frac{1}{\rho} \frac{\partial \rho}{\partial T},$$

β is a general compositional contraction coefficient

$$\beta = \frac{1}{\rho} \frac{\partial \rho}{\partial C}, \quad (2.26)$$

and γ is the isothermal compressibility

$$\gamma = \frac{1}{\rho} \frac{\partial \rho}{\partial p}. \quad (2.27)$$

For ocean modelling applications the most important compositional variation is salinity S (the volume fraction of salt) and the compressibility of water γ is so small that the pressure dependence can be neglected, giving the simple linear equation of state

$$\rho = \rho_0(1 - \alpha(T - T_0) + \beta(S - S_0)), \quad (2.28)$$

where β is the saline contraction coefficient (not to be confused with the beta plane parameters defined in the section 2.4.1.)

2.3.3.3 Padé equation of state for ocean modelling

Within Fluidity it is also possible to relate the density of sea water to the in situ temperature and salinity through the Padé approximation to the equation of state. This approximation offers an accurate and computationally efficient method for calculating the density of seawater and is described by McDougall et al. [2003].

2.3.4 Momentum boundary conditions

As with the scalar equations discussed previously in section 2.2, any well posed problem requires appropriate boundary conditions for the momentum equation. Possible momentum boundary conditions are discussed below.

2.3.4.1 Prescribed Dirichlet condition for momentum — no-slip as a special case

This condition for momentum is set by simply prescribing all three components of velocity. For example, we might specify an inflow boundary where the normal component of velocity is non-zero, but the two tangential directions are zero. A special case is where all three components are zero and this is referred to as no-slip.

2.3.4.2 Prescribed stress condition for momentum — free-stress as a special case

As with the scalar equation (see section 2.2.2.2), applying Green's theorem to the stress term and the pressure gradient in (2.45a) results in a surface integral of the form

$$\bar{\tau} \cdot \mathbf{n} - p\mathbf{n} = \mathbf{T}, \quad \text{on } \partial\Omega, \quad (2.29)$$

where \mathbf{T} is an applied 'traction' force (actually a force per unit area or stress, it becomes a force when the surface integral in the weak form is performed). An example of this might be were we set the vertical component to zero (in the presence of a free surface) and impose the two tangential directions (*e.g.*, a wind stress).

The free-stress condition is the case where we take $\mathbf{T} \equiv \mathbf{0}$.

2.3.4.3 Traction boundary condition for momentum — free-stress as a special case

In this case the normal component of velocity can be prescribed (*e.g.*, inflow or no-flow ($g = 0$) through the boundary)

$$\mathbf{u} \cdot \mathbf{n} = g, \quad \text{on } \partial\Omega.$$

The remaining two degrees of freedom are imposed by taking the tangential component of (2.29) and specifying the tangential component of the force \mathbf{T}_τ , *i.e.*,

$$\boldsymbol{\tau} \cdot (\bar{\tau} \cdot \mathbf{n} - p\mathbf{n}) = \boldsymbol{\tau} \cdot \bar{\tau} \cdot \mathbf{n} = \mathbf{T}_\tau, \quad \text{on } \partial\Omega,$$

An example of this might be where a rigid lid is used (so normal component is zero) and the tangential components are a prescribed wind stress (in which case we take the two tangential directions to correspond to the available stress or wind velocity information, *i.e.*, east-west and north-south) or bottom drag. Also, what we often term free-slip where the tangential components of stress are set to zero.

2.3.4.4 Free surface boundary condition

The free surface height $\eta \equiv \eta(x, y, t)$ measured from the initial state of the ocean may be modelled either via the integrated continuity equation, or for simplicity here by the kinematic boundary condition which simply states that particles on the free surface remain on it, *i.e.*,

$$\frac{\partial \eta}{\partial t} = - \mathbf{u}_H|_{z=\eta} \cdot \nabla_H \eta + w|_{z=\eta} \quad \text{on } \Omega_{FS}, \quad (2.30)$$

where $\Omega_{FS} \subset \partial\Omega$ is the free surface boundary, $\nabla_H \equiv (\partial/\partial x, \partial/\partial y)^T$, and \mathbf{u}_H is the horizontal component of \mathbf{u} . Using the fact that the normal vector \mathbf{n} at the free surface is $\frac{(-\frac{\partial\eta}{\partial x}, -\frac{\partial\eta}{\partial y}, 1)^T}{\|(-\frac{\partial\eta}{\partial x}, -\frac{\partial\eta}{\partial y}, 1)^T\|}$, equation (2.30) can be reformulated to

$$\frac{\partial\eta}{\partial t} = \frac{\mathbf{u} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{k}}, \quad (2.31)$$

where $\mathbf{k} = (0, 0, 1)$ is the vertical standard basis vector. Note that in spherical geometries \mathbf{k} is replaced with $\mathbf{r} = (\sin\theta \cos\varphi, \sin\theta \sin\varphi, \cos\theta)$ where φ and θ are the azimuthal and co-latitudinal angles respectively.

2.3.4.5 Wetting and drying

If wetting and drying occurs, the free surface boundary condition 2.31 needs to be changed to ensure that the water level η does not sink below the bathymetry level b :

$$\frac{\partial \max(\eta, b + d_0)}{\partial t} = \frac{\mathbf{u} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{k}}, \quad (2.32)$$

where d_0 is a threshold defining the minimum water depth.

The details of the wetting and drying are described in [Funke et al. \[2011\]](#).

2.4 Extensions, assumptions and derived equation sets

In certain scenarios it is desirable to simplify the equations of sections 2.2 and 2.3 according to various approximations. Such approximations can drastically reduce the complexity of the system under consideration while maintaining much of the important physics. In this section we consider approximate forms and assumptions of the conservation equations that are appropriate to different problems.

2.4.1 Equations in a moving reference frame

Newton's second law holds in a fixed inertial reference frame, *i.e.*, fixed with respect to the distant stars. Examples of systems which one may wish to study with boundaries moving with respect to this fixed inertial frame include translating and spinning tanks and the rotating Earth. For these systems it is often convenient to rewrite the underlying equations within the moving frame. Extra terms then need to be considered which account for the fact that the acceleration of a fluid parcel relative to the moving reference frame is different to the acceleration with respect to the fixed inertial frame, and the latter is the one which allows us to invoke Newton's Laws. For useful discussions see [[Batchelor, 1967](#), [Cushman-Roisin, 1994](#), [Gill, 1982](#)].

It is possible to form the equations with respect to a moving reference frame as long as the additional force or acceleration term is included. In the case of a rotating frame that is just by replacing the material derivative in the momentum equation by

$$\frac{D\mathbf{u}}{Dt} + 2\boldsymbol{\Omega} \times \mathbf{u}, \quad (2.33)$$

where $\boldsymbol{\Omega}$ is the angular velocity vector of the rotating system.

Consider the Earth with a rotation vector in an inertial reference frame given by

$$\boldsymbol{\Omega} = (0, 0, \Omega)^T. \quad (2.34)$$

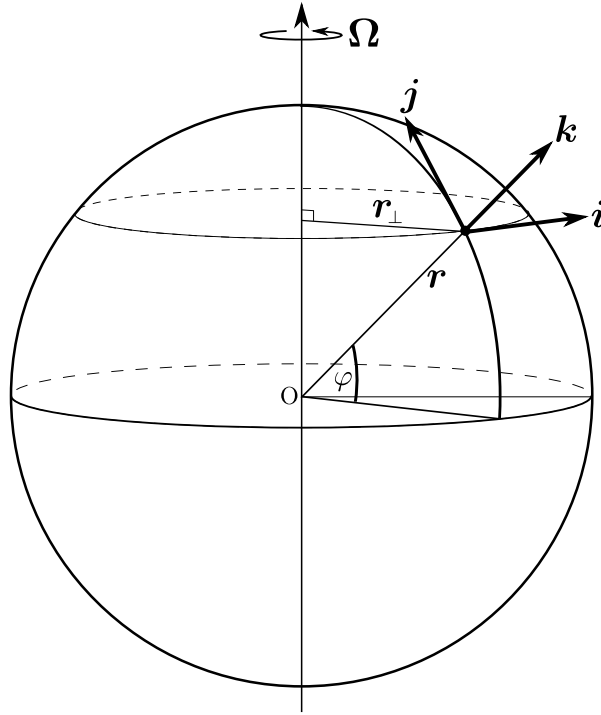


Figure 2.1: Schematic of coordinates in a frame rotating with a sphere. The rotation is about a vector pointing from South to North pole. A point on the surface of the sphere x and its perpendicular distance from the axes of rotation x_{\perp} are shown. The latitude of x is given by φ and the unit vectors i , j and k represent a local coordinate axes at a point x in the rotating frame: i points Eastwards, j points Northwards and k points in the radial outwards direction.

In a local rotating frame of reference where the x -axis is oriented Eastwards, the y -axis is oriented Northwards and the z -axis is the local upwards direction, the Earth's rotation vector is expressed as

$$\boldsymbol{\Omega} = \Omega \cos \varphi \mathbf{j} + \Omega \sin \varphi \mathbf{k} \equiv \Omega(0, \cos \varphi, \sin \varphi)^T,$$

where φ is the latitude. The acceleration terms in the three momentum equations now have the form

$$\begin{aligned} \frac{Du}{Dt} + 2\Omega \cos \varphi w - 2\Omega \sin \varphi v, \\ \frac{Dv}{Dt} + 2\Omega \sin \varphi u, \\ \frac{Dw}{Dt} - 2\Omega \cos \varphi v. \end{aligned}$$

2.4.1.1 The 'traditional' approximation

Define the Coriolis and reciprocal Coriolis parameters [Cushman-Roisin, 1994] respectively by

$$f = 2\Omega \sin \varphi, \quad f^* = 2\Omega \cos \varphi. \quad (2.35)$$

Due to dimensional considerations it is usual to drop the f^* term and hence simply assume that

$$\boldsymbol{\Omega} = (0, 0, f/2)^T, \quad (2.36)$$

in the local frame of reference, *i.e.*, only to consider the locally vertical component of the rotation vector. This approximation, when taken along with the assumption of hydrostatic balance in the vertical constitutes what is generally known as the traditional approximation in geophysical fluid dynamics.

2.4.1.2 The f -plane and β -plane approximations

If the Coriolis parameter f is approximated by a constant value:

$$f = f_0, \quad (2.37)$$

this is termed the f -plane approximation, where $f_0 = 2\Omega \sin \varphi_0$ at a latitude φ_0 . This is obviously only an applicable approximation in a domain of interest that does not have large extent in latitude.

For slightly larger domains a more accurate approximation is to use

$$f = f_0 + \beta y, \quad (2.38)$$

where y is the local coordinate in the Northwards direction. Taking $\varphi = \varphi_0 + y/R_E$ and expanding (2.35) in a Taylor series yields

$$f = f_0 + 2\Omega \cos \varphi_0 \frac{y}{R_E} + \dots, \quad \beta = \frac{2\Omega}{R_E} \cos \varphi_0,$$

where $R_E \approx 6378$ km. Typical values of these terms are:

$$\Omega = \frac{2\pi}{24 \times 60 \times 60} = 7.2722 \times 10^{-5} \text{ rad s}^{-1},$$

(NB. a sidereal day should be used to give a more accurate value of $7.2921 \times 10^{-5} \text{ rad s}^{-1}$) and

	$\varphi_0 = 30$	$\varphi_0 = 45$	$\varphi_0 = 60$
f_0	7.2722e-05	1.0284e-04	1.2596e-04
β	1.9750e-11	1.6124e-11	1.1402e-11

2.4.2 Linear Momentum

Newton's second law states that the sum of forces applied to a body is equal to the time derivative of linear momentum of the body,

$$\sum \mathbf{f} = d(m\mathbf{u})/dt. \quad (2.39)$$

Applying this law to a control volume⁴ of fluid and making use of equation 2.9 leads to the linear momentum equation for a fluid which can be written as

$$\frac{\partial}{\partial t} \int_V \rho \mathbf{u} dV = - \int_S \mathbf{u} \rho \mathbf{u} \cdot \mathbf{n} dA + \int_V \mathbf{F} \rho dV + \int_S \bar{\bar{\sigma}} \cdot \mathbf{n} dA, \quad (2.40)$$

where V is the control volume, S is the surface of the control volume and \mathbf{n} is the unit normal to the surface of the control volume and \mathbf{F} is a volume force per unit mass. Physically, 2.40 states that the sum of all forces applied on the control volume is equal to the sum of the rate of change of momentum inside the control volume and the net flux of momentum through the control surface. More details regarding the derivation and properties of this equation can be found in Batchelor [1967].

2.4.3 Buoyancy and Hydrostacy

For fluids upon which gravity is acting the buoyancy force should be considered when a free surface is present or when the fluid contains variations in density. The buoyancy force is given by $\mathbf{b} = -\rho\mathbf{g}$ and, in the absence of viscosity, a simple form of the vertical momentum equation can be written as

$$\frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + b, \quad (2.41)$$

⁴The concept of a control volume and how they are defined within fluidity is discussed more in section 3.2.4

where $b = \rho g$ is the magnitude of the buoyancy force.

If the fluid is in a state of rest, the left hand side of (2.41) is zero giving

$$\frac{\partial p}{\partial z} = b. \quad (2.42)$$

This is known as hydrostatic balance. It states that the pressure at point is equal to the weight of water above that point, plus any pressure loading on the surface of the domain (e.g., atmospheric pressure or ice load which is here termed p_a):

$$p(\mathbf{x}) = p_a + \int_z b.$$

If vertical accelerations are negligible then (2.42) is often a good approximation to (2.41). Under this assumption it is natural to define pressure in terms of a background pressure that is only dependent on depth and a perturbation to this:

$$p = p_0(z, t) + p'(\mathbf{x}, t),$$

where $p_0(z, t)$ balances the constant ρ_0 part of buoyancy and may be written

$$p_0(z, t) = \int_z^\eta \rho_0 g = \rho_0 g(\eta - z),$$

where $\eta \equiv \eta(\mathbf{x})$ is the free surface height. If a free surface is present an additional term of the form

$$-\rho_0 g \nabla \eta,$$

must therefore be included in the horizontal momentum equations.

2.4.4 The Boussinesq approximation

Under certain conditions, one is able to assume that density does not vary greatly about a mean reference density, that is, the density at a position \mathbf{x} can be written as

$$\rho(\mathbf{x}, t) = \rho_0 + \rho'(\mathbf{x}, t), \quad (2.43)$$

where $\rho' \ll \rho_0$. Such an approximation, namely, the Boussinesq approximation, involves two steps. The first makes use of (2.24) — mass conservation thus becomes volume conservation and sound waves are filtered. The second part of the Boussinesq approximation follows by replacing ρ by ρ_0 in all terms of (2.18), except where density is multiplied by gravity (i.e. in the buoyancy term where full density must be retained — these are the density variations that drive natural convection). This yields

$$\rho_0 \frac{D\mathbf{u}}{Dt} - \nabla \cdot \bar{\bar{\sigma}} = \rho \mathbf{g} + \rho_0 \mathbf{F}, \quad (2.44)$$

where buoyancy has explicitly been removed from the forcing term \mathbf{F} and \mathbf{g} is the gravitational vector (e.g. $\mathbf{g} = -g\mathbf{k}$ in planar problems when gravity points in the negative z direction and $\mathbf{g} = -g\mathbf{r}$ on the sphere).

2.4.4.1 The non-hydrostatic Boussinesq equations

Applying the approximations outlined in section 2.4.4 to (2.18) and (2.12), along with scalar transport equations for salinity and temperature (see (2.1)) and an appropriate equation of state (see section 2.3.3), the three-dimensional non-hydrostatic Boussinesq equations can be written as

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + 2\boldsymbol{\Omega} \times \mathbf{u} = -\nabla p' - g\nabla\eta + \rho' \mathbf{g} + \nabla \cdot \bar{\bar{\tau}} + \mathbf{F}, \quad (2.45a)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (2.45b)$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \nabla \cdot (\bar{\bar{\kappa}}_T \nabla T), \quad (2.45c)$$

$$\frac{\partial S}{\partial t} + \mathbf{u} \cdot \nabla S = \nabla \cdot (\bar{\bar{\kappa}}_S \nabla S), \quad (2.45d)$$

$$f(p, \rho, T, \dots) = 0, \quad (2.45e)$$

where p' is the perturbation pressure (see section 2.4.3), $\rho = \rho_0 + \rho'$ where $\rho' (= (\rho - \rho_0)/\rho_0)$ is the perturbation density, T is the temperature, S is salinity, η is the free surface height and $\bar{\bar{\tau}}, \bar{\bar{\kappa}}_T, \bar{\bar{\kappa}}_S$ are the viscosity, thermal diffusivity and saline diffusivity tensors respectively. The rotation vector is $\boldsymbol{\Omega}$, and \mathbf{F} contains additional source terms such as the astronomical tidal forcing. A discussion regarding the validity of the Boussinesq approximation is given in Gray [1976]

2.4.5 Supplementary boundary conditions and body forces

2.4.5.1 Bulk parameterisations for oceans

In order to simulate real-world ocean scenarios, realistic boundary conditions for the momentum, freshwater and heat fluxes must be applied to the upper ocean surface. Fluidity can apply such boundary conditions in the form of the bulk formulae of Large and Yeager [2004], COARE 3.0 [Fairall et al., 2003] and Kara et al. [2005] in combination with the ERA-40 reanalysis data [Uppala et al., 2005].

Three surface kinematic fluxes calculated: heat – $\langle w\theta \rangle$, salt – $\langle ws \rangle$, and momentum – $\langle wu \rangle$ and $\langle wv \rangle$, which can be related to the surface fluxes of heat Q , the freshwater F , and the momentum $\vec{\tau} = (\tau_u, \tau_v)$, via:

$$\langle w\theta \rangle = Q (\rho C_p)^{-1} \quad (2.46)$$

$$\langle ws \rangle = F (\rho^{-1} S_0) \quad (2.47)$$

$$(\langle wu \rangle, \langle wv \rangle) = \vec{\tau} \rho^{-1} = (\tau_u, \tau_v) \rho^{-1} \quad (2.48)$$

where ρ is the ocean density, C_p is the heat capacity ($4000 \text{ JkS}^{-1}\text{K}^{-1}$) and S_0 is a reference ocean salinity, which is the current sea surface salinity. These fluxes are then applied as upper-surface Neumann boundary conditions on the appropriate fields.

2.4.5.2 Co-oscillating boundary tides

Boundary tides can be applied to open ocean domain boundaries through setting a Dirichlet condition on the non-hydrostatic part of the pressure. Co-oscillating tides are forced as cosine waves of specified phase and amplitude along designated boundaries:

$$h = \sum_i A_i \cos(\sigma_i t - \varphi_i) + \sum_j A_j \cos(\sigma_j t - \varphi_j) + \sum_k A_k \cos(\sigma_k t - \varphi_k), \quad (2.49)$$

where h is the free surface height (m), A is the amplitude of the tidal constituent (m), t is the time (s), φ is the phase of the tidal constituent (radians) [Wells, 2008].

The nature of the co-oscillating tide can take the form of either one fixed cosine wave of constant amplitude and phase applied across the entire length of the boundary, or it can be variable as delimited via an interpolation of different amplitudes and phases at a series of points spread along the boundary [Wells, 2008]. Within Fluidity, co-oscillating boundary tides can be applied through *e.g.*, the FES2004 data set (see 10.11).

2.4.5.3 Astronomical tides

Astronomical forcing can also be applied to a fluid as a body force⁵. The astronomical tidal potential is calculated at each node of the finite element mesh using the multi-constituent equilibrium theory of tides equation:

$$\begin{aligned} \eta_{eq}(\lambda, \theta, t) &= \sin^2 \theta \sum_i A_i \cos(\sigma_i t + \chi_i + 2\lambda) \\ &+ \sin 2\theta \sum_j A_j \cos(\sigma_j t + \chi_j + \lambda) \\ &+ (3 \sin^2 \theta - 2) \sum_k A_k \cos(\sigma_k t + \chi_k), \end{aligned} \quad (2.50)$$

where η_{eq} is the equilibrium tidal potential (m), λ is the east longitude (radians), θ is the colatitude $[(\pi/2) - \text{latitude}]$, χ is the astronomical argument (radians), σ is the frequency of the tidal constituent (s^{-1}), t is universal standard time (s) and A is the equilibrium amplitude of the tidal constituent (m). Subscript i represents the semidiurnal constituents (e.g. M_2), subscript j the diurnal constituents (e.g., K_1) and subscript k the long period constituents (e.g., M_t ; Wells, 2008). The overall forcing is applied as the product of the gradient of the resulting equilibrium tidal potential and the acceleration due to gravity (g) [Mellor, 1996, Kantha and Clayson, 2000, Wells et al., 2007]. The multi-constituent equilibrium theory of tides is flexible in that it enables astronomical tides to be forced as individual constituents (e.g. M_2 or S_2) or as a combination of different constituents (e.g. $M_2 + S_2$) [Wells, 2008].

As there is no interest in calculating the tide for an exact date, the astronomical argument is typically excluded from the multi-constituent theory of tides equation for ICOM applications meaning that all satellites start at 0° latitude [Wells, 2008].

The astronomical tidal potential can be modified to account for the deformation of the solid Earth (the body tide) if desired. The multi-constituent equilibrium theory of tides equation includes the effects of the solid Earth deformation, adding this to the overall free surface height. This is fine when validating model results against measurements that record the overall elevation of the Earth's oceans (e.g. satellite altimeter readings and surface tide gauges). If however, the model is validated against measurements that do not include the effects of the Earth's body tide such as with pelagic pressure gauges, then a correction to the equilibrium tidal potential is required. This can be applied as:

$$\eta = (1 + k - h)\eta_{eq}, \quad (2.51)$$

where η is the corrected tidal potential, η_{eq} is the uncorrected equilibrium tidal potential and k (0.3) and h (0.61) are Love numbers (after Love, 1909). Both k and h are dimensionless measures of the elastic behaviour of the solid Earth. k accounts for the enhancement to the Earth's gravitational potential brought about by the re-distribution of the Earth's mass whereas h is a correction for the physical distortion to the Earth's surface [Pugh, 1987].

The exact values of k and h vary for different tidal constituents and the numbers shown are given for the semidiurnal M_2 constituent. Typical variations to k and h are less than 0.01 which corresponds to a <1% change to the equilibrium tidal potential. These errors are sufficiently small that the stated values are a suitable approximation to use in body tide corrections to the majority of tidal constituents.

2.4.6 Multi-material simulations

The ability to differentiate between regions with distinct material properties is of fundamental importance in the modelling of many physical systems. Two different approaches exist for achieving

⁵Note that use of the word body force in this context should not be confused with the application of a body force through the options tree discussed later in 8. The astronomical forcing is applied separately.

this: the multi-material approach and the multi-phase approach. The multi-material approach is implemented within Fluidity, and the multi-phase approach (discussed in the next subsection) is currently under development. In situations where the model can resolve physical mixing of immiscible materials, or where there is no mixing, only one velocity field (and hence one momentum equation) is required to describe the flow of all materials. The *multi-material* approach, considers all materials to be immiscible materials separated by a sharp interface.

In a multi-material approach, the various forms of the conservation equations can be solved for multiple material flows if the point-wise mass density ρ in the equations is defined as the bulk density at each point. If the flow comprises n materials and the volume fraction of the i^{th} material is denoted φ_i then the bulk density is given by:

$$\rho = \sum_{i=1}^n \varphi_i \rho_i \quad (2.52)$$

where ρ_i is the density of each material. For incompressible materials $\rho_i = \rho_{i0}$; for materials whose density is defined by an equation of state (see section 2.3.3) $\rho_i = f(p, T, S, \dots)$. Conservation of mass at each point also requires that

$$\sum_{i=1}^n \varphi_i = 1. \quad (2.53)$$

In an n -material problem, the multi-material approach requires that $n - 1$ advection equations are solved, to describe the transport of the volume fraction of all but one of the materials. The volume fraction of the remaining material can be derived from the other volume fractions by

$$\varphi_n = 1 - \sum_{i=1}^{n-1} \varphi_i. \quad (2.54)$$

The transport of the i^{th} volume fraction is given by

$$\frac{\partial \varphi_i}{\partial t} + \mathbf{u} \cdot \nabla \varphi_i = 0, \quad (2.55)$$

where the volume fraction field at time zero must be specified.

2.4.7 Multi-phase simulations

Multi-phase flows are defined by Prosperetti and Tryggvason [2007] to be flows in which two or more phases of matter (solid, liquid, gas, etc) are simultaneously present and are allowed to inter-penetrate. Simple examples include the flow of a fizzy drink which is composed of a liquid and a finite number of gas bubbles, the transportation of solid sediment particles in a river, and the flow of blood cells around the human body.

Further to the above definition, each phase is classed as either *continuous* or *dispersed*, where a continuous phase is a connected liquid or gas substance in which dispersed phases (comprising a finite number of solid particles, liquid droplets and/or gas bubbles) may be immersed [Crowe et al., 1998].

To enable the mixing and inter-penetration of phases, a separate velocity field (and hence a separate momentum equation) is assigned to each one and solved for. Extra terms are then included to account for inter-phase interactions. Furthermore, the model currently assumes no mass transfer between phases, incompressible flow, and a common pressure field p so that only one continuity equation is used. Thus, the continuity equation and momentum equation for phase i (based on the derivation in Ishii [1975], written in non-conservative form) are:

$$\sum_{i=1}^N \nabla \cdot (\alpha_i \mathbf{u}_i) = 0, \quad (2.56)$$

$$\alpha_i \rho_i \frac{\partial \mathbf{u}_i}{\partial t} + \alpha_i \rho_i \mathbf{u}_i \cdot \nabla \mathbf{u}_i = -\alpha_i \nabla p + \alpha_i \rho_i \mathbf{g} + \nabla \cdot (\alpha_i \mu_i \nabla \mathbf{u}_i) + \mathbf{F}_i, \quad (2.57)$$

where \mathbf{u}_i , ρ_i , μ_i and α_i are the velocity, density, isotropic viscosity and volume fraction of phase i respectively, and \mathbf{F}_i represents the forces imposed on phase i by the other $N - 1$ phases. Details of momentum transfer terms are given in Chapter 8.

Chapter 3

Numerical discretisation

3.1 Introduction & some definitions

This chapter covers the numerical discretisation of the model equations given in the previous chapter as they are assembled and solved in Fluidity. For a more general introduction into finite element theory we refer to a.o. [Elman et al. \[2005\]](#) and [Gresho and Chan \[1988\]](#).

In this chapter we define the domain we are solving over as Ω . The boundary to Ω is defined as $\partial\Omega$ and can be split into different sections, e.g. $\partial\Omega = \partial\Omega^N \cup \partial\Omega^D$ where $\partial\Omega^N$ is that part of the boundary over which Neumann conditions are applied and $\partial\Omega^D$ is that part of the boundary over which Dirichlet conditions are applied. For clarity a subscript showing the field in question will be given on the N and D characters.

The unit vector \mathbf{n} is always assumed to be the outward facing normal vector to the domain. In the following this notation is used to describe both the normal vector at the boundary and between elements in the interior.

3.2 Spatial discretisation of the advection-diffusion equation

The advection-diffusion equation for a scalar tracer c , is given in conservative form by

$$\frac{\partial c}{\partial t} + \nabla \cdot (\mathbf{u}c) - \nabla \cdot \bar{\bar{\kappa}} \cdot \nabla c = 0. \quad (3.1)$$

We now define a partition of the boundary such that $\partial\Omega = \partial\Omega^N \cup \partial\Omega^D$, and impose boundary conditions on c :

$$\mathbf{n} \cdot \bar{\bar{\kappa}} \cdot \nabla c = g_{N_c} \quad \text{on} \quad \partial\Omega^{N_c}, \quad (3.2)$$

$$c = g_{D_c} \quad \text{on} \quad \partial\Omega^{D_c}. \quad (3.3)$$

3.2.1 Continuous Galerkin discretisation

The Continuous Galerkin method (often abbreviated to CG, not to be confused with Conjugate Gradients) method, is a widely used finite element method in which the solution fields are constrained to be continuous between elements. The fields are only assumed to be C^0 continuous, that is to say there is no assumption that the gradient of a field is continuous over element boundaries.

3.2.1.1 Weak form

Development of the finite element method begins by writing the equations in *weak form*. The weak form of the advection-diffusion equation (here presented in conservative form) is obtained by pre-multiplying it with a test function φ and integrating over the domain Ω , such that

$$\int_{\Omega} \varphi \left(\frac{\partial c}{\partial t} + \nabla \cdot (\mathbf{u}c) - \nabla \cdot \bar{\kappa} \cdot \nabla c \right) = 0. \quad (3.4)$$

Integrating the advection and diffusion terms by parts yields

$$\int_{\Omega} \varphi \frac{\partial c}{\partial t} - \nabla \varphi \cdot \mathbf{u}c + \nabla \varphi \cdot \bar{\kappa} \cdot \nabla c + \int_{\partial\Omega} \varphi (\mathbf{n} \cdot \mathbf{u}c - \mathbf{n} \cdot \bar{\kappa} \cdot \nabla c) = 0. \quad (3.5)$$

For simplicity sake let us first assume the boundaries are closed ($\mathbf{u} \cdot \mathbf{n} = 0$) and we apply a homogeneous Neumann boundary condition everywhere, such that $\partial c / \partial n = 0$, which gives

$$\int_{\Omega} \varphi \frac{\partial c}{\partial t} - \nabla \varphi \cdot \mathbf{u}c + \nabla \varphi \cdot \bar{\kappa} \cdot \nabla c = 0. \quad (3.6)$$

Note that due to the Neumann boundary condition the boundary term has dropped out. The tracer field c is now called a *weak solution* of the equations if (3.6) holds true for all φ in some space of test functions V . The choice of a suitable test space V is dependent on the equation (for more details see [Elman et al. \[2005\]](#)).

An important observation is that (3.5) only contains first derivatives of the field c , so that we can now include solutions that do not have a continuous second derivative. For these solutions the original equation (2.45c) would not be well-defined. These solutions are termed *weak* as they do not have sufficient smoothness to be *classical* solutions to the problem. All that is required for the weak solution is that the first derivatives of c can be integrated along with the test function. A more precise definition of this space, the *Sobolev space*, can again be found [Elman et al. \[2005\]](#).

3.2.1.2 Finite element discretisation

Instead of looking for a solution in the entire function (Sobolev) space, in finite element methods discretisation is performed by restricting the solution to a finite-dimensional subspace. Thus the solution can be written as a linear combination of a finite number of functions, the *trial functions* φ_i that form a basis of the *trial space*, defined such that

$$c(\mathbf{x}) = \sum_i c_i \varphi_i(\mathbf{x}).$$

The coefficients c_i can be written in vector format. The dimension of this vector equals the dimension of the trial space. In the sequel any function in this way represented as a vector will be denoted as \underline{c} .

Since the set of trial functions is now much smaller (or rather finite as opposed to infinite-dimensional), we also need a much reduced set of test functions for the equation in weak form (3.6) in order to find a unique solution. A common choice is, in fact, to choose the same test and trial space. Finite element methods that make this choice are referred to as *Galerkin methods* — the discretisation can be seen as a so called *Galerkin projection* of the weak equation to a finite subspace.

There are many possibilities for choosing the finite-dimensional trial and test spaces. A straightforward choice is to restrict the functions to be polynomials of degree $n \leq N$ within each element. These are referred to as P_N discretisations. As we generally need functions for which the first derivatives are integrable, a further restriction is needed. If we allow the functions to be any polynomial of degree $n \leq N$ within the elements the function can be discontinuous in the boundary between elements. Continuous Galerkin methods therefore restrict the test and trial functions to arbitrary

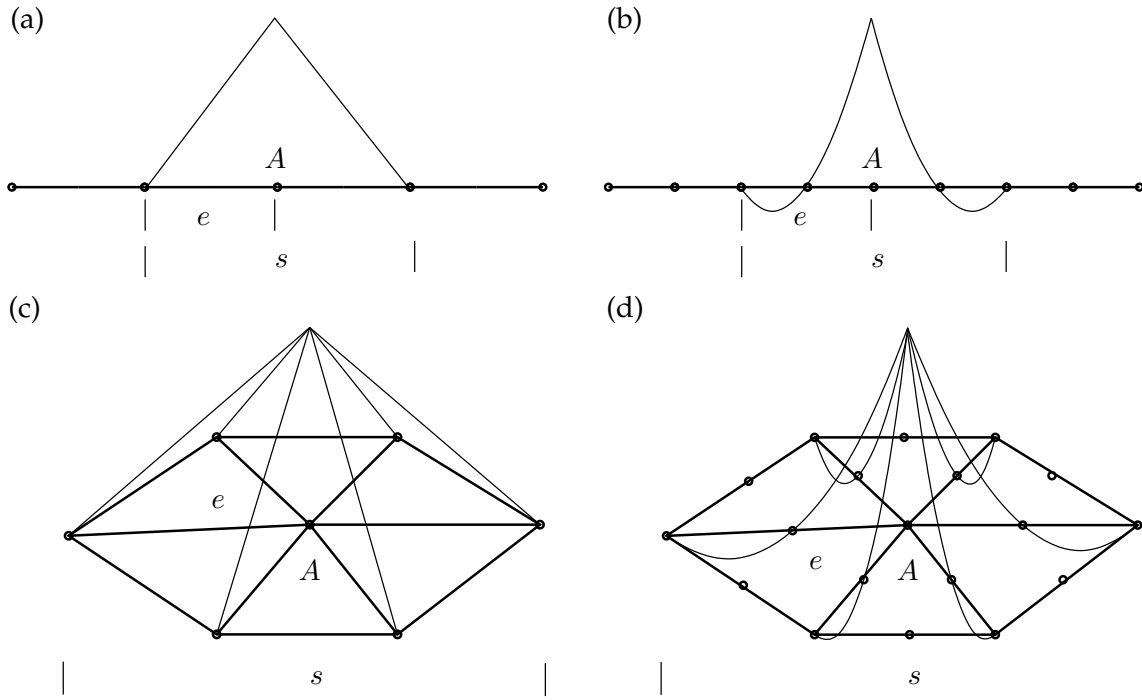


Figure 3.1: One-dimensional (a, b) and two-dimensional (c, d) schematics of piecewise linear (a, c) and piecewise quadratic (b, d) continuous shape functions. The shape function has value 1 at node A descending to 0 at all surrounding nodes. The number of nodes per element, e , depends on the polynomial order while the support, s , extends to all the elements surrounding node A .

polynomials that are continuous between the elements. Discontinuous Galerkin methods, that allow any polynomial, are also possible but require extra care when integrating by parts (see section 3.2.3).

If we choose P_1 as our test and trial functions, *i.e.*, piecewise linear functions, within each element we only need to know the value of the function at 3 points in 2D, and 4 points in 3D. In Fluidity these points are chosen to be the vertices of the triangles (in 2D) or tetrahedra (in 3D) tessellating the domain. For continuous Galerkin the continuity requirement then comes down to requiring the functions to have a single value at each vertex. A set of basis functions φ_i for this space is easily found by choosing the piecewise linear functions φ_i that satisfy:

$$\begin{aligned}\varphi_i(x_i) &= 1, \quad \forall i \\ \varphi_i(x_{j \neq i}) &= 0, \quad \forall i, j,\end{aligned}$$

where x_i are the vertices in the mesh. This choice of basis functions has the following useful property:

$$c_i = c(x_i), \quad \text{for all nodes } x_i \text{ in the mesh.}$$

This naturally describes trial functions that are linearly interpolated between the values c_i in the nodes. Higher order polynomials can be represented using more nodes in the element (see Figure 3.1).

As discussed previously the test space in Galerkin finite element methods is the same as the trial space. So for P_N the test functions can be an arbitrary linear combination of the same set of basis functions. To make sure that the equation we are solving integrates to zero with all such test functions, all we have to do is make sure that the equation tested with the basis functions integrate to zero. The discretised version of (3.6) therefore becomes

$$\sum_j \left\{ \int_{\Omega} \varphi_i \varphi_j \frac{dc_j}{dt} - \nabla \varphi_i \cdot \mathbf{u} \varphi_j c_j + \nabla \varphi_i \cdot \bar{\kappa} \cdot \nabla \varphi_j c_j \right\} = 0, \quad \text{for all } \varphi_i. \quad (3.7)$$

where we have substituted $c = \sum_j \varphi_j c_j$. From this it is readily seen that we have in fact obtained a matrix equation of the following form

$$M \frac{dc}{dt} + A(\mathbf{u})c + Kc = 0, \quad (3.8)$$

where M, A and K are the following matrices

$$M_{ij} = \int_{\Omega} \varphi_i \varphi_j, \quad A_{ij} = - \int_{\Omega} \nabla \varphi_i \cdot \mathbf{u} \varphi_j, \quad K_{ij} = \int_{\Omega} \nabla \varphi_i \cdot \bar{\kappa} \cdot \nabla \varphi_j. \quad (3.9)$$

3.2.1.3 Advective stabilisation for CG

It is well known that a continuous Galerkin discretisation of an advection-diffusion equation for an advection dominated flow can suffer from over- and under-shoots which are qualitatively incorrect errors. Furthermore, these overshoot errors are not localised: they can propagate throughout the simulation domain and pollute the global solution [Hughes, 1987]. Consider a simple 1D linear steady-state advection-diffusion problem for a single scalar c with a source term f :

$$u \frac{\partial c}{\partial x} - \kappa \frac{\partial^2 c}{\partial x^2} = f(x), \quad (3.10)$$

or equivalently in weak form:

$$\int_{\Omega} \left\{ \varphi \left(u \frac{\partial c}{\partial x} - f \right) + \kappa \frac{\partial \varphi}{\partial x} \frac{\partial c}{\partial x} \right\} = 0, \quad (3.11)$$

where we have integrated by parts and applied the natural Neumann boundary condition $\partial c / \partial x = 0$ on $\partial\Omega$. Discretising (3.11) with a continuous Galerkin method leads to truncation errors in the advection term equivalent to a negative diffusivity term of magnitude [Donea and Huerta, 2003]:

$$\bar{\kappa} = \xi \kappa \text{Pe}, \quad (3.12)$$

where:

$$\xi = \frac{1}{\tanh(\text{Pe})} - \frac{1}{\text{Pe}}, \quad (3.13)$$

and:

$$\text{Pe} = \frac{uh}{2\kappa}, \quad (3.14)$$

is a grid Péclet number, with grid spacing h .

This implicit negative diffusivity becomes equal to the explicit diffusivity at a Péclet greater than one, and hence instability can occur for $\text{Pe} \geq 1$. In order to achieve a stable discretisation using a continuous Galerkin method one is therefore required either to increase the model resolution so as to reduce the grid Péclet number, or to apply advective stabilisation methods.

Balancing diffusion A simple way to stabilise the system is to add an extra diffusivity of equal magnitude to that introduced by the discretisation of the advection term, but of opposite sign. This method is referred to as *balancing diffusion*. Note, however, that for two or more dimensions, we require this balancing diffusion to apply in the along-stream direction only [Brooks and Hughes, 1982, Donea and Huerta, 2003]. For this reason this method is also referred to as *streamline-upwind* stabilisation. The multi-dimensional weak-form (assuming we consider non-conservative or advective form) of equation (3.10) is:

$$\int_{\Omega} \left\{ \varphi (\mathbf{u} \cdot \nabla c - f) + \nabla \varphi \cdot \bar{\kappa} \cdot \nabla c - f \right\} = 0, \quad (3.15)$$

is therefore modified to include an additional artificial balancing diffusion term [Donea and Huerta, 2003]:

$$\int_{\Omega} \varphi(\mathbf{u} \cdot \nabla c + \bar{\kappa} \nabla \varphi \cdot \nabla c - f(x)) + \int_{\Omega} \frac{\bar{\kappa}}{\|\mathbf{u}\|^2} (\mathbf{u} \cdot \nabla \varphi)(\mathbf{u} \cdot \nabla c) = 0. \quad (3.16)$$

The exact form of the multidimensional stability parameter $\bar{\kappa}$ is a research issue. See 3.2.1.3 for implementations in Fluidity.

The addition of the balancing diffusion term combats the negative implicit diffusivity of the continuous Galerkin method. However, we are no longer solving the original equation – for pure advection we are now solving a modified version of the original equation with the grid Péclet number artificially reduced from infinity to unity everywhere. Hence streamline-upwind is not a *consistent* stabilisation method, and there can be a reduction in the degree of numerical convergence.

Streamline-upwind Petrov-Galerkin (SUPG) The streamline-upwind stabilisation method can be extended to a consistent (and hence high order accurate) method by introducing stabilisation in the form of a weighted residual [Donea and Huerta, 2003]:

$$\int_{\Omega} \varphi(\mathbf{u} \cdot \nabla c + \bar{\kappa} \nabla \varphi \cdot \nabla c - f(x)) + \int_{\Omega} \tau P(\varphi) R(\varphi) = 0,$$

where:

$$R(\varphi) = \mathbf{u} \cdot \nabla c - \nabla \cdot \bar{\kappa} \nabla c - f(x),$$

is the equation residual, τ is a stabilisation parameter and $P(\varphi)$ is some operator. Note that this is equivalent to replacing the test function in the original equation with $\tilde{\varphi} = \varphi + \tau P(\varphi)$. Looking at equation (3.16), it can be seen that the balancing diffusion term is equivalent to replacing the test function for the advection term only with:

$$\tilde{\varphi} = \varphi + \frac{\kappa}{\|\mathbf{u}\|^2} \mathbf{u} \cdot \nabla. \quad (3.17)$$

This suggests a stabilisation method whereby the test function in the advection-diffusion equation is replaced with the test function in (3.17). This approach defines the *streamline-upwind Petrov-Galerkin* (SUPG) method. The weighted residual formulation of this method guarantees consistency, and hence preserves the accuracy of the method. Furthermore, while this method can still possess under- and over-shoot errors in the presence of sharp solution gradients, these errors remain localised [Hughes, 1987].

Stabilisation parameter Note that, as mentioned in 3.2.1.3, the choice of stabilisation parameter $\bar{\kappa}$ is somewhat arbitrary. Fluidity implements [Brooks and Hughes, 1982, Donea and Huerta, 2003]:

$$\bar{\kappa} = \frac{1}{2} \sum \xi_i u_i h_i, \quad (3.18)$$

where ξ is defined in (3.13) and the summation is over the quadrature points of an individual element. The grid spacings h_i are approximated from the elemental Jacobian.

As an alternative, Raymond and Garder [1976] show that for 1D transient pure-advection a choice of:

$$\bar{\kappa} = \frac{1}{\sqrt{15}} \sum \xi_i u_i h_i, \quad (3.19)$$

minimises phase errors.

Computing the ξ factor at quadrature points is potentially expensive due to the evaluation of a hyperbolic-tangent (3.13). Sub-optimal but more computationally efficient approximations for ξ are the *critical rule* approximation [Brooks and Hughes, 1982]:

$$\xi = \begin{cases} -1 - 1/\text{Pe} & \text{Pe} < -1 \\ 0 & -1 \leq \text{Pe} \leq 1 \\ 1 + 1/\text{Pe} & \text{Pe} > 1, \end{cases} \quad (3.20)$$

and the *doubly-asymptotic* approximation [Donea and Huerta, 2003]:

$$\xi = \begin{cases} \text{Pe}/3 & |\text{Pe}| \leq 3 \\ \text{sgn}(\text{Pe}) & \text{otherwise.} \end{cases} \quad (3.21)$$

Implementation limitations The SUPG implementation in Fluidity does not modify the test function derivatives or the face test functions. Hence the SUPG implementation is only consistent for degree one elements with no non-zero Neumann or weak Dirichlet boundary conditions.

SUPG is considered ready for production use for scalar advection-diffusion equation discretisation, but is still experimental for momentum discretisation.

3.2.1.4 Example

The following example considers pure advection of a 1D top hat of unit magnitude and width 0.25 in a periodic domain of unit size. The top hat is advected with a Courant number of 1/8. Figure 3.2 shows the solution after 80 timesteps using a continuous Galerkin discretisation. Figure 3.3 shows the solution when streamline-upwind stabilisation is applied. Figure 3.4 shows the solution when streamline-upwind Petrov-Galerkin is applied, using a stabilisation parameter as in (3.18).

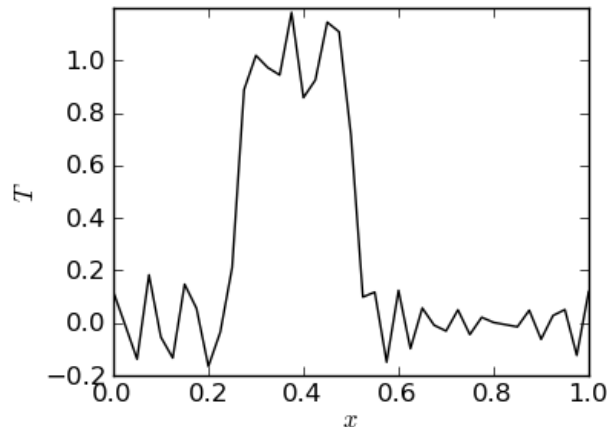


Figure 3.2: Pure advection of a 1D top hat function in a periodic domain at CFL number 1/8 after 80 timesteps using a continuous Galerkin discretisation.

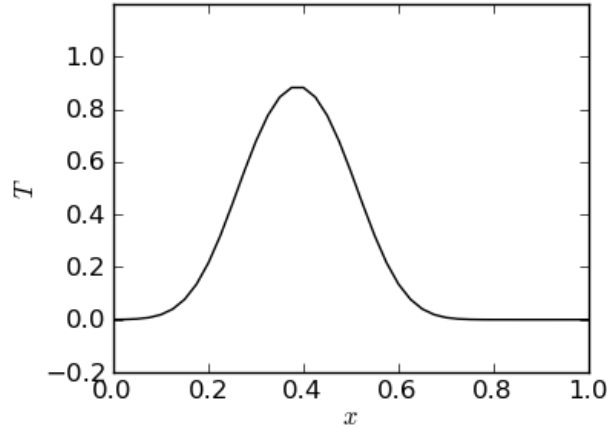


Figure 3.3: Pure advection of a 1D top hat function in a periodic domain at CFL number 1/8 after 80 timesteps using a continuous Galerkin discretisation with streamline-upwind stabilisation.

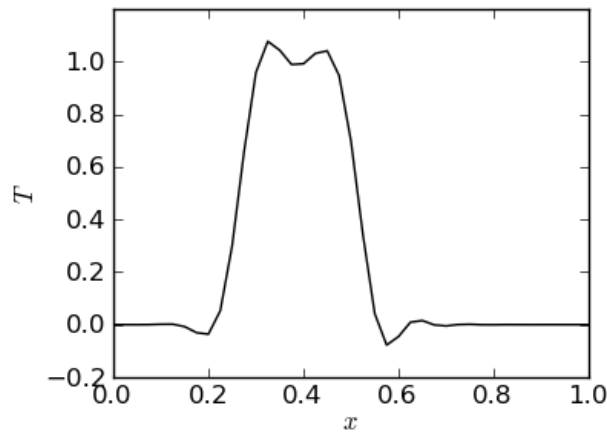


Figure 3.4: Pure advection of a 1D top hat function in a periodic domain at CFL number 1/8 after 80 timesteps using a continuous Galerkin discretisation with streamline-upwind Petrov-Galerkin stabilisation.

3.2.2 Boundary conditions

In the derivation of (3.5) we have assumed a homogeneous Neumann boundary condition on all boundaries. If we are considering all possible solutions c , the boundary term we have left out is

$$\int_{\partial\Omega} \varphi \mathbf{n} \cdot \bar{\bar{\kappa}} \cdot \nabla c. \quad (3.22)$$

A natural way of imposing an inhomogeneous Neumann boundary condition

$$\mathbf{n} \cdot \bar{\bar{\kappa}} \cdot \nabla c = g_N,$$

where g_N can be any prescribed function on the boundary $\partial\Omega$, is to impose it weakly. This is done in the same way as the weak form of the advection-diffusion equation was formed:

$$\int_{\partial\Omega} \varphi \mathbf{n} \cdot \bar{\bar{\kappa}} \cdot \nabla c = \int_{\partial\Omega} \varphi g_N, \quad \text{for all } \varphi. \quad (3.23)$$

Thus (3.23) can be used to replace the missing boundary term (3.22) with an integral of φg_N over the boundary.

In a similar way, a weakly imposed Dirichlet boundary condition can be related to an integration by parts of the advection term. Let us consider a pure advection problem ($\bar{\kappa} \equiv 0$). The weak form of this equation integrated by parts reads:

$$\int_{\Omega} \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \mathbf{u}) c + \int_{\partial\Omega} \varphi \mathbf{n} \cdot \mathbf{u} c = 0.$$

The final (boundary) term can again be substituted with a weakly imposed boundary condition $c = g_D$. In this case however, for physical and consequently numerical reasons, we only want to impose this on the inflow boundary, and the original term remains for the outflow boundary:

$$\int_{\Omega} \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \mathbf{u}) c + \int_{\partial\Omega_-} \varphi \mathbf{n} \cdot \mathbf{u} g_D + \int_{\partial\Omega_+} \varphi \mathbf{n} \cdot \mathbf{u} c = 0, \quad (3.24)$$

where $\partial\Omega_-$ and $\partial\Omega_+$ refer to respectively the inflow ($\mathbf{n} \cdot \mathbf{u} > 0$) and outflow boundaries ($\mathbf{n} \cdot \mathbf{u} < 0$).

It is to be noted that when we are applying boundary conditions weakly we still consider the full set of test functions, even those that don't satisfy the boundary condition. This means the discrete solution will not satisfy the boundary condition exactly. Instead the solution will converge to the correct boundary condition along with the solution in the interior as the mesh is refined.

An alternative way of implementing boundary conditions, so called *strongly imposed* boundary conditions, is to restrict the trial space to only those functions that satisfy the boundary condition. In the discrete trial space this means we no longer allow the coefficients c_i that are associated with the nodes x_i on the boundary, to vary but instead substitute the imposed Dirichlet boundary condition. As this decreases the dimension of the trial space, we also need to limit the size of the test space. This is simply done by removing the test function φ_i associated with the nodes x_i on the boundary, from the test space. Although this guarantees that the Dirichlet boundary condition will be satisfied exactly, it does not at all mean that the discrete solution converges to the exact continuous solution more quickly than it would with weakly imposed boundary conditions. Strongly imposed boundary conditions may sometimes be necessary if the boundary condition needs to be imposed strictly for physical reasons.

3.2.3 Discontinuous Galerkin discretisation

Integration by parts can be used to avoid taking derivatives of discontinuous functions. When using discontinuous test *and* trial functions (see Figure 3.5) however, neither the original advection equation, (3.5) with $\bar{\kappa} \equiv 0$, nor the version (3.24) integrated by parts are well-defined. Within an element e however the functions are continuous, and everything is well defined. So within a single element we may write

$$\int_e \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \mathbf{u}) c + \nabla \varphi \cdot \bar{\kappa} \cdot \nabla c + \int_{\partial e} \varphi \widehat{\mathbf{n} \cdot \mathbf{u} c} - \varphi \widehat{\mathbf{n} \cdot \bar{\kappa} \cdot \nabla c} = 0, \quad (3.25)$$

The hatted terms represent fluxes across the element facets, and therefore from one element to the other. Due to the discontinuous nature of the fields, there is no unique value for these flux terms, however the requirement that c be a conserved quantity does demand that adjacent elements make a consistent choice for the flux between them. The choice of flux schemes therefore forms a critical component of the discontinuous Galerkin method.

The application of boundary conditions occurs in the same manner as for the continuous Galerkin method. The complete system of equations is formed by summing over all the elements. Assuming

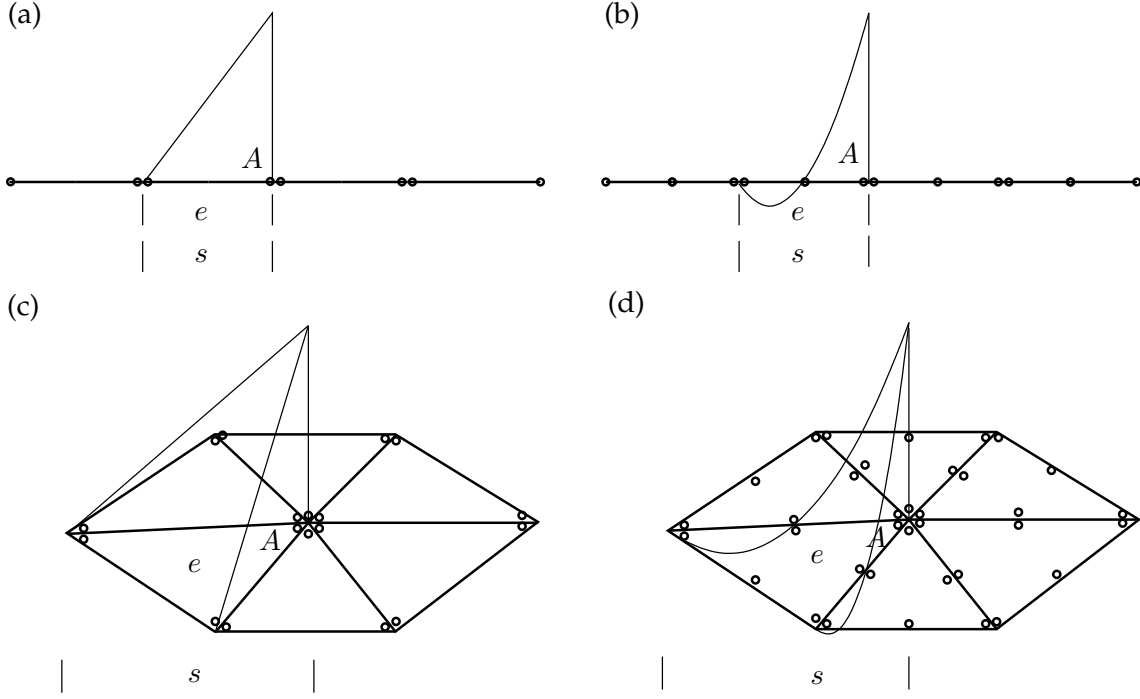


Figure 3.5: One-dimensional (a, b) and two-dimensional (c, d) schematics of piecewise linear (a, c) and piecewise quadratic (b, d) discontinuous shape functions. The shape function has value 1 at node A descending to 0 at all surrounding nodes. The number of nodes per element, e , depends on the polynomial order while the support, s , covers the same area as the element, e .

weakly applied boundary conditions, this results in:

$$\begin{aligned}
 \sum_e \left\{ \int_e \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \mathbf{u}) c + \nabla \varphi \cdot \bar{\bar{\mathbf{k}}} \cdot \nabla c \right. \\
 + \int_{\partial e \cap \partial \Omega_-^{D_c}} \varphi \mathbf{n} \cdot \mathbf{u} g_D + \int_{\partial e \cap \partial \Omega_+^{D_c}} \varphi \mathbf{n} \cdot \mathbf{u} c - \int_{\partial e \cap \partial \Omega^{D_c}} \varphi \mathbf{n} \cdot \bar{\bar{\mathbf{k}}} \cdot \nabla c \\
 \left. + \int_{\partial e \cap \partial \Omega^{N_c}} \varphi \mathbf{n} \cdot \mathbf{u} c - \varphi \mathbf{n} \cdot \bar{\bar{\mathbf{k}}} \cdot \nabla g_N + \int_{\partial e \setminus \partial \Omega} \varphi \widehat{\mathbf{n}} \cdot \mathbf{u} c - \varphi \widehat{\mathbf{n}} \cdot \bar{\bar{\mathbf{k}}} \cdot \nabla c \right\} = 0. \quad (3.26)
 \end{aligned}$$

3.2.3.1 Discontinuous Galerkin advection

Consider first the case in which $\bar{\bar{\mathbf{k}}} \equiv 0$. In this case, equation (3.25) reduces to:

$$\int_e \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \mathbf{u}) c + \int_{\partial e} \varphi \widehat{\mathbf{n}} \cdot \mathbf{u} c = 0, \quad (3.27)$$

and the question becomes, how do we represent the flux $\widehat{\mathbf{n}} \cdot \mathbf{u} c$?

Fluidity supports two different advective fluxes for DG. Upwind and local Lax-Friedrichs. For each flux scheme, there are two potentially discontinuous fields for which a unique value must be chosen. The first is the advecting velocity \mathbf{u} . The default behaviour is to average the velocity on each side of the face. The velocity is averaged at each quadrature point so decisions on schemes such as upwinding are made on a per-quadrature point basis. The second scheme is to apply a Galerkin projection to project the velocity onto a continuous basis. This amounts to solving the following equation:

$$\int_{\Omega} \hat{\psi} \cdot \hat{\mathbf{u}} = \int_{\Omega} \hat{\psi} \cdot \mathbf{u}, \quad (3.28)$$

where the hatted symbols indicate that the quantity in question is continuous between elements. In the following sections, $\hat{\mathbf{u}}$ will be used to indicate the flux velocity, which will have been calculated with one of these methods. Note that using the averaging method, $\hat{\mathbf{u}} = \mathbf{u}$ on the interior of each element with only the inter-element flux differing from \mathbf{u} while for the projection method, $\hat{\mathbf{u}}$ and \mathbf{u} may differ everywhere.

Upwind Flux In this case, the value of c at each quadrature point on the face is taken to be the upwind value. For this purpose the upwind value is as follows: if the flow is out of the element then it is the value on the interior side of the face while if the flow is into the element, it is the value on the exterior side of the face. If we denote the value of c on the upwind side of the face by c_{upw} then equation (3.27) becomes:

$$\int_e \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \hat{\mathbf{u}}) c + \int_{\partial e} \varphi \mathbf{n} \cdot \hat{\mathbf{u}} c_{\text{upw}} = 0, \quad (3.29)$$

Summing over all elements, including boundary conditions and writing c_{int} to indicate flux terms which use the value of c from the current element only, we have:

$$\begin{aligned} \sum_e \left\{ \int_e \varphi \frac{\partial c}{\partial t} - (\nabla \cdot \varphi \hat{\mathbf{u}}) c + \int_{\partial e \cap \partial \Omega_{-}^{Dc}} \mathbf{n} \cdot \hat{\mathbf{u}} g_D \right. \\ \left. + \int_{\partial e \cap \partial \Omega^{Nc} \cap \partial \Omega_{+}^{Dc}} \mathbf{n} \cdot \hat{\mathbf{u}} c_{\text{int}} \right. \\ \left. + \int_{\partial e \setminus (\partial \Omega_D \cap \partial \Omega_N)} \mathbf{n} \cdot \hat{\mathbf{u}} c_{\text{upw}} \right\} = 0. \end{aligned} \quad (3.30)$$

Second integration by parts In Fluidity the advection term with upwinded flux may be subsequently integrated by parts again within each element. As this is just a local operation on the continuous pieces of c within each element, the new boundary integrals take the value of c on the inside of the element, c_{int} . On the outflow boundary of each element this means the c_{int} cancels against c_{upw} (when the summation over all elements occurs). Writing ∂e_{-} for the inflow part of the element boundary, we therefore obtain:

$$\begin{aligned} \sum_e \left\{ \int_e \varphi \frac{\partial c}{\partial t} - \varphi \hat{\mathbf{u}} \cdot \nabla c + \int_{\partial e_{-} \cap \partial \Omega_{-}^{Dc}} \mathbf{n} \cdot \hat{\mathbf{u}} (g_D - c_{\text{int}}) \right. \\ \left. + \int_{\partial e_{-} \setminus (\partial \Omega_D \cap \partial \Omega_N)} \mathbf{n} \cdot \hat{\mathbf{u}} (c_{\text{upw}} - c_{\text{int}}) \right\} = 0. \end{aligned} \quad (3.31)$$

The difference $c_{\text{int}} - c_{\text{upw}}$ on the inflow boundary remains, and is often referred to as a *jump condition*. Note also that the boundary terms on the Neumann domain boundary and the outflow part of the Dirichlet boundary (really also a Neumann boundary) have disappeared. Note that (3.30) and (3.31) are completely equivalent. The advantage of the second form, referred to in Fluidity as “integrated-by-parts-twice”, is that the numerical evaluation of the integrals (quadrature), may be chosen not to be exact (incomplete quadrature). For this reason the second form may be more accurate as the internal outflow boundary integrals are cancelled exactly.

Local Lax-Friedrichs flux The local Lax-Friedrichs flux formulation is defined in [Cockburn and Shu \[2001, p208\]](#). For the particular case of tracer advection, this is given by:

$$\widehat{\mathbf{n} \cdot \mathbf{u} c} = \frac{1}{2} \mathbf{n} \cdot \hat{\mathbf{u}} (c_{\text{int}} + c_{\text{ext}}) - \frac{C}{2} c_{\text{int}} - c_{\text{ext}}, \quad (3.32)$$

where c_{ext} is the value of c on the exterior side of the element boundary and in which for each facet $s \subset \partial e$:

$$C = \sup_{x \in s} |\hat{\mathbf{u}} \cdot \mathbf{n}|. \quad (3.33)$$

3.2.3.2 Advective stabilisation for DG

As described by Cockburn and Shu [2001], the DG method with p -th order polynomials using an appropriate Riemann flux (the upwind flux in the case of the scalar advection equation) applied to hyperbolic systems is always stable and $(p + 1)$ -th order accurate. However, Godunov's theorem states that linear monotone¹ schemes are at most first-order accurate. Hence, for $p > 0$, we expect the DG method to generate new extrema, which are observed as undershoots and overshoots for the scalar advection equation. However, the DG method does have the additional property that if the DG solution fields² are bounded element-wise, *i.e.* at each element face a solution field lies between the average value for that element and the average value for the neighbouring element on the other side of the face, then the element-averaged DG field (*i.e.* the projection of the DG field to P_0) does not obtain any new minima. This result only holds if the explicit Euler timestepping method (or one of the higher-order extensions, namely the Strongly Structure Preserving Runge-Kutta (SSPRK) methods) is used. Hence, the DG field can be made monotonic by adjusting the solution at the end of each timestep (or after each SSPRK stage) so that it becomes bounded element-wise. This is done in a conservative manner *i.e.* without changing the element-averaged values. For P_1 , only the slopes can be adjusted to make the solution bounded element-wise, and hence the adjustment schemes are called *slope limiters*.

Types of slope limiter There are two stages in any slope limiter. First all the elements which do not currently satisfy the element bounded condition must be identified. Secondly, the slopes (and possibly higher-order components of the solution) in each of these elements must be adjusted so that they satisfy the bounded condition. In general, this type of adjustment has the effect of introducing extra diffusion, and so it is important to (a) identify as few elements as possible, and (b) adjust the slopes as little as possible, in order to introduce as little extra diffusion as possible. For high-order elements, exactly how to do this is a very contentious issue but a few approaches are satisfactory for low-order elements.

Vertex-based limiter This limiter, introduced in Kuzmin [2010], works on a hierarchical Taylor expansion. It is only currently implemented for linear elements. In this case, the DG field c in one element e may be written as

$$c = \bar{c} + c', \quad \bar{c} = \frac{\int_e c dV}{Vol(e)},$$

and the limiter replaces c with c_L given by

$$c = \bar{c} + \alpha c',$$

finding the maximum value $\alpha > 0$ such that at each vertex, c is bounded by the maximum and minimum of the values of T at that vertex over all elements that share the vertex. This limiter has no parameters, and is the currently recommended limiter.

Cockburn-Shu limiter This limiter, introduced in Cockburn and Shu [2001], only checks the element bounded condition at face centres. There is a tolerance parameter, the TVB factor, which controls how sensitive the method is to the bounds (the value recommended in the paper is 5) and a limit factor, which scales the reconstructed slope (the value recommended in the paper is 1.1). The method seems not to be independent of scaling, and the paper assumes an $\mathcal{O}(1)$ field, so these factors need tuning for other scalings.

¹A monotone scheme is a scheme that does not generate new extrema.

²For a system of equations this refers to the characteristic variables obtained from the diagonalisation of the hyperbolic system.

Hermite-WENO limiter This limiter makes use of the Weighted Essentially Non-Oscillatory (WENO) interpolation methods, originally used to obtain high-order non-oscillatory fluxes for finite volume methods, to reconstruct the solution in elements which do not satisfy the element-wise bounded condition (sometimes referred to as “troubled elements”). The principle is the following: if we try to reconstruct the solution as a p -th order polynomial in an element by fitting to the cell average of the element and of some neighbouring elements, then if there is a discontinuity in the solution within the elements used then the p -th order polynomial is very wiggly and will exceed its bounds. The WENO approach is as follows:

1. Construct a number of polynomials approximations using various different combinations of elements, each having the same cell-average in the troubled element.
2. Calculate a measure of the wiggly-ness of each polynomial (called an oscillation indicator).
3. The reconstructed solution is a weighted average of each of the polynomials, with the weights decreasing with increasing oscillation indicator.

Thus if there is a discontinuity to one side of the element, the reconstructed solution will mostly use information from the other side of the discontinuity. The power law relating the weights with the oscillation indicators can be selected by the user, but is configured so that in the case of very smooth polynomials, the reconstruction accuracy exceeds the order of the polynomials *e.g.* 5th order for 3rd order polynomials.

In practise, making high order reconstructions from unstructured meshes is complicated since many neighbouring elements must be used. If one also uses the gradients from the element and the direct neighbours (Hermite interpolation) this is sufficient to obtain an essentially non-oscillatory scheme. This is called the Hermite-WENO method. In this method applied to P_1 , the complete set of approximations used for the solution in an element are:

- The original solution in the element E .
- Solutions with gradient constructed from the mean value of the element E and d other elements which share a face with E . In 2D this is 3 solutions, and in 3D this is 4 solutions. Each of these solutions has the same mean value as the original solution.
- Solutions with gradient the same as one of the d neighbouring elements. Each of these solutions has the same mean value as the original solution.

This is a total of $2d+1$ solutions which must be weighted according to their oscillator indicator value.

The advantage of using WENO (and H-WENO) reconstruction is that it preserves the order of the approximation, and hence it is not quite so important to avoid it being used in smooth regions (other limiters would introduce too much diffusion in those regions). However, reconstruction is numerically intensive and so to make H-WENO more computationally feasible, it must be combined with a discontinuity detector which identifies troubled cells. It does not do too much damage to the solution if the discontinuity detector is too strict *i.e.* identifies too many elements as troubled, but will reduce the efficiency of the method.

3.2.3.3 Diffusion term for DG

In this section we describe the discretisation of the diffusion operator using discontinuous Galerkin methods. We concentrate on solving the Poisson equation

$$\nabla^2 c = f, \tag{3.34}$$

although this can easily be extended to the advection-diffusion and momentum equations by replacing f with the rest of the equation. Discretising the Poisson equation (3.34) using discontinuous Galerkin is a challenge since discontinuous fields are not immediately amenable to introducing second-order operators (they are best at advection): the treatment of the diffusion operator is one of the main drawbacks with discontinuous Galerkin methods. The standard continuous finite element approach is to multiply equation (3.34) by a test function and integrate the Laplacian by parts, leading to integrals containing the gradient of the trial and test functions. In DG methods, since the trial and test functions contain discontinuities, these integrals are not defined. There are two approaches to circumventing this problem which have been shown to be essentially equivalent in Arnold et al. [2002], which we describe in this section.

The first approach, which leads to the class of interior penalty methods, is to integrate the Laplacian by parts separately in each element (within which the functions are continuous), and the equations become

$$\sum_e \left(- \int_e \nabla \varphi^\delta \cdot \nabla c^\delta + \int_{\partial e} \varphi^\delta \mathbf{n} \cdot \nabla c^\delta \right) = \sum_e \int_e \varphi^\delta f^\delta, \quad (3.35)$$

where e is the element index, \int_e indicates an integral over element e , $\int_{\partial e}$ indicates an integral over the boundary of e , φ^δ is the DG test function, c^δ is the DG trial function, and f^δ is the DG approximation of f . The next step is to notice that for each facet (face in 3D, edge in 2D or vertex in 1D) there is a surface integral on each side of the facet, and so equation (3.35) becomes

$$- \sum_e \int_e \nabla \varphi^\delta \cdot \nabla c^\delta + \sum_\Gamma \int_\Gamma [[\nabla c^\delta \varphi^\delta]] = \sum_e \int_e \varphi^\delta f^\delta, \quad (3.36)$$

where Γ is the facet index, \int_Γ indicates an integral over facet Γ , and the jump bracket $[[v^\delta]]$ measures the jump in the normal component of v^δ across Γ and is defined by

$$[[v^\delta]] = v^\delta|_{e^+} \cdot \mathbf{n}^+ + v^\delta|_{e^-} \cdot \mathbf{n}^-,$$

where e^+ and e^- are the elements on either side of facet Γ , $v_{e^\pm}^\delta$ is the value of the vector-valued DG field v^δ on the e^\pm side of Γ , and \mathbf{n}^\pm is the normal to Γ pointing out of E^\pm . The problem with this formulation is that there is still no communication between elements, and so the equation is not invertible. The approximation is made consistent by making three changes to equation (3.36). Firstly, in the facet integral, the test function φ^δ (which takes different values either side of the face) is replaced by the average value, $\{\varphi^\delta\}$ defined by

$$\{\varphi^\delta\} = \frac{\varphi^\delta|_{E^+} + \varphi^\delta|_{E^-}}{2},$$

leading to

$$- \sum_e \int_e \nabla \varphi^\delta \cdot \nabla c^\delta + \sum_\Gamma \int_\Gamma [[\nabla c^\delta]] \{\varphi^\delta\} = \sum_e \int_e \varphi^\delta f^\delta. \quad (3.37)$$

Secondly, to make the operator symmetric (required for adjoint consistency, also means that the conjugate gradient method can be used to invert the matrix), an extra jump term is added, leading to

$$- \sum_e \int_e \nabla \varphi^\delta \cdot \nabla c^\delta + \sum_\Gamma \int_\Gamma [[\nabla c^\delta]] \{\varphi^\delta\} + \{c^\delta\} [[\nabla \varphi^\delta]] = \sum_e \int_e \varphi^\delta f^\delta. \quad (3.38)$$

Note that this symmetric averaging couples together each node in element e with all the nodes in the elements which share facets with element e . Thirdly, a penalty term is added which tries to reduce discontinuities in the solution, and the discretised equation becomes

$$- \sum_e \int_e \nabla \varphi^\delta \cdot \nabla c^\delta + \sum_\Gamma \int_\Gamma [[\nabla c^\delta]] \{\varphi^\delta\} + \{c^\delta\} [[\nabla \varphi^\delta]] + \alpha(\varphi^\delta, c^\delta) = \sum_e \int_e \varphi^\delta f^\delta, \quad (3.39)$$

where $\alpha(\cdot, \cdot)$ is the penalty functional which satisfies the following properties:

1. Symmetry: $\alpha(c^\delta, \varphi^\delta) = \alpha(\varphi^\delta, c^\delta)$.
2. Positive semi-definiteness: $\alpha(c^\delta, c^\delta) \geq 0$.
3. Continuity-vanishing: $\alpha(c^\delta, c^\delta) = 0$ when c^δ is continuous across all facets.
4. Discontinuity-detecting: $\alpha(c^\delta, c^\delta)$ increases as the discontinuities across facets increase.

The form of equation (3.39) is called the *primal form*. Note that, due to the continuity-vanishing property, equation (3.39) is satisfied by the exact solution (which is always continuous) to the Poisson equation, which is the required *consistency condition*. In defining the particular form of the penalty functional α it is necessary to maintain a balance: if the functional penalises discontinuities too much then the resulting matrix is ill-conditioned, if it penalises discontinuities too little then there is not enough communication between elements and the numerical solution does not converge to the exact solution. The particular form of α for the Interior Penalty method is described below.

The second approach (the Local Discontinuous Galerkin (LDG) framework [Cockburn and Shu, 1998, Sherwin et al., 2006] which leads to Bassi-Rebay and Compact Discontinuous Galerkin methods) is to introduce a vector field $\boldsymbol{\xi}$, to rewrite the Poisson equation as a system of first-order equations

$$\nabla \cdot \boldsymbol{\xi} = f, \quad \boldsymbol{\xi} = \nabla c, \quad (3.40)$$

and to finally eliminate the vector field $\boldsymbol{\xi}$. This elimination is possible to do locally (*i.e.* only depending on the values of c in nearby elements) since the mass matrix for DG fields is block diagonal and so the inverse mass matrix does not couple together nodes from different elements. For discontinuous Galerkin methods, we introduce a discontinuous vector test function \boldsymbol{w}^δ . Multiplication of equations (3.40) by test functions, integrating over a single element E and applying integration by parts leads to

$$-\int_e \nabla \varphi^\delta \cdot \boldsymbol{\xi}^\delta + \int_{\partial e} \varphi^\delta \boldsymbol{n} \cdot \hat{\boldsymbol{\xi}}^\delta = \int_e \varphi^\delta f^\delta, \quad -\int_e \nabla \cdot \boldsymbol{w}^\delta c^\delta + \int_{\partial e} \boldsymbol{n} \cdot \boldsymbol{w}^\delta \hat{c}^\delta = \int_e \boldsymbol{w}^\delta \cdot \boldsymbol{\xi}^\delta. \quad (3.41)$$

This form of the equations is called the *dual form*. The exact definition of the particular scheme depends on how the surface values (fluxes) $\hat{\boldsymbol{\xi}}^\delta$ and \hat{c}^δ are defined. The choice of these fluxes has an impact on the stability (whether there are any spurious modes), consistency (whether the discrete equation is satisfied by the exact solution), convergence (whether and how fast the numerical solution approaches the exact solution), and sparsity (how many non-zero elements there are in the resulting matrix). It is worth noting at this point that the method of rewriting the second-order operator as a first-order system has some superficial connections with the discrete pressure projection method for continuous finite element methods as described in Section 3.6.1. However, many of the ideas do not carry over to the discontinuous Galerkin framework, for example, it is neither necessary nor sufficient to reduce the polynomial order of φ^δ relative to the polynomial order of $\boldsymbol{\xi}^\delta$. The issues of stability, consistency, convergence and sparsity for DG discretisations of the diffusion operator are extremely subtle and there is an enormous literature on this topic; it remains a dangerous tar pit for the unwary developer looking to invent a new DG diffusion operator discretisation.

It was shown in Arnold et al. [2002] (which is an excellent review of DG methods for elliptic problems) that numerical schemes obtained from this second approach can be transformed to primal form, resulting precisely in discretisations of the form (3.39) with some particular choice of the functional α . Hence, it is possible to describe the three options available in Fluidity together, in the following subsections.

Interior Penalty The Interior Penalty method is a very simple scheme with penalty functional of the form

$$\alpha(\varphi^\delta, c^\delta) = \sum_{\Gamma} C_{\Gamma} \int_{\Gamma} [[\varphi^\delta]] \cdot [[c^\delta]], \quad (3.42)$$

where for a scalar function φ^δ , the jump bracket $[[\cdot]]$ is a vector quantity defined as

$$[[\varphi^\delta]] = \varphi^\delta|_{E^+} \mathbf{n}^+ + \varphi^\delta|_{E^-} \mathbf{n}^-.$$

For convergence, the constant C_Γ should be positive and proportional to h^{-1} , where h is the element edge length. This needs to be carefully defined for anisotropic meshes.

Bassi-Rebay The scheme of Bassi-Rebay [Bassi and Rebay, 1997] is in some sense the most simple choice within the LDG framework, in which the fluxes are just taken from the symmetric averages:

$$\hat{\xi}^\delta = \{\xi^\delta\}, \quad \hat{\varphi}^\delta = \{\varphi^\delta\}.$$

This scheme was analysed in Arnold et al. [2002], and was shown to only converge in the following rather weak sense: if the numerical solution and exact solution are projected to piecewise constant (P_0) functions then these projected solutions converge to each other at order $p + 1$, without this projection they only converge at order p , where p is the polynomial order used in the DG element. Furthermore, the Bassi-Rebay scheme has a very large stencil (large number of non-zero values in the resulting matrix). For this reason, other more sophisticated flux choices have been investigated.

Compact Discontinuous Galerkin The Compact Discontinuous Galerkin (CDG) scheme Peraire and Persson [2008] has a rather complex choice of fluxes based on “lifting operators” (see the paper for more information). When transforming the equations to primal form, this choice of flux results in a sophisticated penalty function with two terms. The first term exactly cancels part of the flux integrals in equation (3.39) so that all of the symmetric fluxes using the averaging bracket $\{\cdot\}$ are replaced by the flux evaluated on one particular (arbitrarily chosen) side of the facet (there are various schemes for making this choice). The second term only couples together nodes which share the same facet. Both of these terms result in a much smaller stencil than for the Bassi-Rebay scheme in particular and other LDG schemes in general. Furthermore, the scheme is observed to be stable, consistent, and optimally convergent (numerical solutions converge at order $(p + 1)$). The scheme optionally includes the interior penalty term from equation (3.42), but the constant may be independent of h . This term only appears to be necessary for the mathematical proofs of stability and convergence since in practise good results are obtained without this term (in fact the results are usually more accurate without this term). Since the penalty term has only one tunable constant (which may be set to zero) which does not depend on h , this makes the CDG scheme very attractive for anisotropic elements, including large aspect ratio meshes such as those used in large scale ocean modelling.

3.2.4 Control volume discretisation

Finite volume discretisations may be thought of as the lowest order discontinuous Galerkin method, using piecewise constant shape functions (see Figure 3.6). In Fluidity this type of element centred discretisation is handled through the discontinuous Galerkin method, however the model also supports an alternative finite volume discretisation referred to as a control volume (CV) discretisation.

The control volume discretisation uses a dual mesh constructed around the nodes of the parent finite element mesh. In two dimensions this is constructed by connecting the element centroids to the edge midpoints while in three dimensions the face centroids are also introduced. For cube meshes (quadrilaterals in 2D and hexahedra in 3D) this produces a staggered mesh of the same type. For simplex meshes this process produces complex polyhedra (see Figures 3.7 and 3.8).

Once the dual control volume mesh has been defined, it is possible to discretise the advection-diffusion equation (3.5) using piecewise constant shape functions within each volume, v (see Figure 3.9). However, as with the discontinuous Galerkin method the equation is only well defined when

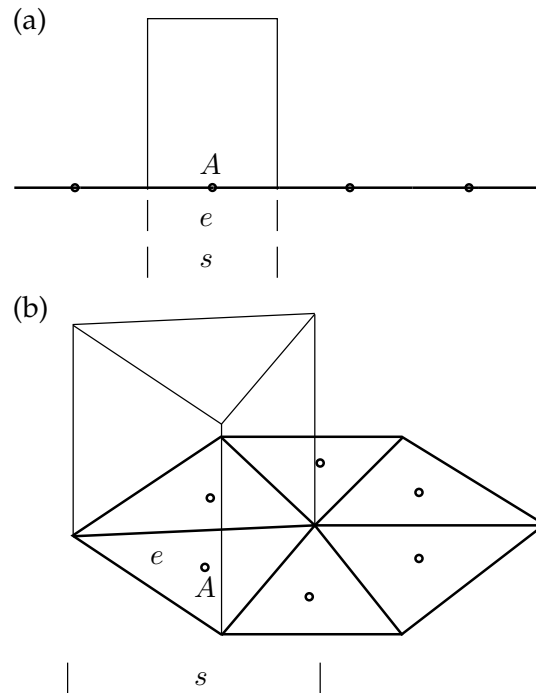


Figure 3.6: One-dimensional (a) and two-dimensional (b) schematics of piecewise constant, element centred shape functions. The shape function has value 1 at node A and across the element, e , descending to 0 at the element boundaries. As with other discontinuous shape functions, the support, s , coincides with the element, e .

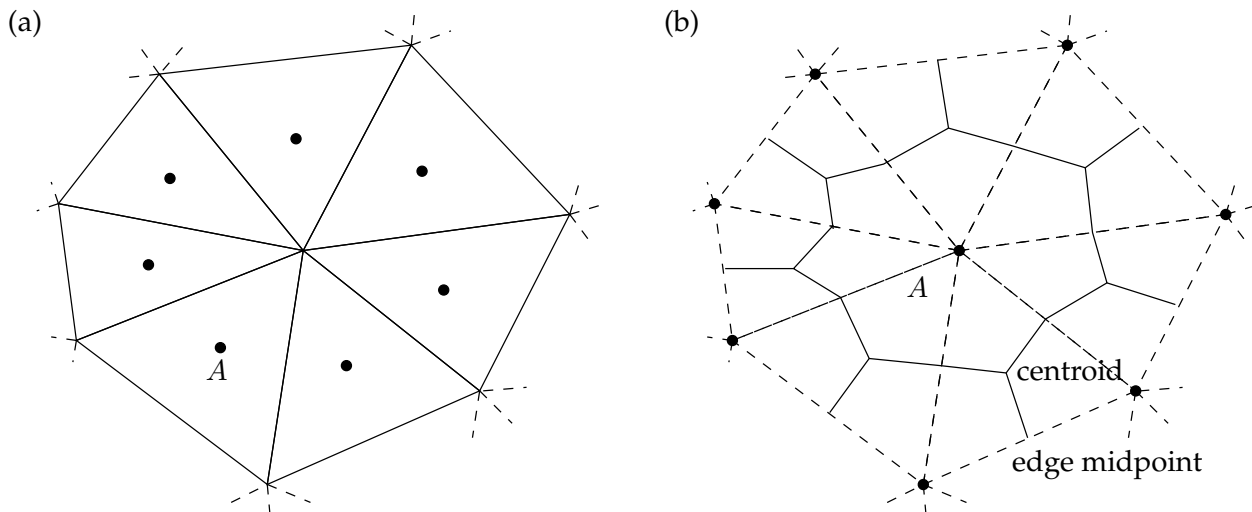


Figure 3.7: Comparison between (a) a two-dimensional finite volume simplex mesh and (b) the equivalent control volume dual mesh (solid lines) constructed around a piecewise linear continuous finite element parent mesh (dashed lines). In the finite volume mesh the nodes (e.g. A) are element centred whereas in the control volume dual mesh the nodes are vertex based. In 2D the control volumes are constructed around A by connecting the centroids of the neighbouring triangles to the edge midpoints. See Figure 3.8 for the equivalent three-dimensional construction.

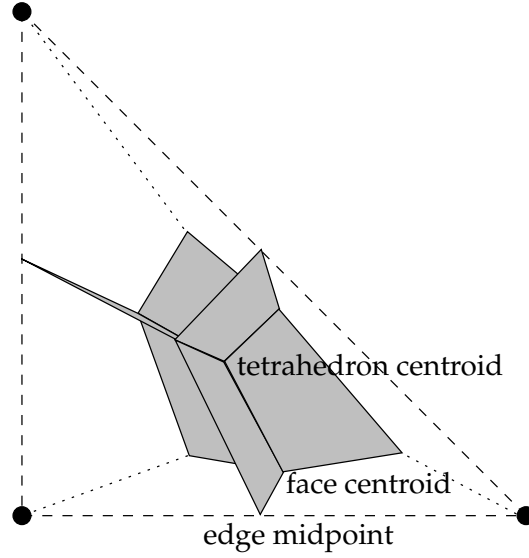


Figure 3.8: The six dual control volume mesh faces within a piecewise linear tetrahedral parent mesh element. Each face is constructed by connecting the element centroid, the face centroids and the edge midpoint.

integrated by parts within such a volume, v , allowing us to write:

$$\int_v \frac{\partial c}{\partial t} + \int_{\partial v} \widehat{\mathbf{n} \cdot \mathbf{u} c} - \widehat{\mathbf{n} \cdot \bar{\kappa} \cdot \nabla c} = 0, \quad (3.43)$$

Note that the test function present in the previous discretisation sections, φ , can be dropped from the equation as it is 1 everywhere within the volume v . Furthermore, terms involving the gradient of either φ or c can be dropped as both are constant functions. The boundary integral for the diffusivity $\bar{\kappa}$ is a special case that will be dealt with in a later section.

As with the discontinuous Galerkin discretisation, the hatted terms represent fluxes across the volume facets: and therefore from one volume to the other. Due to the discontinuous nature of the fields, there is no unique value for these flux terms, however the requirement that c be a conserved quantity does demand that adjacent volumes make a consistent choice for the flux between them. The choice of flux schemes therefore forms a critical component of the control volume method.

The application of boundary conditions occurs in the same manner as for the discontinuous Galerkin method. The complete system of equations is formed by summing over all the volumes. Assuming weakly applied boundary conditions, this results in:

$$\sum_v \left\{ \int_v \frac{\partial c}{\partial t} + \int_{\partial v \cap \partial \Omega_-^{Dc}} \mathbf{n} \cdot \mathbf{u} g_D + \int_{\partial v \cap \partial \Omega_+^{Dc}} \mathbf{n} \cdot \mathbf{u} c - \int_{\partial v \cap \partial \Omega^{Dc}} \mathbf{n} \cdot \bar{\kappa} \cdot \nabla c + \int_{\partial v \cap \partial \Omega^{Nc}} \mathbf{n} \cdot \mathbf{u} c - \mathbf{n} \cdot \bar{\kappa} \cdot \nabla g_N + \int_{\partial v \setminus \partial \Omega} \widehat{\mathbf{n} \cdot \mathbf{u} c} - \widehat{\mathbf{n} \cdot \bar{\kappa} \cdot \nabla c} \right\} = 0. \quad (3.44)$$

3.2.4.1 Control Volume advection

Consider first the case in which $\bar{\kappa} \equiv 0$. In this case, equation (3.43) reduces to:

$$\int_v \frac{\partial c}{\partial t} + \int_{\partial v} \widehat{\mathbf{n} \cdot \mathbf{u} c} = 0, \quad (3.45)$$

and the question becomes, how do we represent the flux $\widehat{\mathbf{n} \cdot \mathbf{u} c}$?

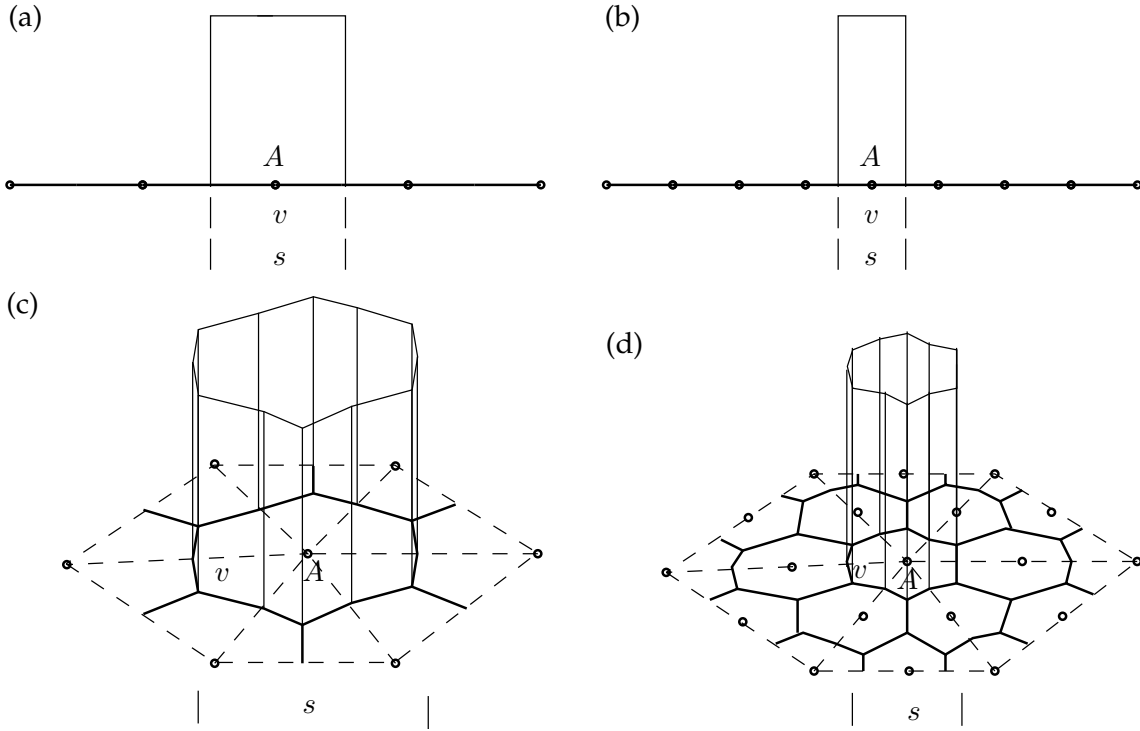


Figure 3.9: One-dimensional (a, b) and two-dimensional (c, d) schematics of piecewise constant control volume shape functions and dual meshes based on the parent (dashed lines) linear (a, c) and quadratic (b, d) finite element meshes. The shape function has value 1 at node A descending to 0 at the control volume boundaries. The support, s , coincides with the volume, v .

Fluidity supports multiple different advective fluxes for CV. Unlike with DG the advection velocity \mathbf{u} is always well-defined at the control volume facets owing to the fact that they cross through the centre of the elements of the parent mesh where the velocity is continuous. Therefore it is only necessary to describe how the face value of c is defined.

In the following paragraphs we will refer to the donor or central value, c_{c_k} , and the downwind value, c_{d_k} . These are associated with facet k of the control volume dual mesh, and are defined depending on the direction of the flux across that facet. If the flow across facet k is exiting volume v , i.e. $\mathbf{n} \cdot \mathbf{u}|_k > 0$, then the donor value, c_{c_k} is the value in the volume, c_v and the downwind value, c_{d_k} is the value in the volume that the flux is entering. Similarly if the flow is entering the volume v , i.e. $\mathbf{n} \cdot \mathbf{u}|_k < 0$, then the donor value, c_{c_k} , is the value from the neighbouring control volume while the downwind value, c_{d_k} , is the value in the volume, c_v . By default only first order quadrature is performed on the control volume facets, however if higher order control volume facet quadrature is selected then k refers to each quadrature point on the facet.

First Order Upwinding In this case, the value of c at each quadrature point on each facet is taken to be the donor value, c_{c_k} . Then (3.45) becomes:

$$\int_v \frac{\partial c}{\partial t} + \sum_k \int_{\partial v_k} \mathbf{n} \cdot \mathbf{u} c_{c_k} = 0. \quad (3.46)$$

First order upwinding is stable in the sense of boundedness, assuming an appropriate temporal discretisation is selected, it is however very diffusive so normally a higher order but less stable face value scheme is selected.

Trapezoidal In this case, the average of the donor and downwind values, $\frac{c_{c_k} + c_{d_k}}{2}$, is taken as the value of c at each facet. This is generally unstable regardless of the temporal discretisation chosen so requires limiting (see below).

Finite Element Interpolation In this case, the value of c at each quadrature point of the facet is interpolated using the finite element basis functions on the parent mesh. This is possible as the nodes of both the dual and parent meshes are co-located. Like the trapezoidal face value this method is also generally unstable so normally requires limiting (see below).

First Order Downwinding In this case, the value of c at each quadrature point of the facet is set to the downwind value, c_{d_k} . First order downwinding is unconditionally unstable and is intended for demonstration purposes only.

Face Value Limiting As noted above, several of the face value schemes above result in unstable, unbounded advective fluxes. To compensate for this it is possible to limit the face value in an attempt to maintain boundedness. This requires an estimate of the upwind flux entering the control volume so we introduce a new value, c_{u_k} , for the value upwind of the donor volume, c_{c_k} . On a fully unstructured mesh this value is not directly available so instead it must be estimated. For cube meshes the default behaviour is to estimate the upwind value from the minimum or maximum of the surrounding nodes depending on the gradient between the donor and downwind values. On simplex meshes Fluidity defaults to more accurate schemes that project the value at the upwind position based on all the nodes in the parent element immediately upwind of the donor node (see Figure 3.10(a)). As this value is not necessarily bounded itself it is also possible to bound it so that it falls within the values of the upwind element. Additionally, on a boundary it is possible to reflect the upwind value back into the domain (see Figure 3.10(b)), which may be more appropriate.

Once an upwind value, c_{u_k} , is available it is possible to estimate whether the chosen face value, c_{f_k} , will cause the solution to become unbounded. Fluidity uses a normalised variable diagram [NVD, Waterson and Deconinck, 2007, Wilson, 2009] based scheme to attempt to enforce a total variation diminishing [TVD, LeVeque, 2002] definition of boundedness. First, a normalised donor value:

$$\bar{c}_{c_k} = \frac{c_{c_k} - c_{u_k}}{c_{d_k} - c_{u_k}}, \quad (3.47)$$

and a normalised face value:

$$\bar{c}_{f_k} = \frac{c_{f_k} - c_{u_k}}{c_{d_k} - c_{u_k}}, \quad (3.48)$$

are defined. These are then used to define the axes of the NVD. This has the advantage of dealing with multiple configurations in a single coordinate system [NVD, Waterson and Deconinck, 2007, Wilson, 2009].

Many face value limiting schemes can be implemented on the NVD [Leonard, 1991]. Fluidity provides two options: the Sweby [Sweby, 1984] and ULTIMATE [Leonard, 1991] limiters (see Figure 3.11).

Sweby Limiter The Sweby limiter [Sweby, 1984] defines a region on the NVD (shaded grey in Figure 3.11(a)) that is considered bounded. Any combination of normalised face and donor values falling within this region is left unchanged. Values falling outside this area are 'limited' along lines of constant normalised donor value back onto the top or bottom of the stable region, or if they fall outside the range $0 < \bar{c}_{c_k} < 1$, back to first order upwinding (represented by the diagonal line on a NVD).

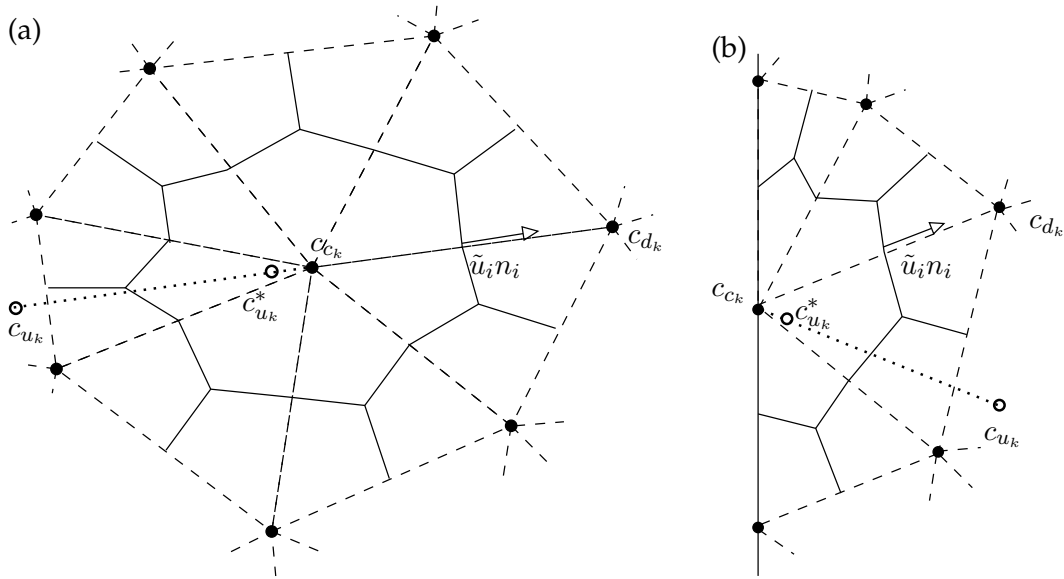


Figure 3.10: Calculation of the upwind value, c_{u_k} , on an unstructured simplex mesh (a) internally and (b) on a boundary. The control volume mesh is shown (solid lines) around the nodes (black circles) which are co-located with the solution nodes of the parent piecewise linear continuous finite element mesh (dashed lines). An initial estimate of the upwind value, $c_{u_k}^*$, is found by interpolation within the upwind parent element. The point-wise gradient between this estimate and the donor node, c_{c_k} , is then extrapolated the same distance between the donor and downwind, c_{d_k} , to the upwind value, c_{u_k} .

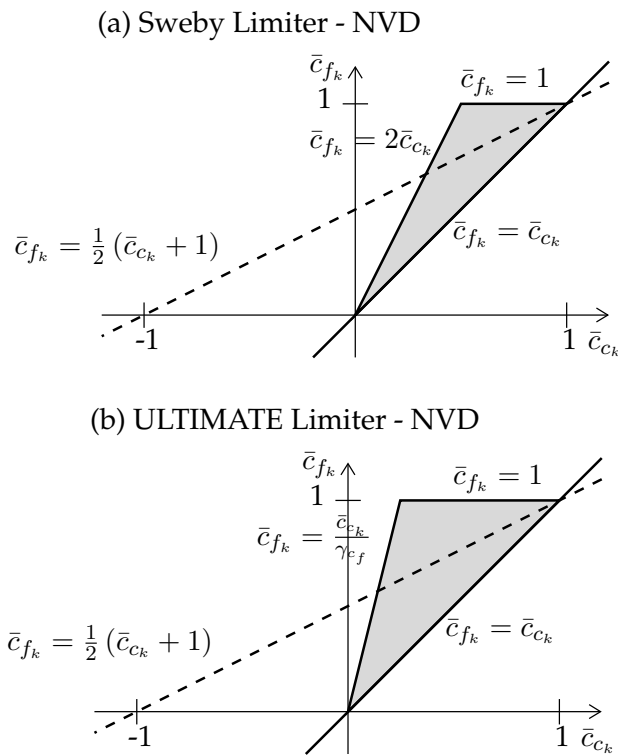


Figure 3.11: The Sweby (a) and ULTIMATE (b) limiters (shaded regions) represented on a normalised variable diagram (NVD). For comparison the trapezoidal face value scheme is plotted as a dashed line in both diagrams. γ is the Courant number at the control volume face.

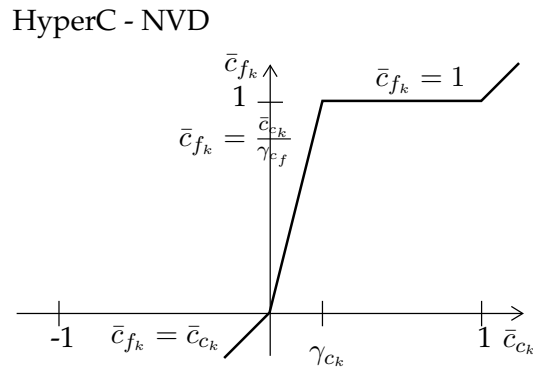


Figure 3.12: The HyperC face value scheme represented on a normalised variable diagram (NVD).

As an example the trapezoidal face value scheme is plotted as a dashed line in Figure 3.11(a). It only crosses the Sweby limiter region for a small range of normalised donor values and it is only across this range that the face value will not be limited. Outside of this range the trapezoidal face value scheme is considered unstable and must be limited. This is akin to saying that the trapezoidal face value scheme may only be used in regions where the solution is sufficiently smooth. In particular, outside the range $0 < \bar{c}_{c_k} < 1$, the solution is already locally unbounded (in a TVD sense) and hence only first order upwinding may be used to restore local boundedness.

ULTIMATE Limiter An alternative definition of the sufficiently smooth region of the NVD is provided by the ULTIMATE limiter [see Figure 3.11(b), Leonard, 1991]. This works using the same principals as the Sweby limiter however the region defined as being stable now incorporates an upper bound that depends on the Courant number at the control volume face to the donor value, γ_{c_f} . Hence the region expands and contracts depending on the Courant number, collapsing back entirely to first order upwinding once a Courant number of one is attained. It is therefore principally designed for explicit advection. In fact for explicit advection, with the correct control volume based definition of the Courant number, this limiter exactly defines the total variation bounded region, guaranteeing boundedness in that sense [Leonard, 1991, Després and Lagoutière, 2001].

HyperC HyperC uses the same principals as the ULTIMATE limiter but instead of limiting other face value schemes using it, HyperC uses the NVD as a face value scheme directly [Leonard, 1991]. Given a donor, downwind and upwind value (and hence a normalised donor value) it simply uses the upper boundary of the TVD zone to calculate the normalised face value (and hence a face value, see Figure 3.12). For explicit advection, with the correct definition of the control volume based Courant number, γ_{c_f} , this scheme aims to produce a minimally diffusive advection scheme. It is however, only intended for the advection of step-like functions as it results in distortion and staircasing of smoother functions. Wilson [2009] discusses the implementation of HyperC in Fluidity and its extension to multiple dimensions.

UltraC Like HyperC, UltraC uses the NVD to define a face value directly. The difference is that it uses a total variation bounded (TVB) rather than a total variation diminishing (TVD) definition of boundedness. As with HyperC this aims to minimise numerical diffusion while advecting step-like functions but will distort smoother fields.

UltraC can be represented on an NVD (see Figure 3.13(a)), where it is obvious that the bounded range has been extended beyond TVD range used by HyperC, $0 < \bar{c}_{c_k} < 1$ (shown as a dotted line in Figure 3.13(a)). Instead it is now dependent on two new values: a target upwind value, c_{tur} and a target

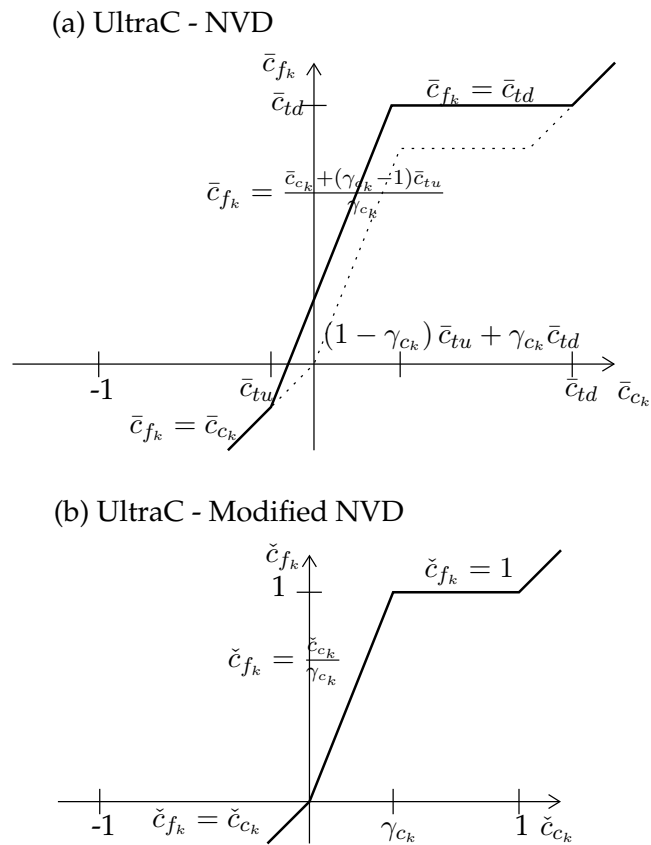


Figure 3.13: The UltraC face value scheme represented on (a) a normalised variable diagram (NVD) and (b) a modified normalised variable diagram. For comparison (a) also shows the HyperC face value scheme as a dotted line.

downwind value, c_{td} and their equivalent normalised versions:

$$\bar{c}_{tu_k} = \frac{c_{tu_k} - c_{u_k}}{c_{d_k} - c_{u_k}}, \quad (3.49)$$

and a normalised face value:

$$\bar{c}_{td_k} = \frac{c_{td_k} - c_{u_k}}{c_{d_k} - c_{u_k}}, \quad (3.50)$$

respectively. These are defined based on user prescribed target maximum and target minimum values depending on the local slope of the solution. These maximum and minimum values describe the range of values in which any solution is bounded.

The target upwind and downwind values can also be used to modify the definition of the normalised donor and face values:

$$\check{c}_{c_k} = \frac{c_{c_k} - c_{tu_k}}{c_{td_k} - c_{tu_k}}, \quad (3.51)$$

and a normalised face value:

$$\check{c}_{f_k} = \frac{c_{f_k} - c_{tu_k}}{c_{td_k} - c_{tu_k}}, \quad (3.52)$$

which allows a modified normalised variable diagram to be defined. In this case UltraC looks superficially like HyperC (see Figure 3.13(b)).

For more information on the development of UltraC in Fluidity see [Wilson \[2009\]](#).

PotentialUltraC One side effect of using a total variation bounded scheme, as in UltraC, is that small isolated regions of field that are already easily within the minimum and maximum bounds are advected at spurious velocities. This can be solved by either switching to HyperC or modifying the flux in the vicinity of these regions. PotentialUltraC implements these options. More details of this scheme can be found in [Wilson \[2009\]](#).

Coupled Limiting Under some circumstances it becomes necessary to limit the face values of a field based not only on the boundedness of the field itself but based on the boundedness of other fields as well. This is implemented in Fluidity as a coupled control volume spatial discretisation. This allows the user to select any of the above face value schemes, which are then limited to ensure that both the field and the sum of the field with the other related fields is in some sense bounded. This is particularly useful for multiple material simulations where the volume fractions must not only remain individually bounded between 0 and 1 but their sum must also be similarly bounded.

Limiting based on a summation of fields requires the user to specify a priority ordering describing the order in which the individual fields should be advected and summed. Hence we now consider a field c^I , where I indicates the priority of the field from 1 (highest priority) to the number of related fields, N (lowest priority). We then introduce the variables:

$$c^{\sum I} = \sum_{i=1}^I c^i, \quad (3.53)$$

and

$$c^{\sum I-1} = \sum_{i=1}^{I-1} c^i, \quad (3.54)$$

along with their related normalised versions. Using these variable and the total variation bounded (TVB) definition of boundedness introduced in UltraC it is possible to check whether the current field face value, c_f^I (and its normalised equivalent \bar{c}_f^I), falls within a region on the normalised variable diagram in which the summation up to the current field, $c^{\sum I}$, is itself bounded, in a TVB sense (see Figure 3.14).

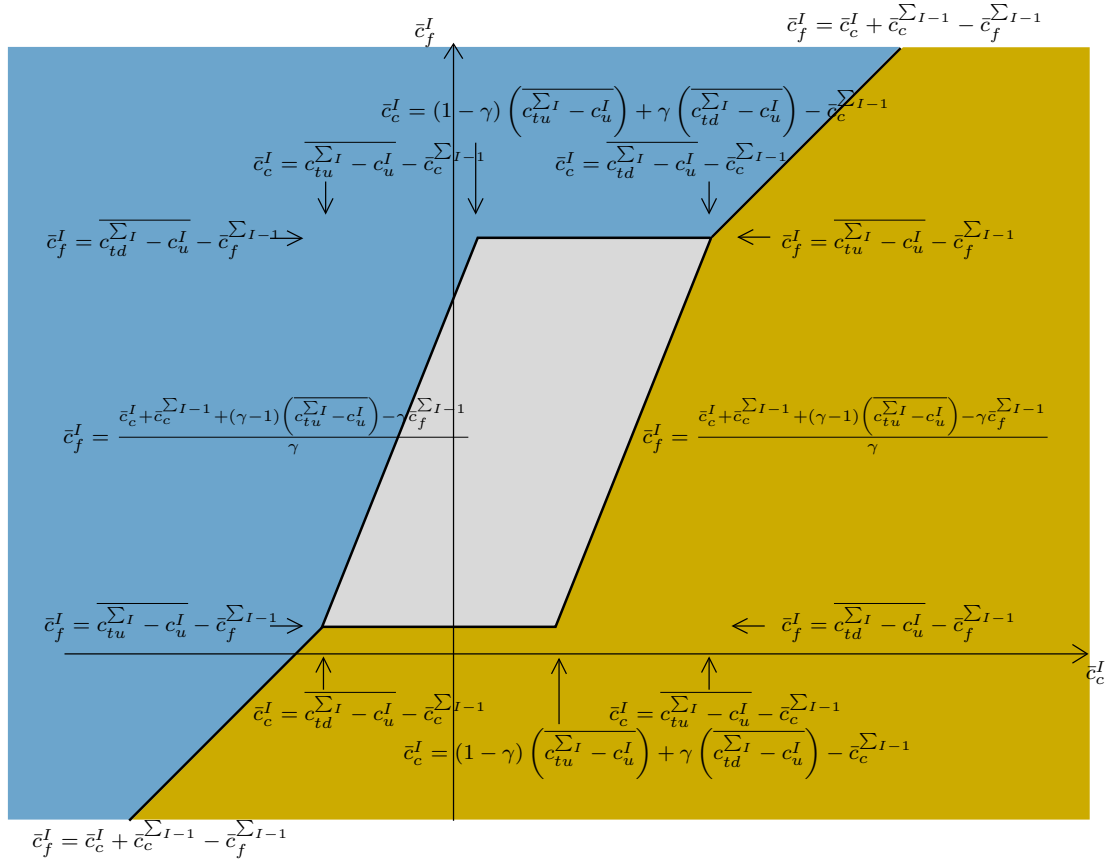


Figure 3.14: The coupled limiter for the field c^I represented by the grey shaded area on a normalised variable diagram (NVD). Labels in the upper left blue region refer to the case when the difference between the parent sum downwind and upwind values has the same sign as the limited field, $\text{sign}(c_{d_k}^{\Sigma^I} - c_{u_k}^{\Sigma^I}) = \text{sign}(c_{d_k}^I - c_{u_k}^I)$. Similarly, labels in the lower right yellow region refer to the case when the signs of the slopes are opposite, $\text{sign}(c_{d_k}^{\Sigma^I} - c_{u_k}^{\Sigma^I}) \neq \text{sign}(c_{d_k}^I - c_{u_k}^I)$. The regions are separated by the upwinding line, $\bar{c}_f^I = \bar{c}_c^I + \bar{c}_c^{\Sigma^{I-1}} - \bar{c}_f^{\Sigma^{I-1}}$.

The coupled limiter described in Figure 3.14 uses information from the previous summation of fields, $c^{\sum_{I-1}}$, to assess whether the addition of the current field will make the new summation, c^{\sum_I} unbounded. Hence on the highest priority field no additional restrictions will be imposed beyond ensuring that the field itself is TVB and the coupled limiter is simplified significantly. Subsequent fields are increasingly restricted by the constraints of previously advected fields, however only in control volumes that have values greater than zero for more than one field. If the initial state of a field or any of the higher priority fields already breaks the boundedness criterion then the coupled limiter is unable to guarantee boundedness of the sum.

Further details of the coupled control volume algorithm can be found in [Wilson \[2009\]](#).

3.2.4.2 Control Volume diffusion

In the case where diffusion, $\overline{\kappa}$, is not zero we need to discretise the term:

$$- \int_{\partial v} \mathbf{n} \cdot \widehat{\overline{\kappa}} \cdot \nabla c, \quad (3.55)$$

in (3.43). As with discontinuous Galerkin methods this is complicated by the fact that ∇c is now defined on the boundary of the volume. Fluidity offers three strategies for dealing with this with control volumes - element based gradients, equal order Bassi-Rebay and staggered mesh Bassi-Rebay discretisations.

Element Gradient As previously discussed the control volume boundaries intersect the parent elements at points where the parent basis functions are continuous. The element gradient control volume diffusion scheme uses this fact to estimate the field gradients on the control volume boundaries using the parent element basis functions. This is possible because the nodes of the parent finite element mesh and its control volume dual mesh are co-located (see Figure 3.7). This scheme is somewhat similar to standard finite volume diffusion schemes on structured meshes described in [Ciarlet and Lions \[2000\]](#).

Bassi-Rebay The Bassi-Rebay discretisation [[Bassi and Rebay, 1997](#)] method was discussed in section 3.2.3.3 for discontinuous Galerkin discretisations. It introduces an auxiliary variable and equation for the gradient. As in DG, this equation can be directly solved and implicitly reinserted into the control volume equation. However this has the disadvantage over the element gradient scheme that the sparsity structure is extended resulting in a larger matrix and more computationally expensive solves. It may also produce spurious modes in the solution.

Both disadvantages of the simplest Bassi-Rebay discretisation may be overcome for fields, c , represented on piecewise linear parent finite element meshes by selecting a piecewise constant (element centred) representation of the diffusivity. This results in the auxiliary equation for the gradient being solved for on the elements rather than at the nodes. This can still be implicitly inserted into the equation but results in the same first order sparsity structure as the element gradient scheme. Additionally the use of staggered finite/control volume meshes stabilises the equation and helps eliminate spurious modes.

3.3 The time loop

Fluidity solves a coupled system of nonlinear equations with (typically) time-varying solutions. The time-marching algorithm employed uses a non-linear iteration scheme known as Picard iteration in which each equation is solved using the best available solution for the other variables. This process

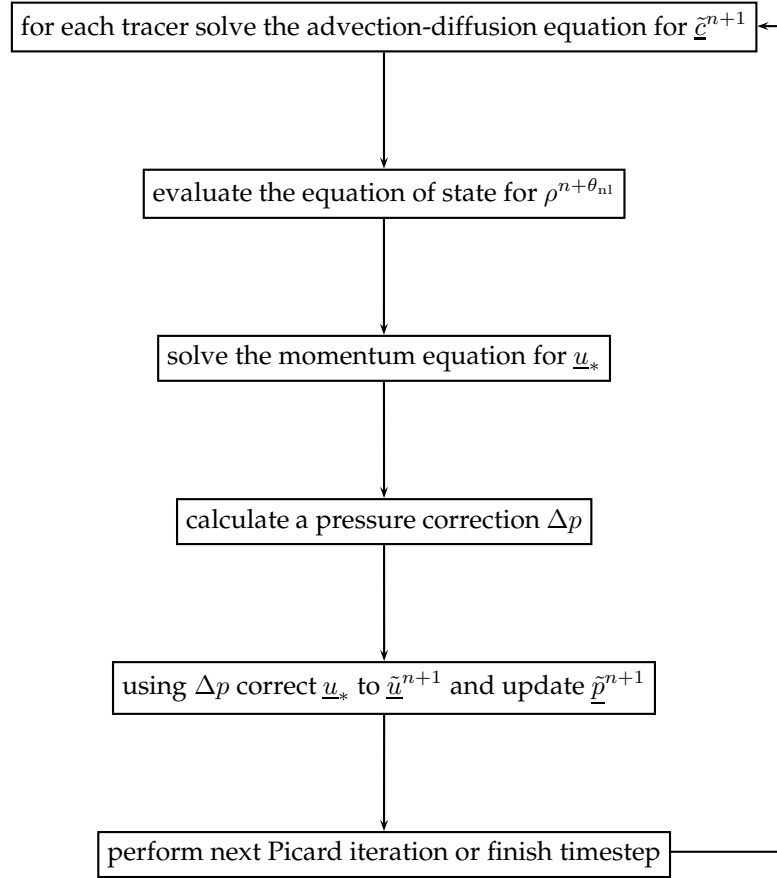


Figure 3.15: Outline of the principal steps in the nonlinear iteration sequence.

is then repeated either a fixed number of times or until convergence is achieved. Figure 3.15 shows the sequence of steps in the Picard iteration loop.

3.3.1 Time notation

It is assumed that we know the state of all variables at the n th timestep and that we wish to calculate their value at the $(n + 1)$ st step. To take as an example the general tracer denoted c , the value at the n th timestep will be denoted \underline{c}^n and that at the new timestep \underline{c}^{n+1} . The Picard iteration process results in a series of tentative results for c at the next timestep on the basis of the best available data. This tentative result will be written \tilde{c}^{n+1} . At the end of the final Picard iteration, the tentative results become final (*i.e.*, $\underline{c}^{n+1} := \tilde{c}^{n+1}$). Conversely, at the start of the timestep, the only available value of c is \tilde{c}^{n+1} so at the start of the timestep, $\tilde{c}^{n+1} := \underline{c}^n$. This notation naturally applies *mutatis mutandis* to all other variables.

3.3.2 Nonlinear relaxation

Where a variable is used in the solution of another variable, there are two available values of the first variable which might be employed: that at time n and the latest result for time $n + 1$. For instance the velocity, \underline{u} , is used in solving the advection-diffusion equation for \underline{c} so \underline{u}^n and $\tilde{\underline{u}}^{n+1}$ are available values of \underline{u} . The choice between these values is made using the nonlinear relaxation parameter θ_{nl} which must lie in the interval $[0, 1]$. This allows us to define:

$$\underline{u}^{n+\theta_{nl}} = \theta_{nl}\tilde{\underline{u}}^{n+1} + (1 - \theta_{nl})\underline{u}^n. \quad (3.56)$$

Note at the first nonlinear iteration we generally have $\tilde{\mathbf{u}}^{n+1} = \mathbf{u}^n$.

3.3.3 The θ scheme

The θ timestepping scheme requires the expression of a linear combination of the known field values at the present timestep and the as-yet unknown values at the next timestep. Taking the example of \underline{c} we write:

$$\underline{c}^{n+\theta_c} = \theta_c \tilde{\underline{c}}^{n+1} + (1 - \theta_c) \underline{c}^n. \quad (3.57)$$

θ_c must lie in $[0, 1]$. The subscript c in θ_c indicates that it is generally possible to choose different theta timestepping schemes for different solution variables.

3.4 Time discretisation of the advection-diffusion equation

Regardless of the spatial discretisation options used, the advection-diffusion equation produces a semi-discrete matrix equation of the following form:

$$M \frac{d\underline{c}}{dt} + A(\mathbf{u})\underline{c} + K\underline{c} = \underline{r}, \quad (3.58)$$

in which M is the mass matrix, $A(\mathbf{u})$ is the advection operator, K is the diffusion operator and \underline{r} is the right-hand side vector containing boundary, source and absorption terms. For continuous Galerkin, the matrices take the following form:

$$M_{ij} = \int_{\Omega} \varphi_i \varphi_j, \quad A_{ij} = - \int_{\Omega} \nabla \varphi_i \cdot \mathbf{u} \varphi_j, \quad K_{ij} = \int_{\Omega} \nabla \varphi_i \cdot \bar{\mathbf{k}} \cdot \nabla \varphi_j. \quad (3.59)$$

This is discretised using a classical θ scheme while \mathbf{u} is as given in equation (3.56):

$$M \frac{\tilde{\underline{c}}^{n+1} - \underline{c}^n}{\Delta t} + A(\mathbf{u}^{n+\theta_{nl}})\underline{c}^{n+\theta_c} + K\underline{c}^{n+\theta_c} = \underline{r}^{n+\theta_c}. \quad (3.60)$$

Here, $\underline{r}^{n+\theta_c}$ indicates that the boundary condition functions will be evaluated at $\theta_c \Delta t$ after the time at timestep n . Using equation (3.57), this can be rearranged as a single matrix equation for the unknown vector $\tilde{\underline{c}}^{n+1}$:

$$\left(M + \theta_c \Delta t \left(A(\mathbf{u}^{n+\theta_{nl}}) + K \right) \right) \tilde{\underline{c}}^{n+1} = \left(M - (1 - \theta_c) \Delta t \left(A(\mathbf{u}^{n+\theta_{nl}}) + K \right) \right) \underline{c}^n + \underline{r}^{n+\theta_c}. \quad (3.61)$$

Fluidity actually uses a somewhat different rearrangement of the equations however this is an implementation detail which has no impact for the user.

3.4.1 Discontinuous Galerkin

The slope limiters used with the discontinuous Galerkin formulation only guarantee a bounded solution in conjunction with an explicit advection scheme. While the whole equation could be treated explicitly, it can be advantageous to be able to treat only certain terms explicitly while others are treated implicitly. To achieve this, the equation is considered in two stages: first the tracer is advected, then diffusion occurs. This produces the following system:

$$M \frac{\underline{c}_* - \underline{c}^n}{\Delta t} + A(\mathbf{u}^{n+\theta_{nl}})\underline{c}^n = \underline{r}_D^{n+\theta_c} \quad (3.62)$$

$$M \frac{\tilde{\underline{c}}^{n+1} - \underline{c}_*}{\Delta t} + K\underline{c}^{n+\theta_c} = \underline{r}_N^{n+\theta_c} + \underline{r}_S^{n+\theta_c}, \quad (3.63)$$

where now \underline{r} has been split into Dirichlet and Neumann boundary components, and a source component. Equation (3.63) can be solved directly in exactly the manner of the preceding section however the explicit Euler scheme shown in equation (3.62) is subject to a tight CFL criterion. Accordingly, the timestep is split into n subimesteps to satisfy a user-specified Courant number and the following equation is solved for each:

$$M\underline{c}_{\text{new}} = \left(M - \frac{\Delta t}{n} A(\mathbf{u}^{n+\theta_{\text{nl}}}) \right) \underline{c}_{\text{old}} + \underline{r}_D^{n+\theta_c}. \quad (3.64)$$

At the start of the timestep, $\underline{c}_{\text{old}} := \underline{c}^n$ and at the end of the timestep $\underline{c}_* := \underline{c}_{\text{new}}$. Since the discontinuous Galerkin mass matrix is block diagonal with only the nodes on each element being connected, it is trivial to construct M^{-1} so this equation may be solved trivially with minimal cost. Note also that the matrix $M - (\Delta t/n)A(\mathbf{u}^{n+\theta_{\text{nl}}})$ is constant within one timestep so assembling and solving each subimestep reduces to two matrix multiplies and a vector addition.

If a slope limiter is employed, the slope limiter is applied to $\underline{c}_{\text{new}}$ after each subimestep.

3.4.2 Control Volumes

Advection subcycling based upon a CFL criterion or a fixed number of subcycles is also available for control volume discretisations, although in this case the θ value is applied globally so no advection diffusion splitting takes place. This is generally applied to explicit discretisations of the advection equation.

When face value limiting (see section 3.2.4.1) is used in implicit control volume discretisations a non-linearity is introduced by the requirement to ‘test’ a face value, c_{f_k} , against an upwind value value, c_{u_k} , which must be estimated. This restricts any such high order or limited face values to the right hand side of the equation, which severely limits the timestep that can be used. To overcome this a lower order implicit pivot face value is introduced and the face value in equation 3.43 is replaced by:

$$c_{f_k} = \theta_p c_{f_k}^{LOn+1} + \theta \tilde{c}_{f_k}^{HO} + \theta c_{f_k}^{HO n} - \theta_p \tilde{c}_{f_k}^{LO} \quad (3.65)$$

where $c_{f_k}^{LO}$ is the low order face value and $\tilde{c}_{f_k}^{HO}$ and $\tilde{c}_{f_k}^{LO}$ are the current best estimates, based on the most recent solution, of the high order and low order face values respectively [see LeVeque, 2002, for further details].

In Fluidity the low order pivot value uses first order upwinding (see section 3.2.4.1) and the pivot implicitness factor, θ_p , defaults to 1. This implicit pivot is then used to overcome the timestep restriction and an extra advection iteration loop is introduced to update the values of $\tilde{c}_{f_k}^{HO}$ and $\tilde{c}_{f_k}^{LO}$. If this converges, then after a number of iterations $c_{f_k}^{LOn+1} \approx \tilde{c}_{f_k}^{LO}$ and $c_{f_k} \approx \theta c_{f_k}^{HO n+1} + \theta c_{f_k}^{HO n}$. Hence an implicit high order face value is achieved. If the iteration does not converge, either due to too few advection iterations being selected or as a result of non-convergent behaviour in the face value limiter, then c_{f_k} is just a linear combination of low order and high order face values. This still results in a valid face value, although as is generally the case for implicit advection methods, it is not possible to guarantee the boundedness of this scheme.

3.5 Momentum equation

The discretisation of the momentum equation in non-conservative form (2.20b) is very similar to that of the advection-diffusion equation. Assuming a tensor form for viscosity, we can write it in the same matrix form as (3.8)

$$M \frac{d\mathbf{u}}{dt} + A(\mathbf{u})\underline{u} + K\underline{u} + C\underline{p} = 0, \quad (3.66)$$

with a mass matrix M , advection matrix A , viscosity matrix K and pressure gradient matrix C . For a continuous Galerkin discretisation of velocity and ignoring boundary conditions, M , A and K are given by:

$$M_{ij} = \int_{\Omega} \rho \varphi_i \cdot \varphi_j, \quad A_{ij} = \int_{\Omega} \varphi_i \cdot (\rho \mathbf{u} \cdot \nabla \varphi_j), \quad K_{ij} = \sum_{\alpha, \beta, \gamma} \int_{\Omega} (\partial_{\beta} \varphi_{i, \alpha}) \bar{\bar{k}}_{\beta \gamma} (\partial_{\gamma} \varphi_{j, \alpha}), \quad (3.67)$$

where α, β and γ are summed over the spatial dimensions. (Weak) Dirichlet and Neuman boundary conditions are implemented by surface integrals over the domain boundary in the same way as for the advection-diffusion equation. The discontinuous Galerkin discretisation is again the same as that for advection-diffusion involving additional integrals over the faces of the elements.

The pressure gradient term is new. In Fluidity it is possible to use a different set of test/trial functions for pressure and velocity — a so-called mixed element discretisation. From this section on, we will use the notation φ_i for the velocity basis functions and ψ_i for the pressure basis functions.

The pressure gradient matrix is then simply given by

$$C_{ij} = \int_{\Omega} \varphi_i \cdot \nabla \psi_j. \quad (3.68)$$

In fact this is a vector of matrices, where each matrix corresponds to one of the derivatives contained in the ∇ operator. Note that Fluidity only supports a continuous pressure discretisation so that the same pressure gradient matrix is well defined for a discontinuous Galerkin velocity.

3.5.1 Boussinesq approximation

The discretisation of the velocity equation (2.45a) in the Boussinesq approximation is given by simply dropping the density in the mass and advection terms:

$$M \frac{d\mathbf{u}}{dt} + A(\mathbf{u})\mathbf{u} + \text{Cor } \mathbf{u} - K\mathbf{u} + C\mathbf{p} = \underline{b}(\rho') + \underline{F},$$

where (again assuming CG and ignoring boundary conditions):

$$M_{ij} = \int_{\Omega} \varphi_i \cdot \varphi_j, \quad A_{ij} = \int_{\Omega} \varphi_i \cdot (\mathbf{u} \cdot \nabla \varphi_j), \quad \text{Cor}_{ij} = \int_{\Omega} \varphi_i \cdot (2\boldsymbol{\Omega} \times \varphi_j), \\ K_{ij} = \sum_{\alpha, \beta, \gamma} \int_{\Omega} (\partial_{\beta} \varphi_{i, \alpha}) \bar{\bar{u}}_{\beta \gamma} (\partial_{\gamma} \varphi_{j, \alpha}), \quad C_{ij} = \int_{\Omega} \varphi_i \cdot \nabla \psi_j, \quad \underline{b}_i(\rho') = \int_{\Omega} \varphi_i \cdot \mathbf{g} \rho', \quad \underline{F}_i = \int_{\Omega} \varphi_i \cdot \mathbf{F}$$

3.6 Pressure equation for incompressible flow

For incompressible flow the momentum equation needs to be solved in conjunction with the continuity equation $\nabla \cdot \mathbf{u} = 0$. Using test functions represented as ζ_i the discretised weak form of the continuity equation, integrated by parts, is given by

$$\sum_j \int_{\Omega} (\nabla \zeta_i) \cdot \varphi_j u_j - \int_{\partial \Omega} \zeta_i \varphi_j \cdot \mathbf{n} u_j = 0.$$

Similar to the scalar advection equation, Dirichlet boundary conditions for the normal component of the velocity can be implemented by replacing the $\mathbf{u} \cdot \mathbf{n}$ in the boundary integral by its prescribed value g_D . For this purpose we split up the boundary $\partial \Omega$ into a part $\partial \Omega_D$ where we prescribe $\mathbf{u} \cdot \mathbf{n} = g_D$ and $\partial \Omega_{\text{open}}$ where the normal component is left free.

If we define a gradient matrix as

$$B_{ij} = \int_{\Omega} \boldsymbol{\varphi}_i \cdot \nabla \zeta_j - \int_{\partial\Omega_{\text{open}}} \boldsymbol{\varphi}_i \cdot \mathbf{n} \zeta_j,$$

we can write the continuity equation in the following form:

$$B^T \underline{u} = \mathbf{M}_D \underline{g}_D, \quad \text{with } \mathbf{M}_{D,ij} = \int_{\partial\Omega_D} \zeta_i \boldsymbol{\varphi}_j \cdot \mathbf{n}. \quad (3.69)$$

If we choose the continuity equation test functions ζ_i to be the same as the pressure basis functions ψ_i the discrete continuity equation (3.69) becomes

$$C^T \underline{u} = \mathbf{M}_D \underline{g}_D, \quad \text{with } \mathbf{M}_{D,ij} = \int_{\partial\Omega_D} \psi_i \boldsymbol{\varphi}_j \cdot \mathbf{n}, \quad (3.70)$$

where we have redefined the pressure gradient matrix (3.68) as

$$C_{ij} = \int_{\Omega} \boldsymbol{\varphi}_i \cdot \nabla \psi_j - \int_{\partial\Omega_{\text{open}}} \boldsymbol{\varphi}_i \cdot \mathbf{n} \psi_j.$$

The extra surface integral over $\partial\Omega_{\text{open}}$ in C in the momentum equation will enforce a $p = 0$ boundary condition at this part of the boundary (together with the boundary condition of the viscosity term this will form a no normal stress condition). An inhomogeneous pressure boundary condition can also be applied by adding the corresponding surface integral term into the right-hand side of the momentum equation. Using the transpose of the pressure gradient operator, including its boundary terms, for the continuity equation thus automatically enforces the correct physical boundary conditions in the normal direction. The boundary conditions in the tangential direction (slip/no-slip) are independent of this choice.

For a discontinuous Galerkin (DG) discretisation of velocity with a continuous Galerkin (CG) pressure space, the integration by parts of the continuity equation means only derivatives of continuous functions are evaluated and hence no additional face integrals are required. Vice versa, for a DG pressure with a CG velocity, we simply do not integrate by parts so that again no face integrals are required. For this reason, Fluidity currently only supports mixed velocity, pressure finite element pairs where at least one of them is continuous.

3.6.1 Pressure correction

After solving the momentum equation (3.66) for the velocity using a pressure guess, the solution does not satisfy the continuity equation. A common way to enforce the continuity is to project the velocity to the set of divergence-free functions. After this projection step the continuity equation is satisfied, but in general the solution does not satisfy the momentum equation anymore, which is why the combination of momentum solve and pressure correction has to be repeated until a convergence criterium has been reached. More information about pressure correction methods can be found in [Gresho and Chan \[1988\]](#).

The derivation of the pressure-correction step starts with two variations of the time-discretisation of the momentum equation (3.66). The first one is used to solve for a preliminary \underline{u}_*^{n+1} :

$$\mathbf{M} \frac{\underline{u}_*^{n+1} - \underline{u}^n}{\Delta t} + \mathbf{A}(\mathbf{u}^{n+\theta_{nl}}) \underline{u}_*^{n+\theta} + \mathbf{K} \underline{u}_*^{n+\theta} + \mathbf{C} \underline{p}_* = 0, \quad (3.71)$$

where we use an initial guess pressure \underline{p}_* , that may be obtained from the previous time step (denoted by $\underline{p}^{n-\frac{1}{2}}$). The second variation describes the momentum equation that will be satisfied by \underline{u}^{n+1} after the pressure correction:

$$\mathbf{M} \frac{\tilde{\underline{u}}^{n+1} - \underline{u}^n}{\Delta t} + \mathbf{A}(\mathbf{u}^{n+\theta_{nl}}) \underline{u}_*^{n+\theta} + \mathbf{K} \underline{u}_*^{n+\theta} + \mathbf{C} \tilde{\underline{p}}^{n+\frac{1}{2}} = 0, \quad (3.72)$$

here $\underline{\tilde{p}}^{n+\frac{1}{2}}$ is the pressure obtained after the pressure correction.

Note that A depends on \mathbf{u} itself. For the definition of the $\mathbf{u}^{n+\theta_{n1}}$ term used to calculate A , see section 3.3.2.

Subtracting (3.71) from (3.72) yields:

$$\mathbf{M} \frac{\underline{\tilde{u}}^{n+1} - \underline{u}_*^{n+1}}{\Delta t} + \mathbf{C}(\underline{\tilde{p}}^{n+\frac{1}{2}} - \underline{p}_*) = 0, \quad (3.73)$$

Left-multiplying by $\mathbf{B}^T \mathbf{M}^{-1}$ and some rearrangement results in:

$$\mathbf{B}^T \mathbf{M}^{-1} \mathbf{C}(\underline{\tilde{p}}^{n+\frac{1}{2}} - \underline{p}_*) = - \frac{\mathbf{B}^T (\underline{\tilde{u}}^{n+1} - \underline{u}_*^{n+1})}{\Delta t},$$

The left hand side of this equation contains the sought after pressure correction $\Delta \underline{p} = \underline{\tilde{p}}^{n+\frac{1}{2}} - \underline{p}_*$. Taking into account the discretised incompressible continuity (3.69) evaluated at t^{n+1} , we finally arrive at the pressure correction equation:

$$\mathbf{B}^T \mathbf{M}^{-1} \mathbf{C} \Delta \underline{p} = \frac{\mathbf{B}^T \underline{u}_*^{n+1} - \mathbf{M}_D \underline{g}_D}{\Delta t}. \quad (3.74)$$

If we choose the continuity equation test functions ζ_i to be the same as the pressure basis functions ψ_i the discrete pressure correction equation (3.74) becomes

$$\mathbf{C}^T \mathbf{M}^{-1} \mathbf{C} \Delta \underline{p} = \frac{\mathbf{C}^T \underline{u}_*^{n+1} - \mathbf{M}_D \underline{g}_D}{\Delta t}. \quad (3.75)$$

If the continuity test functions are chosen to be the same as the pressure basis functions then the symmetric discrete Poisson equation given by (3.75) can now be solved for $\Delta \underline{p}$. If the continuity test functions are chosen to be the different to the pressure basis functions then the non symmetric discrete Poisson equation given by (3.74) can now be solved for $\Delta \underline{p}$. For the latter case to be well posed there must be the same number of continuity test functions ζ_i as pressure basis functions ψ_i such that the matrix $\mathbf{B}^T \mathbf{M}^{-1} \mathbf{C}$ is square.

After obtaining the pressure correction the pressure and velocity can be updated by:

$$\begin{aligned} \underline{\tilde{p}}^{n+\frac{1}{2}} &= \underline{p}_* + \Delta \underline{p}, \\ \underline{\tilde{u}}^{n+1} &= \underline{u}_*^{n+1} - \Delta t \mathbf{M}^{-1} \mathbf{C} \Delta \underline{p} \end{aligned}$$

Note that by construction the obtained $\underline{\tilde{u}}^{n+1}$ and $\underline{\tilde{p}}^{n+\frac{1}{2}}$ satisfy both the continuity equation (3.69) and the momentum equation (3.72). This momentum equation however still used the intermediate \underline{u}_* and \underline{p}_* . A more accurate answer can therefore be obtained using multiple non-linear iterations within a time-step. At the beginning of the subsequent non-linear iterations the pressure \underline{p}_* and the advective velocity $\underline{u}_*^{n+\theta_{n1}}$ are updated using the best available values $\underline{\tilde{p}}^{n+\frac{1}{2}}$ and $\underline{\tilde{u}}^{n+1}$ from the previous non-linear iteration.

3.6.1.1 Inverting the mass matrix

The pressure correction equation (3.74) contains the inverse of the mass matrix \mathbf{M} . For a continuous Galerkin discretisation of velocity, this inverse will result in a dense matrix, so in general we do not want to explicitly construct this. This can be avoided by approximating the mass matrix by the so-called *lumped* mass matrix:

$$\mathbf{M}_{L,ii} = \sum_k \mathbf{M}_{ik}, \quad \mathbf{M}_{L,ij \neq i} = 0.$$

This lumped mass matrix M_L replaces M in the discretised momentum equation. Since the lumped mass matrix is diagonal it is trivial to invert. This lumping procedure is more often used to avoid mass matrix inversions, for instance in Galerkin projections. The lumping procedure is conservative, but leads to a loss in accuracy.

With a discontinuous Galerkin discretisation of velocity, the mass matrix becomes easier to invert. This is because the test functions only overlap if they are associated with nodes in the same element. Therefore the mass matrix takes on a block-diagonal form, with $n \times n$ blocks along the diagonal, where n is the number of nodes per element. These blocks can be independently inverted and the inverse mass matrix still has the same block-diagonal form. As a consequence the matrix $B^T M^{-1} C$ (or $C^T M^{-1} C$) is sparse and can be explicitly constructed. Moreover for the case where the continuity test functions and pressure basis functions are the same, with a continuous P_{n+1} discretisation of pressure, and a discontinuous $P_m, m \leq n$ discretisation of velocity, we have the following property[Cotter et al., 2009]:

$$C^T M^{-1} C_{ij} = \int_{\Omega} \nabla \psi_i \cdot \nabla \psi_j.$$

That is, the discretised pressure equation (3.74) is the same as that would be obtained from a direct P_{n+1} discretisation of the continuous pressure poisson equation.

3.7 Velocity and pressure element pairs

As we have seen, various methods are available in Fluidity for the discretisation of velocity and pressure. The momentum equation can be discretised using continuous or discontinuous Galerkin for velocity, with arbitrary degree polynomials P_N . For pressure the continuous Galerkin (again with arbitrary degree P_N) and control volume (here denoted by P_{1CV}) are available. Not every available pair of velocity and pressure elements is suitable however.

An important criteria for selecting a suitable element pair, is based on the LBB stability condition. For a precise definition and detailed discussion of this condition, see Gresho and Chan [1988]. Element pairs that do not satisfy this condition suffer from spurious pressure modes. All equal order element pairs $P_N P_N$ (same order N for velocity and pressure), including the popular $P_1 P_1$, suffer from this problem. A common workaround is to add a stabilisation term to the pressure equation. This however introduces a numerical error in the continuity equation – i.e. equation (3.6) is not strictly adhered to. The fact that the discrete velocity is no longer divergence free may also have repercussions for the solution of the advection-diffusion equations.

Another consideration for choosing the right element pair is based on an analysis of the dominant terms in the momentum equation. For instance, for ocean applications the so called geostrophic balance between Coriolis, buoyancy and the pressure gradient is very important. For a balance between Coriolis and pressure gradient it is necessary that pressure itself is discretised in a higher order polynomial space than velocity, so that its gradient can match the Coriolis term. Another approach is to separate out this balance in an additional balance solve as discussed in the next section. If the viscous term is dominant in the momentum equation, a higher order velocity discretisation than pressure may be desirable. For instance in Stokes problems the $P_2 P_1$ element pair is popular.

The following table gives an overview of element pairs available in Fluidity.

- P_1P_1 Because of its simplicity this element pair in which pressure and velocity are both piecewise linear, is very popular. It does however have a number of disadvantages. Because it is not stable, a stabilisation of the pressure equation is usually required. Also in problems where buoyancy or Coriolis are dominant terms, an additional balance solve may be required (see next section).
- $P_{1DG}P_2$ This is element pair (P_{1DG} for velocity and P_2 for pressure), is highly recommended for large scale ocean applications of Fluidity. It is LBB stable and can represent the discrete geostrophic balance exactly. See [Cotter et al. \[2009\]](#) for more information on the use of this element pair in ocean applications. One of the advantages of choosing a discontinuous element pair is that the mass matrix can be inverted locally, so that the mass matrix does not have to be lumped, as explained in section 3.6.1.1.
- P_0P_1 The $P_{1DG}P_2$ pair can be extended to a family of velocity, pressure pairs $P_{N+1DG}P_N$. The P_0P_1 pair can therefore be seen as a lower order version of $P_{1DG}P_2$, which is less accurate but also cheaper to compute. Unfortunately it is known that for P_0 velocity, the viscosity schemes available in fluidity only really give an accurate answer for structured meshes.
- P_0P_{1CV} This is similar to the P_0P_1 pair but with a P_{1CV} discretisation for pressure. This has the advantage that in the advection equation for P_{1CV} tracers, the advective velocity will be exactly divergence free, as the continuity equation is tested with P_{1CV} test functions. This is therefore the element pair of choice for multimaterial runs.
- P_2P_1 This is a well known, stable element pair (also known as Taylor Hood. It does require a special mass lumping procedure. It is often used in problems with a dominant viscosity term (e.g. pure Stokes problems).

3.7.1 Continuous Galerkin pressure with control volume tested continuity

As was shown in section 3.6 the continuity test functions ζ_i need not be chosen to be the same as the pressure basis functions ψ_i . As well as being required to form a valid discrete finite element function space the continuity test function space must contain the same number of functions as the pressure basis function space. This is necessary such that the pressure correction matrix is square.

A particular choice for a different continuity test function space is to use the control volume dual space of the pressure basis standard finite element function space (for example consisting of Lagrangian functions). This is available in Fluidity when the pressure basis functions are continuous. The purpose of this choice is to select a velocity and pressure element pair to satisfy a particular balance and to then also ensure a discrete velocity divergence suitable for the transport of tracers (or volume fractions) using a control volume discretisation. An immediate drawback of this test function choice is that the pressure correction matrix is not symmetric, which must be considered when selecting a linear algebra solver.

Currently the capabilities and stability of using this method with different element pairs has not been fully established.

3.8 Balance pressure

In a balanced pressure decomposition the pressure correction equation (3.73) is modified to:

$$M \frac{\tilde{u}^{n+1} - u_*^{n+1}}{\Delta t} + C(\tilde{p}_r^{n+\frac{1}{2}} - \underline{p}_{r,*}) + C_b \underline{p}_b = 0, \quad (3.76)$$

where \underline{p}_b is some solution for the pressure field associated with buoyancy and Coriolis accelerations, \underline{p}_r is the residual pressure enforcing incompressibility, and \mathbf{C}_b is a balanced pressure gradient matrix. If a solution for \underline{p}_b can be found via some method that is more accurate than the pressure projection method used to solve for the residual pressure \underline{p}_r , then this leads to a more accurate representation of geostrophic and hydrostatic balance. In particular, for the P_1P_1 element pair, a second-order accurate solution for \underline{p}_b enables a second-order accurate solution for the Helmholtz decomposition of the velocity increment associated with the buoyancy and Coriolis accelerations, even with the introduction of pressure stabilisation.

The pressure correction equation is the Galerkin projection of the Helmholtz decomposition of the divergent velocity increment computed from the discretised momentum equation. In the continuous space, this is equivalent to the Helmholtz decomposition of the forcing terms in the momentum equation, and takes the form:

$$\mathbf{F} = \mathbf{F}_* - \nabla p, \quad (3.77)$$

$$\nabla \cdot \mathbf{F} = 0, \quad (3.78)$$

where \mathbf{F}_* are all forcing terms in the continuous momentum equation (including advection, buoyancy, Coriolis, and any other forcings). Decomposing the pressure into a component p_b associated with the buoyancy and Coriolis accelerations \mathbf{B}_* , and a residual component p_r associated with all other forcing terms $\mathbf{F}_* - \mathbf{B}_*$, yields:

$$\mathbf{B} = \mathbf{B}_* - \nabla p_b, \quad (3.79a)$$

$$\mathbf{F} - \mathbf{B} = \mathbf{F}_* - \mathbf{B}_* - \nabla p_r, \quad (3.79b)$$

$$\nabla \cdot \mathbf{B} = 0, \quad (3.79c)$$

$$\nabla \cdot (\mathbf{F} - \mathbf{B}) = 0. \quad (3.79d)$$

Taking the divergence of equation (3.79a) yields a Poisson equation for the diagnostic balanced pressure component p_b :

$$0 = \nabla \cdot \mathbf{B}_* - \nabla^2 p_b. \quad (3.80)$$

For no-slip boundary conditions on $\partial\Omega$ bounding Ω , this equation has boundary conditions $(\mathbf{B}_* + \nabla p_b) \cdot \hat{\mathbf{n}} = 0$ on $\partial\Omega$. This corresponds to no acceleration of fluid parcels in the direction normal to the boundary by the non-divergent (and dynamically significant) component of the buoyancy and Coriolis accelerations. Performing a continuous Galerkin discretisation of equation (3.80) subject to these boundary conditions yields:

$$\int_{\Omega} \nabla \xi_i \nabla \xi_j \underline{p}_b = \int_{\Omega} \nabla \xi_i \cdot \mathbf{B}_*, \quad (3.81)$$

where the ξ_i are the balanced pressure elemental basis functions. Hence equation (3.81) can be used to gain a solution for p_b , which can in turn be used in equation (3.76). Since p_b is computed via a Galerkin projection of the Poisson equation for the balanced pressure, rather than a Galerkin projection of the Helmholtz decomposition of the buoyancy and Coriolis accelerations, LBB stability constraints do not apply in the selection of a space for \underline{p}_b .

3.9 Free surface

Using the fact that the pressure is zero on the free surface and therefore $p = g\eta$, the weak form of the free surface kinematic boundary (2.31) condition is written:

$$\frac{1}{g} \int_{\Omega_{fs}} \psi_i \frac{\partial p}{\partial t} \mathbf{z} \cdot \mathbf{n} = \int_{\Omega_{fs}} \psi_i \mathbf{n} \cdot \mathbf{u} \quad \forall \psi_i, \quad (3.82)$$

and Ω_{fs} is the boundary where the free surface is to be applied. This condition is now incorporated into the boundary integrals of the continuity equation (3.6), taking the continuity test functions to be the same as the pressure basis functions, giving:

$$-\int_{\Omega} \nabla \psi_i \cdot \mathbf{u} + \int_{\partial\Omega_D} \psi_i u_n + \frac{1}{g} \int_{\partial\Omega_{FS}} \psi_i \frac{\partial p}{\partial t} \mathbf{z} \cdot \mathbf{n} + \int_{\partial\Omega \setminus \partial\Omega_D \setminus \partial\Omega_{FS}} \psi_i \mathbf{n} \cdot \mathbf{u} = 0, \quad \forall \psi_i. \quad (3.83)$$

Following the spatial and temporal discretisation described in section 3.6, the matrix form of the continuity equation now includes the free surface boundary term (compare with (3.70)):

$$\theta C^T u^{n+1} + (1 - \theta) C^T u^n + M_s \frac{\hat{p}^{n+1} - \hat{p}^n}{g \Delta t} = 0 \quad (3.84)$$

Repeating the steps from section 3.6.1, the pressure correction equation takes the form:

$$\left(\theta C^T \left(\frac{M_u}{\Delta t} \right)^{-1} \theta C + \frac{M_s}{g(\Delta t)^2} \right) \Delta \hat{p} = - \frac{\theta C^T u_*^{n+1} + (1 - \theta) C^T u^n}{\Delta t} - \frac{M_s}{g(\Delta t)^2} (\hat{p}_*^{n+1} - \hat{p}^n) \quad (3.85)$$

3.10 Wetting and drying

The discretisation of free surface boundary condition with wetting and drying is very similar to the derivation in section 3.9, but uses $p = g \max(\eta, b + d_0)$ as relationship between pressure and free surface elevation. Hence the free surface boundary condition with wetting and drying becomes (compare with equation 3.82):

$$\frac{1}{g} \int_{\Omega_{fs}} \psi_i \frac{\partial \max(p, g(b + d_0))}{\partial t} \mathbf{z} \cdot \mathbf{n} = \int_{\Omega_{fs}} \psi_i \mathbf{n} \cdot \mathbf{u} \quad \forall \psi_i, \quad (3.86)$$

where Ω_{fs} is the boundary where the free surface is to be applied.

Following the steps in section 3.82 yields the pressure correction equation with free surface and wetting and drying:

$$\begin{aligned} \left(\theta C^T \left(\frac{M_u}{\Delta t} \right)^{-1} \theta C \right) \Delta \hat{p} + \frac{M_s^{n+1, wet}}{g(\Delta t)^2} \hat{p}^{n+1} + \frac{M_s^{n+1, dry}}{(\Delta t)^2} (b + d_0) = \\ - \frac{\theta C^T u_*^{n+1} + (1 - \theta) C^T u^n}{\Delta t} + \frac{M_s^{n, wet}}{g(\Delta t)^2} \hat{p}^n + \frac{M_s^{n, dry}}{(\Delta t)^2} (b + d_0), \end{aligned} \quad (3.87)$$

where a wet (dry) superscript denotes that the matrix is assembled on only wet (dry) mesh elements.

3.11 Linear solvers

The discretised equations (such as (3.8) or (3.66)) form a linear system of equations that can be written in the following general form:

$$A \underline{x} = \underline{b},$$

where A is a matrix, \underline{x} is a vector of the values to solve for (typically the values at the nodes of a field x), and \underline{b} contains all the terms that do not depend on x . One important property of the matrices that come from finite element discretisations is that they are very *sparse*, *i.e.*, most of the entries are zero. For the solution of large sparse linear systems so called iterative methods are usually employed as they avoid having to explicitly construct the inverse of the matrix (or a suitable decomposition thereof), which is generally dense and therefore costly to compute (both in memory and computer time).

3.11.1 Iterative solvers

Iterative methods try to solve a linear system by a sequence of approximations \underline{x}^k such that \underline{x}^k converges to the exact solution $\hat{\underline{x}}$. In each iteration one can calculate the *residual*

$$\underline{r}^k = \underline{b} - A\underline{x}^k.$$

When reaching the exact solution $\underline{x}^k \rightarrow \hat{\underline{x}}$ the residual $\underline{r}^k \rightarrow 0$. The following important relation holds

$$\underline{r}^k = A (\hat{\underline{x}} - \underline{x}^k). \quad (3.88)$$

3.11.1.1 Stationary iterative methods

Suppose we have an approximation M of A , for which the inverse M^{-1} is easy to compute, and we apply this inverse to (3.88), we get the following approximation of the error in each iteration

$$\underline{e}^k = \hat{\underline{x}} - \underline{x}^k \approx M^{-1}\underline{r}^k \quad (3.89)$$

This leads to an iterative method of the following form

$$\underline{x}^{k+1} = \underline{x}^k + M^{-1}\underline{r}^k. \quad (3.90)$$

Because the same operator is applied at each iteration, these are called *stationary methods*.

A well known example is the *Jacobi* iteration, where for each row i the associated unknown x_i is solved approximately by using the values of the previous iteration for all other $x_j, j \neq i$. So in iteration k we solve for x_i^{k+1} in

$$\sum_{j<i} A_{ij}x_j^k + A_{ii}x_i^{k+1} + \sum_{j>i} A_{ij}x_j^k = b_i$$

Using the symbols L, U and D for respectively the lower and upper diagonal part, and the diagonal matrix, this is written as

$$L\underline{x}^k + D\underline{x}^{k+1} + U\underline{x}^k = \underline{b}.$$

Because when solving for x_i^{k+1} all the $x_{j<i}^{k+1}$ are already known, one could also include these in the approximate solve. This leads to the *Gauss-Seidel* iterative method

$$L\underline{x}^{k+1} + D\underline{x}^{k+1} + U\underline{x}^k = \underline{b}.$$

After some rewriting both can be written in the form of (3.90)

$$\begin{aligned} \text{Jacobi: } \underline{x}^{k+1} &= \underline{x}^k + D^{-1}\underline{r}^k, \\ \text{Gauss-Seidel forward: } \underline{x}^{k+1} &= \underline{x}^k + (L + D)^{-1}\underline{r}^k, \\ \text{Gauss-Seidel backward: } \underline{x}^{k+1} &= \underline{x}^k + (U + D)^{-1}\underline{r}^k. \end{aligned} \quad (3.91)$$

3.11.1.2 Krylov subspace methods

Another class of iterative methods are the so called *Krylov Subspace* methods. Consider the following very simple iterative method

$$\underline{x}^{k+1} = \underline{x}^k + \alpha_k \underline{r}^k,$$

where α_k is a scalar coefficient which, in contrast to that used in stationary methods, may be different in each iteration. It is then easy to show that

$$\underline{r}^{k+1} = \underline{r}^k + \alpha_k A \underline{r}^k.$$

Since therefore the residual in each iteration is the linear combination of the residual in the previous iteration and A applied to the previous residual, one can further derive that the residual is a linear combination of the following vectors:

$$\underline{r}^0, A \underline{r}^0, A^2 \underline{r}^0, \dots, A^k \underline{r}^0. \quad (3.92)$$

The subspace spanned by these vectors is called the *Krylov subspace* and *Krylov subspace* methods solve the linear system by choosing the optimal set of coefficients α_k that minimises the residual.

A well known, generally applicable Krylov method is GMRES (Generalised Minimum RESidual, see Saad [1993]). Because the approximate solution is built from all vectors in (3.92), all of these need to be stored in memory. For solves that require a large number of iterations this will become too expensive. Restarted GMRES therefore sets a maximum of those vectors to be stored (specified by the user), after reaching this maximum the corresponding coefficients are fixed and a new Krylov subspace is built. This typically leads to a temporary decay in the convergence

A very efficient Krylov method that only works for symmetric positive definite (SPD) matrices is the Conjugate Gradient (CG) method (see Shewchuk [1994] for an excellent non-expert explanation of the method). An SPD matrix is a symmetric matrix for which

$$\langle \underline{x}, A \underline{x} \rangle \geq 0, \quad \forall \underline{x} \in \mathbb{R}^n.$$

Using this property the CG method can find an optimal solution in the Krylov subspace without having to store all of its vectors. If the matrix is SPD, CG is usually the most effective choice. Linear systems to be solved in Fluidity that are SPD comprise the pressure matrix (only for incompressible flow), and the diffusion equation.

3.11.2 Preconditioned Krylov subspace methods

Because Krylov methods work by repeatedly applying the matrix A to the initial residual, eigenvectors with large eigenvalues will become dominant in the subsequent iterations, whereas eigenvectors with small eigenvalues are rapidly “overpowered”. This means the component of the error associated with small eigenvalues is only very slowly reduced in a basic Krylov method. A measure for the spread in the magnitude of eigenvalues is the so called *condition number*

$$C_A = \frac{\lambda_{\text{largest}}}{\lambda_{\text{smallest}}},$$

the ratio between the smallest and largest eigenvalue. A large condition number therefore means the matrix system will be hard to solve.

A solution to this problem is to combine the stationary methods of section 3.11.1.1 with the Krylov subspace methods of section 3.11.1.2

By pre-multiplying the equation $A \underline{x} = \underline{b}$ by the approximate inverse M^{-1} , we instead solve for

$$M^{-1} A \underline{x} = M^{-1} \underline{b}.$$

If M^{-1} is a good approximation of the inverse of A , then $M^{-1}A \approx I$ and therefore $M^{-1}A$ should have a much better condition number than the original matrix A . This way of transforming the equation to improve the conditioning of the system is referred to as preconditioning. Note that we in general do not compute $M^{-1}A$ explicitly as a matrix, but instead each iteration apply matrix A followed by a multiplication with M^{-1} , the preconditioner.

For SPD matrix systems solved with CG we can use a change of variables to keep the transformed system SPD as long as the preconditioner M^{-1} is SPD as well.

3.11.2.1 Multigrid methods

Even though simple preconditioners such as SOR will improve the conditioning of the system it can be shown that they don't work very well on systems in which multiple length scales are present. The preconditioned iterative method will rapidly reduce local variations in the error, but the larger scale, smooth error only decreases very slowly. Multigrid methods (see e.g. Trottenberg et al. [2001] for an introduction) tackle this problem by solving the system on a hierarchy of fine to coarse meshes.

Because Fluidity is based on a fully unstructured mesh discretisation only so called algebraic multigrid (AMG) methods are applicable (see e.g. Stüben [2001] for an introduction). An AMG method that in general gives very good results for most Fluidity runs is the smoothed aggregation approach by Vanek et al. [1996] (available in Fluidity as the "mg" preconditioner). For large scale ocean simulations a specific multigrid technique, called vertical lumping [Kramer et al., 2010], has been developed, that deals with the large separation between horizontal (barotropic) and vertical modes in the pressure equation.

3.11.3 Convergence criteria

In each iterative method we need some way of telling when to stop. As we have seen in equation (3.89) the preconditioner applied to the residual gives a good estimate of the error we have. So a good stop condition might be

$$\|M^{-1}\underline{r}^k\| \leq \varepsilon_{\text{atol}},$$

with a user specified *absolute tolerance* $\varepsilon_{\text{atol}}$, a small value that indicates the error we are willing to tolerate.

One problem is that we quite often don't know how big the typical value of our field is going to be (also it might change in time), so we don't know how small $\varepsilon_{\text{atol}}$ should be. A better choice is therefore to use a *relative tolerance* $\varepsilon_{\text{rtol}}$ which relates the tolerated error to a rough order estimate of the answer:

$$\|M^{-1}\underline{r}^k\| \leq \varepsilon_{\text{rtol}} \|M^{-1}\underline{b}\|. \quad (3.93)$$

In some exceptional cases the right-hand side of the equation may become zero. For instance the lock exchange problem (see 10.3) starts with an unstable equilibrium in which the righthand side of the momentum equation will be zero. The right solution in this case is of course $\underline{x} = 0$, but due to numerical round off the solver may never exactly reach this, and therefore never satisfy 3.93. In this case it is best to specify both a relative and an absolute tolerance.

Note, that the stopping criterion is always based on an *approximation* of the actual error. Especially in ill-conditioned systems this may not always be a very good approximation. The quality of the error estimate is then very much dependent on the quality of the preconditioner.

3.12 Algorithm for detectors (Lagrangian trajectories)

Detectors can be set in the code at specific positions where the user wants to know the values of certain variables at each time step. They are virtual probes in the simulation and can be fixed in space (static detectors) or can move with the flow (Lagrangian detectors). The configuration of detectors in Fluidity is detailed in chapter 8. This section summarises the method employed in the calculation of the new position of each Lagrangian detector as it moves with the flow.

Lagrangian detectors are advected using an explicit Runge-Kutta method, defined as

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \Delta t \sum_{i=1}^s b_i f_i \quad (3.94)$$

where \mathbf{x}^n denotes the value $\mathbf{x}(t^n)$,

$$f_i = f \left(\mathbf{x}^n + \Delta t \sum_{j=1}^{i-1} a_{ij} f_j, t^n + c_i \Delta t \right) \quad (3.95)$$

s is the number of stages and Δt is the timestep. The values b_i , c_i and a_{ij} are the entries of the corresponding *Butcher array*

$$\begin{array}{c|ccc} c_i & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

In combination with the previous algorithm, the Guided Search method proposed by [Coppola et al. \[2001\]](#) is used to track the detector across elements and, in parallel runs, across partitions. This method allows a detector leaving an element to be traced without resorting to an iterative procedure, and is based on the observation that each stage of the Runge-Kutta scheme can be considered as a linear substep.

Starting from an initial position P in Figure 3.16, a linear step in physical space would take the detector to position Q. For the Guided Search procedure we now evaluate the velocity field in physical space and translate the intermediate position for each RK stage to parametric space. In general, the detector will have left the element. However, we can determine the new containing element by inspecting the local coordinates of the arrival point (R for the first stage, S for the second) with respect to the element enclosing the previous intermediate position. This procedure is now repeated for all stages of the RK scheme, using the intermediate positions to sample the velocity field in their respective elements. The final position T of the detector can thus be evaluated with a fixed number of substeps.

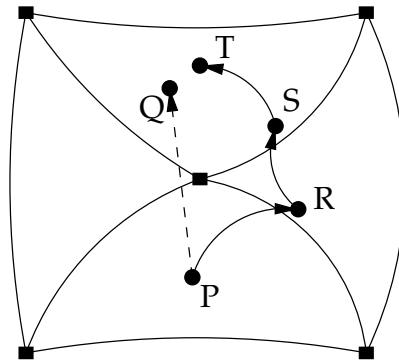


Figure 3.16: A sketch representing the Guided Search method used in combination with an explicit Runge-Kutta algorithm to advect the Lagrangian detectors with the flow.

Chapter 4

Parameterisations

Although Fluidity is capable of resolving a range of scales dynamically using an adaptive mesh, it is not always feasible to resolve all processes and spatial scales that are required for a simulation, and therefore some form of parameterisation is required. This chapter introduces the parameterisations that are available in Fluidity.

4.1 Turbulent flow modelling and simulation

4.1.1 Reynolds Averaged Navier Stokes Modelling

4.1.1.1 Generic length scale turbulence parameterisation

The generic length scale (GLS) turbulence parameterisation is capable of simulating vertical turbulence at a scale lower than that of the mesh. There is no dependency on the mesh resolution, so is ideal for adaptive ocean-scale problems. GLS has the additional advantage that it can be set-up to behave as a number of classical turbulence models: $k - \epsilon$, $k - kl$, $k - \omega$, and an additional model based on [Umlauf and Burchard \[2003\]](#), the *gen* model.

Briefly, all implementations rely on a local, temporally varying, kinematic eddy viscosity K_M that parametrises turbulence (local Reynolds stresses) in terms of mean-flow quantities (vertical shear) as, along with a buoyancy term that parameterises the kinematic eddy diffusivity, K_H :

$$\overline{u'w'} = -\nu_M \frac{\partial u}{\partial z}, \quad \overline{v'w'} = -\nu_M \frac{\partial v}{\partial z}, \quad \overline{w'\rho'} = -\nu_H \frac{\partial \rho}{\partial z}, \quad (4.1)$$

with

$$\nu_M = \sqrt{kl}S_M + \nu_M^0, \quad \nu_H = \sqrt{kl}S_H + \nu_H^0, \quad (4.2)$$

Here, we follow the notation of [Umlauf and Burchard \[2003\]](#), where u and v are the horizontal components of the Reynolds-averaged velocity along the x - and y -axes, w is the vertical velocity along the vertical z -axis, positive upwards, and u' , v' and w' are the components of the turbulent fluctuations about the mean velocity. ν_H^0 is the background diffusivity, ν_M^0 is the background viscosity, S_M and S_H are often referred to as stability functions, k is the turbulent kinetic energy, and l is a length-scale. When using GLS the values of ν_M and ν_H become the vertical components of the tensors $\overline{\tau}$ and $\overline{\kappa}_T$ in equation 2.45 respectively. Other tracer fields, such as salinity use the same diffusivity as temperature, i.e. $\overline{\kappa}_T = \overline{\kappa}_S$.

The generic length scale turbulence closure model [[Umlauf and Burchard, 2003](#)] is based on two equations, for the transport of turbulent kinetic energy (TKE) and a generic second quantity, Ψ . The

TKE equation is:

$$\frac{\partial k}{\partial t} + \mathbf{u}_i \frac{\partial k}{\partial x_i} = \frac{\partial}{\partial z} \left(\frac{\nu_M}{\sigma_k} \frac{\partial k}{\partial z} \right) + P + B - \varepsilon, \quad (4.3)$$

where σ_k is the turbulence Schmidt number for k , u_i are the velocity components (u , v and w in the x , y and z directions respectively), and P and B represent production by shear and buoyancy which are defined as:

$$P = -\overline{u'w'} \frac{\partial u}{\partial z} - \overline{v'w'} \frac{\partial v}{\partial z} = \nu_M M^2, \quad M^2 = \left(\frac{\partial u}{\partial z} \right)^2 + \left(\frac{\partial v}{\partial z} \right)^2, \quad (4.4)$$

$$B = -\frac{g}{\rho_0} \overline{\rho'w'} = -\nu_H N^2, \quad N^2 = -\frac{g}{\rho_0} \frac{\partial \rho}{\partial z} \quad (4.5)$$

Here N is the buoyancy frequency; u and v are the horizontal velocity components. The dissipation is modelled using a rate of dissipation term:

$$\varepsilon = (c_\mu^0)^{3+\frac{p}{n}} k^{\frac{3}{2}+\frac{m}{n}} \Psi^{-\frac{1}{n}} \quad (4.6)$$

where c_μ^0 is a model constant used to make Ψ identifiable with any of the transitional models, e.g. kl , ε , and ω , by adopting the values shown in Table 4.1 [Umlauf and Burchard, 2003]. The second equation is:

$$\frac{\partial \Psi}{\partial t} + \mathbf{u}_i \frac{\partial \Psi}{\partial x_i} = \frac{\partial}{\partial z} \left(\frac{\nu_M}{\sigma_\Psi} \frac{\partial \Psi}{\partial z} \right) + \frac{\Psi}{k} (c_1 P + c_3 B - c_2 \varepsilon F_{wall}), \quad (4.7)$$

The parameter σ_Ψ is the Schmidt number for Ψ and c_i are constants based on experimental data. The value of c_3 depends on whether the flow is stably stratified (in which case $c_3 = c_3^-$) or unstable ($c_3 = c_3^+$). Here,

$$\Psi = (c_\mu^0)^p k^m l^n, \quad (4.8)$$

and

$$l = (c_\mu^0)^3 k^{\frac{3}{2}} \varepsilon^{-1}, \quad (4.9)$$

By choosing values for the parameters p , m , n , σ_k , σ_Ψ , c_1 , c_2 , c_3 , and c_μ^0 one can recover the exact formulation of three standard GLS models, $k - \varepsilon$, $k - kl$ (equivalent of the Mellor-Yamada formulation), $k - \omega$, and an additional model based on Umlauf and Burchard [2003], the *gen* model (see Tables 4.1 and 4.2 for values).

Wall functions

The $k - kl$ closure scheme requires that a wall function as the value of n is positive (see Umlauf and Burchard [2003]). There are four different wall functions enabled in Fluidity. In standard Mellor-Yamada models [Mellor and Yamada, 1982], F_{wall} is defined as:

$$F_{wall} = \left(1 + E_2 \left(\frac{l}{\kappa} \frac{d_b + d_s}{d_b d_s} \right)^2 \right) \quad (4.10)$$

where $E_2 = 1.33$, and d_s and d_b are the distance to the surface and bottom respectively.

An alternative suggestion by Burchard et al. [1998] suggests a symmetric linear shape:

$$F_{wall} = \left(1 + E_2 \left(\frac{l}{\kappa} \frac{1}{\text{MIN}(d_b, d_s)} \right)^2 \right) \quad (4.11)$$

Burchard [2001] used numerical experiments to define a wall function simulating an infinitely deep basin:

$$F_{wall} = \left(1 + E_2 \left(\frac{l}{\kappa d_s} \right)^2 \right) \quad (4.12)$$

Finally, Blumberg et al. [1992] suggested a correction to the wall function for open channel flow:

$$F_{wall} = \left(1 + E_2 \left(\frac{l}{\kappa d_b} \right)^2 + E_4 \left(\frac{l}{\kappa d_s} \right)^2 \right) \quad (4.13)$$

where $E_4 = 0.25$.

Stability functions

Setting the parameters described above, *i.e.*, selecting which GLS model to use, closes the second-order moments, bar the definition of the stability functions, S_M and S_H , which are a function of α_M and α_N , defined as:

$$\alpha_M = \frac{k^2}{\varepsilon^2} M^2, \quad \alpha_N = \frac{k^2}{\varepsilon^2} N^2.$$

The two stability can be defined as:

$$S_M(\alpha_M, \alpha_N) = \frac{n_0 + n_1 \alpha_N + n_2 \alpha_M}{d_0 + d_1 \alpha_N + d_2 \alpha_M + d_3 \alpha_N \alpha_M + d_4 \alpha_N^2 + d_5 \alpha_M^2},$$

and

$$S_H(\alpha_M, \alpha_N) = \frac{n_{b0} + n_{b1} \alpha_N + n_{b2} \alpha_M}{d_0 + d_1 \alpha_N + d_2 \alpha_M + d_3 \alpha_N \alpha_M + d_4 \alpha_N^2 + d_5 \alpha_M^2}.$$

However, using the equilibrium condition for the turbulent kinetic energy as $(P + B)/\varepsilon = 1$, one can write α_M and a function of α_N , allowing elimination of α_M in the above equations [Umlauf and Burchard, 2005]:

$$S_M(\alpha_M, \alpha_N) \alpha_M - S_N(\alpha_M, \alpha_N) \alpha_N = 1$$

eliminating some of the terms. A limit on negative values of α_N needs to be applied to ensure α_M does not also become negative.

The parameters $n_0, n_1, n_2, d_0, d_2, d_3, d_4, n_{b0}, n_{b1}, n_{b2}$ depend on the model parameters chosen and can be related to traditional stability functions [Umlauf and Burchard, 2005].

Fluidity contains four choices of stability functions, GibsonLauder78 [Gibson and Launder, 1978], KanthaClayson94 [Kantha and Clayson, 1994], CanutoA and CanutoB [Canuto et al., 2001], each of which can be used in conjunction with any of $gen, k - \varepsilon$, and $k - \omega$ closures; and CanutoA and KanthaClayson94 available with the $k - kl$ closure scheme.

Boundary conditions

The boundary conditions for the two GLS equations can be either of Dirichlet or Neumann type. For the turbulent kinetic energy, the Dirichlet condition can be written as:

$$k = \frac{(u^*)^2}{(c_\mu^0)^2}, \quad (4.14)$$

where u^* is the friction velocity. However, as the viscous sublayer is not resolved, the Dirichlet condition can be unstable unless the resolution at the boundary is very high [Burchard et al., 1999]. It is therefore advisable to use the Neumann condition:

$$\frac{\nu_M}{\sigma_k} \frac{\partial k}{\partial z} = 0. \quad (4.15)$$

Similarly for the generic quantity, Ψ , the Dirichlet condition is written as:

$$\Psi = (c_\mu^0)^p l^n k^m \quad (4.16)$$

At the top of the viscous sublayer the value of Ψ can be determined from equation 4.8, specifying $l = \kappa z$ and k from equation 4.14, giving:

$$\Psi = (c_\mu^0)^{p-2m} \kappa^n (u_s^*)^{2m} (\kappa z_s)^n \quad (4.17)$$

where z_s is the distance from the boundary surface (either surface or bottom) and u_s^* is the friction at the surface or bottom respectively.

Calculating the corresponding Neumann conditions by differentiating with respect to z , yields:

$$\left(\frac{K_M}{\sigma_\Psi} \frac{\partial \Psi}{\partial z} \right) = n \frac{K_M}{\sigma_\Psi} (c_\mu^0)^p k^m \kappa^n z_s^{n-1} \quad (4.18)$$

Note that it is also an option to express the Neumann condition above in terms of the friction velocity, u_s^* . Previous work has shown this causes numerical difficulties in the case of stress-free surface boundary layers [Burchard et al., 1999].

4.1.1.2 Standard $k - \varepsilon$ Turbulence Model

Available under `.../subgridscale_parameterisations/k-epsilon`.

The widely-used $k - \varepsilon$ turbulence model has been implemented in Fluidity based on the description given in Mathieu and Scott [2000]. It is distinct from the $k - \varepsilon$ option in the generic length scale (GLS) model (see Section 4.1.1.1), in that it uses a 3D isotropic eddy-viscosity tensor, and it can be applied to any geometry. The eddy viscosity is added to the user-specified molecular (background) viscosity when solving for velocity, and if solving for additional prognostic scalar fields, it is scaled by a user-specified Prandtl number to obtain the field eddy diffusivity.

The model is based on the unsteady Reynolds-averaged Navier-Stokes (RANS) equations, in which the velocity is decomposed into quasi-steady (moving average) and fluctuating (turbulent) components:

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho} \frac{\partial p}{\partial x_j} + \nu \frac{\partial^2 u_i}{\partial x_i \partial x_j} - \frac{\partial u_i' u_j'}{\partial x_j}, \quad (4.19)$$

where u_i are the components of the steady velocity, u_i' are the fluctuating components, and p is the steady pressure. The third term on the right, containing the Reynolds stress tensor $u_i' u_j'$, represents the effect of turbulent fluctuations on the steady flow and is modelled as:

$$u_i' u_j' = -\frac{2}{3} k \delta_{ij} + \nu_T \left(\frac{\partial u_i}{\partial x_j} \frac{\partial u_j}{\partial x_i} \right), \quad (4.20)$$

where $k = u_i' u_i' / 2$ is the turbulent kinetic energy and $\nu(x, t)$ is the eddy viscosity. The equations are closed by solving transport equations for k and the turbulent energy dissipation ε :

$$\frac{\partial k}{\partial t} + u_i \frac{\partial k}{\partial x_i} = \Pi - \varepsilon + \frac{\partial}{\partial x} \left[\frac{\nu_T}{\sigma_k} \frac{\partial k}{\partial x_i} \right], \quad (4.21)$$

$$\frac{\partial \varepsilon}{\partial t} + u_i \frac{\partial \varepsilon}{\partial x_i} = C_{\varepsilon 1} \frac{\varepsilon}{k} \Pi - C_{\varepsilon 2} \frac{\varepsilon^2}{k} + \frac{\partial}{\partial x} \left[\frac{\nu_T}{\sigma_\varepsilon} \frac{\partial \varepsilon}{\partial x_i} \right], \quad (4.22)$$

where ν_T is the isotropic tensor eddy viscosity ν_T and Π is the turbulent kinetic energy production. The first terms in both equations are production (source), the second terms are absorption and the last terms are diffusion. ν_T and Π are approximated by the eddy viscosity hypothesis as:

$$\nu_T = C_\mu \frac{k^2}{\varepsilon}, \quad (4.23)$$

$$\Pi = \nu_T \frac{\partial u_i}{\partial x_j} : \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right), \quad (4.24)$$

where $:$ denotes the double-dot product, defined for two tensors t, s as:

$$t : s = \sum_{i,j=1}^n t_{ij} \times s_{ji}. \quad (4.25)$$

A turbulence length scale is associated with the dissipation of turbulent kinetic energy by the subgrid scale motions:

$$l = \frac{k^{3/2}}{\varepsilon}. \quad (4.26)$$

The five model coefficients are in Table 4.3. These are the default values but they can be changed in Diamond.

Initial and Boundary Conditions

Low- and high-Reynolds-number wall functions are available under the option `.../boundary_conditions/type::k_epsilon/` (see 4.1.1.2). When simulating low Reynolds numbers ($Re < 10^4$) the `low_Re` wall functions are recommended [Wilcox, 1998]:

$$\mathbf{u} = 0, \quad (4.27)$$

$$k = 0, \quad (4.28)$$

$$\varepsilon = \nu \frac{\partial^2 k}{\partial y^2} = 2\nu \left(\frac{\partial k^{1/2}}{\partial n} \right). \quad (4.29)$$

For high Reynolds numbers ($Re > 10^4$), the `low_Re` wall functions are recommended [Wilcox, 1998]:

$$\mathbf{u} = u_* \left[\frac{1}{\kappa} \ln(u_* y / \nu) + B \right], \quad (4.30)$$

$$k = \frac{u_*^2}{C_\mu^{1/2}}, \quad (4.31)$$

$$\varepsilon = \frac{u_*^3}{\kappa y}, \quad (4.32)$$

$$u_* = (\tau_w / \rho)^{1/2} = \left(\nu_T \frac{\partial U_S}{\partial n} \right)^{1/2}, \quad (4.33)$$

where B is a model constant, y is the distance to the wall, u_* is the friction velocity, U_S is the stream-wise velocity parallel to the wall, $\kappa = 0.43$ is Von Karman's constant, ν is the kinematic laminar (background) viscosity, ν_T is the kinematic eddy (turbulent) viscosity, ρ is the density and $\frac{\partial}{\partial n}$ is the gradient operator in the wall-normal direction. For implementation details see 8.11.2.

4.1.2 Large-Eddy Simulation (LES)

In Large Eddy Simulations the large scales in the flow are captured while the effect of the small scales is modelled. Formally, a filtering operator is defined and a decomposition, similar to the Reynolds decomposition, is introduced:

$$\bar{u}_i = \int_{-\infty}^{\infty} G_m(\vec{r}') u_i(\vec{x} - \vec{r}') d\vec{r}' \quad (4.34)$$

and

$$u_i = \bar{u}_i + u'_i \quad (4.35)$$

where u'_i denotes the subgrid-scale fluctuation. Applying the filtering operator to the continuity and momentum equations for constant-property, incompressible flow and introducing the decomposition (4.35) to the non-linear terms gives,

$$\frac{\partial \bar{u}_i}{\partial x_i} = 0 \quad (4.36)$$

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial \bar{u}_i \bar{u}_j}{\partial x_j} = -\frac{1}{\rho} \frac{\partial \bar{p}}{\partial x_i} + 2\nu \frac{\partial \bar{S}_{ij}}{\partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j} \quad (4.37)$$

where \bar{S}_{ij} denotes the strain-rate tensor of the filtered velocity field and τ_{ij} is usually termed the residual stress tensor,

$$\bar{S}_{ij} = \frac{1}{2} \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right) \quad (4.38)$$

$$\tau_{ij} = \overline{u_i u_j} - \bar{u}_i \bar{u}_j \quad (4.39)$$

The above, formal, definition of LES involves an explicit filtering operation. Various options are available for the filtering kernel $G_m(\vec{r}')$ in the convolution (4.34), see Pope [2000] and Sagaut [1998]. However, in most implementations the filtering kernel is tied to the mesh and the numerical approximation.

The residual stress tensor is an unknown and a subgrid-scale model is used to close the Navier-Stokes equations. All three subgrid-scale models implemented in Fluidity are based on the eddy-viscosity concept: The small scales in the flow act as a diffusive agent, so the subgrid stress can be expressed in a way similar to the viscous stress:

$$\tau_{ij_a} = \tau_{ij} - \delta_{ij} \frac{1}{3} \tau_{kk} = -2\nu_\tau \bar{S}_{ij} \quad (4.40)$$

where only the anisotropic part of τ_{ij} is treated explicitly. This is usual practise, as the isotropic part can be added to the pressure. In addition, this allows for the diagonal components of τ_{ij} to be non-zero when $S_{ij} = 0$. Equation (4.37) becomes:

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = -\frac{1}{\rho} \frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j} \left[(\nu + \nu_\tau) \left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right) \right], \quad (4.41)$$

In comparison with the unfiltered Navier-Stokes equations, equation (4.37) features an additional term. This is reflected in the options that must be adjusted in order to perform large eddy simulations using Fluidity. In particular, the user must activate the option with path `...Velocity/prognostic/spatial_discretisation/continuous_galerkin/les_model` (See chapter 8 for how to configure Fluidity options and interact with diamond). Note however, currently Fluidity supports “explicit” LES only with continuous Galerking discretisations. Once the aforementioned option is activated, the user can select and configure one of the three models outlined in the next section. The available models (and option paths) are:

- Second order dissipation model (`.../les_model/second_order`)
- Fourth order dissipation model (`.../les_model/fourth_order`)
- Dynamic Smagorinsky model (`.../les_model/dynamic_les`)

4.1.2.1 Subgrid-scale modelling

The subgrid-scale models available in Fluidity are based on the Smagorinsky model [Smagorinsky, 1963]. The Smagorinsky model itself is a mixing length model, where by dimensional analysis [Deardorff, 1970, Germano, 1992] the eddy viscosity is expressed as:

$$\nu_\tau = c l^{\frac{4}{3}} \varepsilon^{\frac{1}{3}} \quad (4.42)$$

where c is a dimensionless constant, l is the *Smagorinsky length-scale* (the mixing length) and ε is the rate of dissipation. The assumption that ε is equal to the production rate \mathcal{P} is then used, correct only if the cut-off frequency of the filter is placed in the inertial sub-range of the spectrum:

$$\varepsilon = \mathcal{P} = \tau_{ij_a} \overline{S}_{ij} \quad (4.43)$$

Equations (4.40), (4.42) and (4.43) give:

$$\nu_\tau = C_s^2 l^2 |\overline{S}| \quad (4.44)$$

where C_s is the *Smagorinsky coefficient* and $|\overline{S}|$ is the local strain rate (the second invariant of the strain-rate tensor):

$$|\overline{S}| = (2\overline{S}_{ij}\overline{S}_{ij})^{1/2} \quad (4.45)$$

Second-Order Dissipation model

The following model developed by Bentham [2003] is similar to the original Smagorinsky model, but allows for an anisotropic eddy viscosity that gives better results for flow simulations on unstructured grids:

$$\frac{\partial \tau_{ij}}{\partial x_j} = \frac{\partial}{\partial x_i} \left[\nu_{jk} \frac{\partial \overline{u}_j}{\partial x_k} \right], \quad (4.46)$$

with the anisotropic tensorial eddy viscosity:

$$\nu_{jk} = 4C_s^2 |\overline{S}| \mathcal{M}^{-1}, \quad (4.47)$$

where \mathcal{M} is the length-scale metric from the adaptivity process (see Pain et al. [2001]), used here to relate eddy viscosity to the local grid size. The factor of 4 arises because the filter width separating resolved and unresolved scales is assumed to be twice the local element size, which is squared in the viscosity model.

The options for this model are available under the spud path `.../les_model/second_order`. The following options are available to the user:

- `.../les_model/second_order/smagorinsky_coefficient` : (Compulsory) The user can here specify the value of the Smagorinsky coefficient C_s . A value of 0.1 is recommended for most flows. However, many researchers have carried out calibration studies for particular flows and mesh resolutions, for example see Deardorff [1971], Nicoud and Durcos [1999], Germano et al. [1991] and Canuto and Cheng [1997].

- `.../les_model/second_order/tensor_field::EddyViscosity` : (Optional) When this option is active the eddy viscosity is calculated as a diagnostic field. Further sub-options allow the user to store this field for later visualisation and post-processing.

Fourth-Order Dissipation model

The fourth-order method is designed as an improvement to the second-order eddy viscosity method, which can be too dissipative [Candy, 2008]. The fourth-order term is taken as the difference of two second-order eddy viscosity discretisations, where one is larger than the other. Usually a smaller time-step and finer grid are necessary to make fourth-order worthwhile.

Currently the only option for this model allows the specification of the Smagorinsky coefficient. This option is available at the spud path `.../les_model/fourth_order/smagorinsky_coefficient`, see the corresponding option of the second-order dissipation model above for more information.

Dynamic Smagorinsky model

The most important disadvantage of Smagorinsky-type models such as discussed in sections 4.1.2.1 and 4.1.2.1 is the behaviour of the eddy viscosity near walls and non-turbulent regions. As shown in Pope [2000], in a fully developed channel flow the sub-grid eddy viscosity ν_τ should diminish as $\nu_\tau \propto z^3$, z the wall-normal coordinate. Conversely, it is shown in Nicoud and Durcos [1999] that the Smagorinsky model gives $\sqrt{2\tilde{S}_{ij}\tilde{S}_{ij}} \sim O(1)$ near walls leading to an incorrect subgrid-scale viscosity. In addition, the Smagorinsky model is absolutely dissipative: energy transfer is only allowed from resolved scales to subgrid scales. The opposite (commonly termed as backscatter), commonly occurs in transitional flows, and absolutely dissipative models have been shown in Piomelli et al. [1990] to under-predict the growth rate of perturbations in the flow, leading to delayed transition to turbulence.

Dynamical models were designed to overcome the aforementioned disadvantages by using arguments with better physical grounding as a starting point towards the evaluation of the subgrid viscosity. In particular, at the core of dynamical Smagorinsky type models lies the idea of scale similarity [Bardina et al., 1980], which states that in the inertial subrange, the statistical properties of the fluctuations at a given wave-number are similar to the statistical properties of fluctuations of near-by wave-numbers.

Formally, a second filtering of equations (4.36) and (4.37) is introduced:

$$\tilde{\tilde{u}}_i = \int_{-\infty}^{\infty} G_t(\vec{r}) \bar{u}_i(\vec{x} - \vec{r}) d\vec{r} \quad (4.48)$$

$$\frac{\partial \tilde{\tilde{u}}_i}{\partial x_i} = 0 \quad (4.49)$$

$$\frac{\partial \tilde{\tilde{u}}_i}{\partial t} + \frac{\partial \tilde{\tilde{u}}_i \tilde{\tilde{u}}_j}{\partial x_j} = -\frac{1}{\rho} \frac{\partial \tilde{\tilde{p}}}{\partial x_i} + 2\nu \frac{\partial \tilde{\tilde{S}}_{ij}}{\partial x_j} - \frac{\partial T_{ij}}{\partial x_j} \quad (4.50)$$

The first filter is the mesh filter $G_m(x)$ (see equation (4.34)) and has a characteristic filter width $\bar{\Delta}$ in physical space. \bar{u}_i is simply the velocity represented on the computational mesh. The second filter, $G_t(x)$, is called the test filter and has a characteristic filter width $\tilde{\Delta}$. The filter widths are related by,

$$\tilde{\Delta}/\bar{\Delta} = \alpha \quad (4.51)$$

where $\alpha = 2$ for best results [Germano et al., 1991]. The residual stress resulting from mesh-filtering, τ_{ij} is given in equation (4.39). The residual stress tensor resulting from test-filtering, T_{ij} is:

$$T_{ij} = \widetilde{\tilde{u}_i \tilde{u}_j} - \tilde{\tilde{u}}_i \tilde{\tilde{u}}_j, \quad (4.52)$$

The Germano identity postulates the relation between the test-filtered τ_{ij} and T_{ij} :

$$\mathcal{L}_{ij} = T_{ij} - \tilde{\tau}_{ij} = \widetilde{\bar{u}_i \bar{u}_j} - \widetilde{\tilde{u}_i \tilde{u}_j}. \quad (4.53)$$

The Smagorinsky model is used to obtain an expression for the stress tensors on the right-hand-side of the Germano identity above. The Smagorinsky model is here written as:

$$\tau_{ij} = -2\nu_\tau \bar{S}_{ij} \quad \text{where, } \nu_\tau = C \bar{\Delta}^2 |\bar{S}| \quad (4.54)$$

Where $C = C_s^2$ is used, for convenience. The Germano identity is used in conjunction with equation (4.54) and after contraction with the strain rate tensor (4.38) an expression for the model coefficient C is derived (see Germano et al. [1991], Germano [1992] for details):

$$C(\mathbf{x}, t) = -\frac{1}{2} \frac{\mathcal{L}_{ij} \bar{S}_{ij}}{\bar{\Delta}^2 |\bar{S}| \widetilde{\bar{S}_{ij} \bar{S}} - \widetilde{\tilde{S}_{ij} \tilde{S}}}. \quad (4.55)$$

Then the eddy viscosity is given by

$$\nu_\tau(\mathbf{x}, t) = -\frac{1}{2} \frac{\mathcal{L}_{ij} \bar{S}_{ij}}{(1 + \alpha^2) |\bar{S}| \widetilde{\bar{S}_{ij} \bar{S}} - \widetilde{\tilde{S}_{ij} \tilde{S}}} |\bar{S}|. \quad (4.56)$$

Unlike the Smagorinsky-based models of Bentham [2003], the eddy viscosity is isotropic in equation (4.56), since the anisotropic filter widths have cancelled top and bottom. The anisotropy of the mesh is accounted for in the definition of the test filter $G_t(x)$.

A major drawback of the Germano variant of the dynamic Smagorinsky model is that the denominator in equations (4.55) and (4.56) can become very small. In addition, equation (4.56) can give large fluctuations in the subgrid scale viscosity, which can also lead to instability. Planar averaging of the terms in the numerator and denominator of equations (4.55) and (4.56) is used in Germano et al. [1991] to stabilise turbulent channel flow simulations. In Fluidity the Germano variant is implemented as well as the Lilly variant. In the latter, in order to remove the instability caused by the denominator of (4.55) becoming small, a modified expression for C is proposed in Lilly [1992]:

$$C(\mathbf{x}, t) = -\frac{1}{2} \frac{\mathcal{L}_{ij} M_{ij}}{M_{ij}^2}, \quad (4.57)$$

where

$$M_{ij} = \bar{\Delta} |\bar{S}| \widetilde{\bar{S}_{ij}} - \widetilde{\bar{\Delta} |\bar{S}| \bar{S}_{ij}}. \quad (4.58)$$

This represents a least-squares error minimisation of the equation $\mathcal{L}_{ij} - \frac{1}{3} \delta_{ij} \mathcal{L}_{kk} = 2CM_{ij}$. The modification is reported to remove the need for planar averaging.

The numerators of (4.55) and (4.57) can become negative, resulting in negative eddy viscosity. In this case, energy is transferred from the subgrid scales to the resolved scales. Germano et al. [1991] report that the stresses were closer to DNS as a result. This is an option in Fluidity.

Equations (4.55) and (4.57) require the calculation of the test-filtered velocity field \tilde{u}_i . The inverse Helmholtz filter is implemented in Fluidity for use with the dynamic model:

$$\bar{u}_i = (1 - \alpha^2 \nabla^2) \tilde{u}_i \quad (4.59)$$

where α is the filter width ratio defined above. It is related to the local mesh size by $\alpha^2 = \bar{\Delta}^2/24$ [Pope, 2000]. In Fluidity equation (4.59) is constructed in weak finite element form with a strong

Dirichlet boundary condition on the filtered field: $\tilde{u}_i = \bar{u}_i$. Similar problems must be solved for calculation of $\widetilde{|\overline{S}|S_{ij}}$ in equations (4.55), (4.56) and (4.58)

The options available to the user are located under path `.../les_model/dynamic_les`.

- `.../les_model/dynamic_les/alpha` : (Compulsory) The test-to-mesh filter ratio, see equation (4.51).
- `.../les_model/dynamic_les/solver` : (Compulsory) Sub-options allow the user to select the matrix solver used in solving equation (4.59), during the calculation of test-filtered fields. See section 3.11 for available linear solvers and their options.
- `.../les_model/dynamic_les/enable_lilly` : (Optional) When active, the Lilly variant of the model is used.
- `.../les_model/dynamic_les/enable_backscatter` : (Optional) When inactive the subgrid eddy viscosity is constrained to be positive, $\nu_\tau \geq 0$.
- `.../les_model/dynamic_les/vector_field::FilteredVelocity` : (Optional) Sub-options allow the user to store realisations of the twice-filtered velocity field, \tilde{u}_i $i = 1, 2, 3$, for post-processing.
- `.../les_model/dynamic_les/tensor_field::FilterWidth` : (Optional) Sub-options allow the user to store realisations of the mesh filter width field, $\tilde{\Delta}$, for post-processing.
- `.../les_model/dynamic_les/tensor_field::StrainRate` : (Optional) Sub-options allow the user to store realisations of the strain rate of the mesh-filtered velocity, for post-processing.
- `.../les_model/dynamic_les/tensor_field::FilteredStrainRate` : (Optional) Sub-options allow the user to store realisations of the strain rate of the twice-filtered velocity, for post-processing.
- `.../les_model/dynamic_les/tensor_field::EddyViscosity` : (Optional) Sub-options allow the user to store realisations of the eddy viscosity, ν_τ , for post-processing.

4.2 Ice shelf parameterisation

Exchange of heat and salt between the ice and ocean drives the circulation. The temperature T_b and salinity S_b at the ice-ocean interface are determined by the balance of heat and salt fluxes between the ice and ocean [e.g. McPhee, 2008, Jenkins and Bombosch, 1995]:

$$mL + mc_I(T_b - T_I) = c_o\gamma_T|u|(T - T_b); \quad mS_b = \gamma_S|u|(S - S_b), \quad (4.60)$$

where $c_o = 3974 \text{ J kg}^{-1} \text{ }^\circ\text{C}^{-1}$ and $c_I = 2009 \text{ J kg}^{-1} \text{ }^\circ\text{C}^{-1}$ are the specific heat capacity of seawater and ice, respectively. The variable m and $L = 3.35 \times 10^5 \text{ J kg}^{-1}$ represent the melt rate and the latent heat of ice fusion. We assume the temperature of ice to be $T_I = -25 \text{ }^\circ\text{C}$. The velocity, temperature, and salinity of the ocean are u , T , and S , respectively. These two flux balances are linked with constraining the interface to be at the local freezing temperature:

$$T_b = aS_b + b + cP, \quad (4.61)$$

where $a = -0.0573 \text{ }^\circ\text{C}$, $b = 0.0832 \text{ }^\circ\text{C}$ and $c = -7.53e^{-8} \text{ }^\circ\text{C Pa}^{-1}$. The three unknowns T_b , S_b , and m are solved by the three equations (4.60)-(4.61).

4.2.1 Boundary condition at ice surface

At the ice surface, heat and salt fluxes to the ocean based on the melt rate are

$$F_H = c_0(\gamma_T|u| + m)(T - T_b); \quad F_S = (\gamma_S|u| + m)(S - S_b). \quad (4.62)$$

These boundary conditions are applied to the surface field specified in the flml file.

Model:	$k - kl$	$k - \varepsilon$	$k - \omega$	gen
$\Psi =$	$k^1 l^1$	$(c_\mu^0)^3 k^{\frac{3}{2}} l^1$	$(c_\mu^0)^{-1} k^{\frac{1}{2}} l^1$	$(c_\mu^0)^2 k^1 l^{\frac{2}{3}}$
p	0.0	3.0	-1.0	2.0
m	1.0	1.5	0.5	1.0
n	1.0	-1.0	-1.0	-0.67
σ_k	2.44	1.0	2.0	0.8
σ_Ψ	2.44	1.3	2.0	1.07
c_1	0.9	1.44	0.555	1.0
c_2	0.5	1.92	0.833	1.22
c_3^-	See Table 4.2			
c_3^+	1.0	1.0	1.0	1.0
k_{\min}	5.0×10^{-6}	7.6×10^{-6}	7.6×10^{-6}	7.6×10^{-6}
Ψ_{\min}	1.0×10^{-8}	1.0×10^{-12}	1.0×10^{-12}	1.0×10^{-12}
F_{wall}	See sec 4.1.1.1	1.0	1.0	1.0

Table 4.1: Generic length scale parameters

Model:	$k - kl$	$k - \varepsilon$	$k - \omega$	gen
$\Psi =$	$k^1 l^1$	$(c_\mu^0)^3 k^{\frac{3}{2}} l^1$	$(c_\mu^0)^{-1} k^{\frac{1}{2}} l^1$	$(c_\mu^0)^2 k^1 l^{\frac{2}{3}}$
KC	2.53	-0.41	-0.58	0.1
CA	2.68	-0.63	-0.64	0.05
CB	-	-0.57	-0.61	0.08
GL	-	-0.37	-0.492	0.1704

Table 4.2: Values for the c_3^+ parameter for each combination of closure scheme and stability function

Coefficient	Standard value
C_μ	0.09
$C_{\varepsilon 1}$	1.44
$C_{\varepsilon 2}$	1.92
σ_ε	1.3
σ_k	1.0

Table 4.3: $k - \varepsilon$ model coefficients

Chapter 5

Embedded models

The parameterisations described in chapter 4 are used to model fluids processes which cannot be resolved by the model. In contrast, the models described in this chapter detail models embedded in Fluidity which model non-fluids processes.

5.1 Biology

The Biology model in Fluidity contains a number of different submodels. All are currently population level models where variables evolve under an advection-diffusion equation similar to that for other tracers, such as temperature and salinity, but modified by the addition of a source term which contains the interactions between the biological fields. The fluxes depend on which model is selected and which tracers are available.

There are two models currently distributed with Fluidity: a four-component model and a six-component model.

5.1.1 Four component model

Figure 5.1 shows the interactions between the biological tracers in the four component model. Nutrients (plus sunlight) are converted into phytoplankton. Zooplankton feed on phytoplankton and detritus but in doing so produce some nutrients and detritus so grazing also results in fluxes from phytoplankton and detritus to nutrient and detritus. Both phytoplankton and zooplankton die producing detritus and detritus is gradually converted to nutrient by a remineralisation process.

5.1.1.1 Biological source terms

The source terms for phytoplankton (P), zooplankton (Z), nutrient (N) and detritus (D) respectively are given by the following expressions:

$$S_P = R_P - G_P - De_P, \quad (5.1)$$

$$S_Z = \gamma\beta(G_P + G_D) - De_Z, \quad (5.2)$$

$$S_N = -R_P + De_D + (1 - \gamma)\beta(G_P + G_D), \quad (5.3)$$

$$S_D = -De_D + De_P + De_Z + (1 - \beta)G_P - \beta G_D. \quad (5.4)$$

The definitions of each of these terms are given below. It is significant that the right-hand sides of these equations sum to zero. This implies that, for a closed biological system, the total of the

Symbol	Meaning	Typical value	Section
R_P	phytoplankton growth rate		5.1.1.1
G_P	rate of zooplankton grazing on phytoplankton		5.1.1.1
De_P	death rate of phytoplankton		5.1.1.1
G_D	rate of zooplankton grazing on detritus		5.1.1.1
De_Z	death rate of zooplankton		5.1.1.1
De_D	death rate of detritus		5.1.1.1
I	photosynthetically active radiation		5.1.3
α	sensitivity of phytoplankton to light	$0.015 \text{ m}^2 \text{ W}^{-1} \text{ day}^{-1}$	5.1.1.1
β	assimilation efficiency of zooplankton	0.75	
γ	zooplankton excretion parameter	0.5	
μ_P	phytoplankton mortality rate	0.1 day^{-1}	5.1.1.1
μ_Z	zooplankton mortality rate	0.2 day^{-1}	5.1.1.1
μ_D	detritus remineralisation rate	0.05 day^{-1}	5.1.1.1
g	zooplankton maximum growth rate	1 day^{-1}	5.1.1.1
k_N	half-saturation constant for nutrient	0.5	5.1.1.1
k	zooplankton grazing parameter	0.5	5.1.1.1
p_P	zooplankton preference for phytoplankton	0.75	5.1.1.1
v	maximum phytoplankton growth rate	1.5 day^{-1}	5.1.1.1

Table 5.1: Meanings of symbols in the biology model. Typical values are provided for externally set parameters.

biological constituents is always conserved. The terms in these equations are given in table 5.1. The variable terms are explained in more detail below.

R_P , the phytoplankton growth rate

R_P is the growth-rate of phytoplankton which is governed by the current phytoplankton concentration, the available nutrients and the available light:

$$R_P = J P Q, \quad (5.5)$$

where J is the light-limited phytoplankton growth rate which is in turn given by:

$$J = \frac{v\alpha I}{(v^2 + \alpha^2 I^2)^{1/2}}. \quad (5.6)$$

In this expression, v is the maximum phytoplankton growth rate, α controls the sensitivity of growth rate to light and I is the available photosynthetically active radiation.

Q is the nutrient limiting factor and is given by:

$$Q = \frac{N}{k_N + N}, \quad (5.7)$$

where k_N is the half-saturation constant for nutrient.

G_P , the rate of phytoplankton grazing by zooplankton The rate at which zooplankton eat phytoplankton is given by:

$$G_P = \frac{gp_P P^2 Z}{k^2 + p_P P^2 + (1 - p_P) D^2}, \quad (5.8)$$

in which p_P is the preference of zooplankton for grazing phytoplankton over grazing detritus, g is the maximum zooplankton growth rate and k is a parameter which limits the grazing rate if the concentration of phytoplankton and detritus becomes very low.

G_D , the rate of detritus grazing by zooplankton The rate at which zooplankton eat detritus is given by:

$$G_D = \frac{g(1 - p_P) D^2 Z}{k^2 + p_P P^2 + (1 - p_P) D^2}, \quad (5.9)$$

in which all of the parameters have the meaning given in the previous section.

De_P , the phytoplankton death rate

A proportion of the phytoplankton in the system will die in any given time interval. The dead phytoplankton become detritus:

$$De_P = \frac{\mu_P P^2}{P + 1}, \quad (5.10)$$

in which μ_P is the phytoplankton mortality rate.

De_Z , the zooplankton death rate

A proportion of the zooplankton in the system will die in any given time interval. The dead zooplankton become detritus:

$$De_Z = \mu_Z Z^2. \quad (5.11)$$

De_D , the detritus remineralisation rate

As the detritus falls through the water column, it gradually converts to nutrient:

$$De_D = \mu_D D. \quad (5.12)$$

5.1.2 Six-component model

The six-component model is based on the model of Popova et al. [2006] which is designed to be applicable globally. Figure 5.2 shows the interactions between the six biological tracers. Nutrients, either ammonium or nitrate, (plus sunlight) are converted into phytoplankton. Zooplankton feed on phytoplankton and detritus but in doing so produce some nutrients and detritus so grazing also results in fluxes from phytoplankton and detritus to nutrient and detritus. Both phytoplankton and zooplankton die producing detritus and detritus is gradually converted to nutrient by a re-mineralisation process. In addition, chlorophyll is present as a subset of the phytoplankton.

The source terms for phytoplankton (P), Chlorophyll-a (Chl), zooplankton (Z), nitrate (N), ammonium (A), and detritus (D) respectively are given by the following expressions:

$$S_P = PJ(Q_N + Q_A) - G_P - De_P, \quad (5.13)$$

$$S_{Chl} = (R_P * J * (Q_N + Q_A) * P + (-G_P - De_P)) * \theta / \zeta, \quad (5.14)$$

$$S_Z = \delta * (\beta_P * G_P + \beta_D * G_D) - De_Z, \quad (5.15)$$

$$S_N = -J * P * Q_N + De_A, \quad (5.16)$$

$$S_A = -J * P * Q_A + De_D + (1 - \delta) * (\beta_P * G_P + \beta_D * G_D) + (1 - \gamma) * De_Z - De_A, \quad (5.17)$$

$$S_D = -De_D + De_P + \gamma * De_Z + (1 - \beta_P) * G_P - \beta_D * G_D \quad (5.18)$$

The terms in these equations are given in table

5.2. The variable terms are explained in more detail below.

5.1.2.1 Biological source terms

Unlike the model of Popova et al. [2006] we use a continuous model, with no change of equations (bar one exception) above or below the photic zone. For our purposes, the photic zone is defined as 100m water depth. First we calculate θ :

$$\theta = \frac{Chl}{P\zeta} \quad (5.19)$$

However, at low light levels, Chl might be zero, therefore we take the limit that $\theta \rightarrow \zeta$ at low levels ($1e^{-7}$) of chlorophyll-a or phytoplankton.

We then calculate α :

$$\alpha = \alpha_c \theta \quad (5.20)$$

Using the PAR available at each vertex of the mesh, we now calculate the light-limited phytoplankton growth rate, J :

$$J = \frac{v \alpha I_n}{\sqrt{v^2 + \alpha^2 + I_n^2}} \quad (5.21)$$

The limiting factors on nitrate and ammonium are now calculated:

$$Q_N = \frac{N \exp^{-\Psi A}}{K_N + N}, \quad (5.22)$$

$$Q_A = \frac{A}{K_A + A} \quad (5.23)$$

From these the diagnostic field, primary production (X_P), can be calculated:

$$X_P = J (Q_N + Q_A) P \quad (5.24)$$

The chlorophyll growth scaling factor is given by:

$$R_P = Q_N Q_A \left(\frac{\theta_m}{\theta} \right) \left(\frac{v}{\sqrt{v^2 + \alpha^2 + I_n^2}} \right) \quad (5.25)$$

The zooplankton grazing terms are now calculated:

$$G_P = \frac{g p_P P^2 Z}{\varepsilon + (p_P P^2 + p_D D^2)}, \quad (5.26)$$

$$G_D = \frac{g p_D D^2 * Z}{\varepsilon + (p_P P^2 + p_D D^2)} \quad (5.27)$$

Finally, the four death rates and re-mineralisation rates are calculated:

$$De_P = \frac{\mu_p P^2}{P + k_p} + \lambda_{\text{bio}} * P, \quad (5.28)$$

$$De_Z = \frac{\mu_z Z^3}{Z + k_z} + \lambda_{\text{bio}} * Z, \quad (5.29)$$

$$De_D = \mu_D D + \lambda_{\text{bio}} * P + \lambda_{\text{bio}} * Z, \quad (5.30)$$

$$De_A = \lambda_A A \text{ where } z < 100 \quad (5.31)$$

5.1.3 Photosynthetically active radiation (PAR)

Phytoplankton depends on the levels of light in the water column at the frequencies useful for photosynthesis. This is referred to as photosynthetically active radiation. Sunlight falls on the water surface and is absorbed by both water and phytoplankton. This is represented by the following equation:

$$\frac{\partial I}{\partial z} = (A_{\text{water}} + A_P P) I, \quad (5.32)$$

where A_{water} and A_P are the absorption rates of photosynthetically active radiation by water and phytoplankton respectively.

This equation has the form of a one-dimensional advection equation and therefore requires a single boundary condition: the amount of light incident on the water surface. This is a Dirichlet condition on I .

5.1.4 Detritus falling velocity

Phytoplankton, zooplankton and nutrient are assumed to be neutrally buoyant and therefore move only under the advecting velocity of the water or by diffusive mixing. Detritus, on the other hand, is assumed to be denser than water so it will slowly sink through the water-column. This is modelled by modifying the advecting velocity in the advection-diffusion equation for detritus by subtracting a sinking velocity u_{sink} from the vertical component of the advecting velocity.

5.2 Sediments

Fluidity is capable of simulating an unlimited number of sediment concentration classes. Each class has a separate grain size, density and settling velocity. The sediment behaves as any other tracer field, except it is subject to a settling velocity:

$$\frac{\partial S_m}{\partial t} + \nabla \cdot (\mathbf{u}S_m) = \nabla \cdot (\bar{k}\nabla S_m) - \sigma S_m \quad (5.33)$$

The advection in the vertical direction is then modified with a downwards sinking velocity u_{sink} .

5.2.1 Deposition and erosion

A surface can be defined, the sea-bed, which is a sink for sediment. Once sediment fluxes through this surface it is removed from the system and stored in a separate field: the SedimentFlux field. Each sediment class has an equivalent Flux field.

Erosion occurs when the bed-shear stress is greater than the critical shear stress. Each sediment class has a separate shear stress, which can be input or calculated depending on the options chosen. Erosion flux, E_m is implemented as a Neumann boundary condition on the deposition/erosion surface.

$$E_m = E_{0m} (1 - \varphi) \frac{\tau_{sf} - \tau_{cm}}{\tau_{cm}} \quad (5.34)$$

where E_{0m} is the bed erodibility constant ($\text{kgm}^{-1}\text{s}^{-1}$) for sediment class m , τ_{sf} is the bed-shear stress, φ is the bed porosity (typically 0.3) and τ_{cm} is the critical shear stress for sediment class m . The critical shear stress can be input by the user or automatically calculated using:

$$\tau_{cm} = 0.041 (s - 1) \rho g D \quad (5.35)$$

where s is the relative density of the sediment, i.e. $\frac{\rho_{S_m}}{\rho}$ and D is the sediment diameter (mm). The SedimentFlux field effectively mixes the deposited sediment, so order of deposition is not preserved.

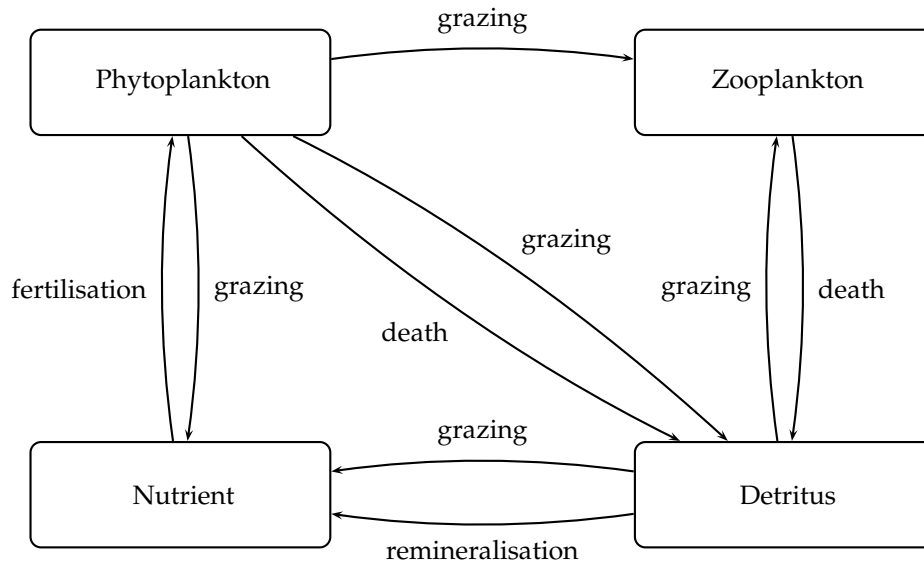


Figure 5.1: The fluxes between the biological tracers. Grazing refers to the feeding activity of zooplankton on phytoplankton and detritus.

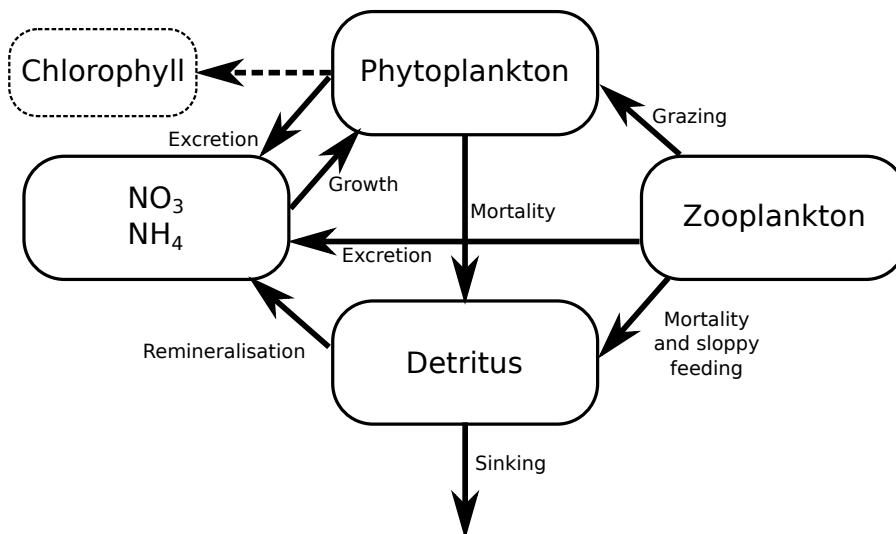


Figure 5.2: Six-component biology model.

Symbol	Meaning	Typical value	Equation
α	initial slope of $P - I$ curve, $(W m^{-2})^{-1} day^{-1}$		(5.20)
α_c	Chl-a specific initial slope of $P - I$ curve	$2 (gCgChl^{-1}Wm^{-2})^{-1}day^{-1}$	
β_P, β_D	assimilation coefficients of zooplankton	0.75	
De_D	rate of breakdown of detritus to ammonium		(5.30)
De_P	rate of phytoplankton natural mortality		(5.28)
De_Z	rate of zooplankton natural mortality		(5.29)
De_A	ammonium nitrification rate		(5.31)
δ	excretion parameter	0.7	
ε	grazing parameter relating capture of prey items to prey density	0.4	
G_P	rate of zooplankton grazing on phytoplankton		(5.26)
G_D	rate of zooplankton grazing on detritus		(5.27)
g	zooplankton maximum growth rate	$1.3 day^{-1}$	
γ	fraction of zooplankton mortality going to detritus	0.5	
I_0	photosynthetically active radiation (PAR) immediately below surface of water. Assumed to be 0.43 of the surface radiation		
J	light-limited phytoplankton growth rate, day^{-1}		(5.21)
k_A	half-saturation constant for ammonium uptake	$0.5 mmol m^{-3}$	
k_N	half-saturation constant for nitrate uptake	$0.5 mmol m^{-3}$	
k_P	half-saturation constant for phytoplankton mortality	$1 mmol m^{-3}$	
k_Z	half-saturation constant for zooplankton mortality	$3 mmol m^{-3}$	
k_w	light attenuation due to water	$0.04 m^{-1}$	
k_c	light attenuation due to phytoplankton	$0.03 m^2 mmol^{-1}$	
λ_{bio}	rate of the phytoplankton and zooplankton transfer into detritus	$0.05 day^{-1}$	
λ_A	nitrification rate	$0.03 day^{-1}$	
μ_P	phytoplankton mortality rate	$0.05 day^{-1}$	
μ_Z	zooplankton mortality rate	$0.2 day^{-1}$	
μ_D	detritus reference mineralisation rate	$0.05 day^{-1}$	
Ψ	strength of ammonium inhibition of nitrate uptake	$2.9 (mmol m^{-3})^{-1}$	
p_P	relative grazing preference for phytoplankton	0.75	
p_D	relative grazing preference for detritus	0.25	
Q_N	non-dimensional nitrate limiting factor		(5.22)
Q_A	non-dimensional ammonium limiting factor		(5.23)
R_P	Chl growth scaling factor		(5.25)
v	Maximum phytoplankton growth rate	$1 day^{-1}$	
w_g	detritus sinking velocity	$10 m day^{-1}$	
z	depth		
θ	Chl to carbon ratio, $mg Chl mgC^{-1}$		
θ_m	maximum Chl to carbon ratio	$0.05 mg Chl mgC^{-1}$	
ζ	conversion factor from gC to $mmolN$ based on C:N ratio of 6.5	$0.0128 mmolN ngC^{-1}$	

Table 5.2: Meanings of symbols in the six-component biology model. Typical values are provided for externally set parameters.

Chapter 6

Meshes in Fluidity

In each run of Fluidity an input mesh needs to be provided. Even in adaptive mesh runs an initial mesh is needed to define the initial condition of the fields on. This chapter covers a number of topics related to meshes in Fluidity: what mesh formats are supported (section 6.1), how different regions and boundary surfaces can be marked (section 6.2), the relation between meshes and function spaces (section 6.3), the extrusion of horizontal meshes in the vertical (section 6.4), the creation of periodic meshes (section 6.5), and finally an overview of meshing tools provided with Fluidity (section 6.6).

6.1 Supported mesh formats

Fluidity supports two mesh file formats:

1. Gmsh .msh files. Gmsh is a mesh generator freely available on the web at <http://geuz.org/gmsh/>, and is included in Linux distributions such as Ubuntu. This is the recommended file format.
2. Triangle format. This is stored as a set of 3 files: a .node file, a .ele file and a .face (3D) or .edge (2D) or .bound (1D) file. This file format is mainly supported for special purposes, like 1D meshes, and some offline tools.

For a detailed, technical description of the two mesh file formats see appendix E. In addition, instructions on how to generate a mesh using Gmsh in simple geometries as well as complex ocean domains can be found in the Gmsh tutorial available at <http://amcg.ese.ic.ac.uk/files/amcg-gmsh-tutorial.pdf>

6.2 Surface and regions ids

Surface ids are used in Fluidity to mark different parts of the boundary of the computational domain so that different boundary conditions can be associated with them. Regions ids are used to mark different parts of the domain itself. Both ids can be defined in Gmsh by assigning physical ids to different geometrical objects. In two dimensions surface ids can be defined by assigning a physical id to each group of lines that make up a part of the boundary that needs to be considered separately. Region ids can be defined by dividing the domain up in different (groups of) surfaces and assigning different physical ids to them. Similarly, in three dimensions surface ids are defined by assigning physical surface ids in Gmsh, and regions ids by assigning physical volume ids.

It is recommended, and required in parallel, that all parts of the domain boundary are marked with a surface id. Region ids are optional. They can be used to instruct the adaptivity library to strictly

maintain the interface between different regions of the domain. They are also useful to set different constant field values in the regions.

6.3 Meshes and function spaces

The choice of computational mesh directly controls how the fields (velocity, pressure, temperature, etc.) of the model are discretised. For example, using a triangular mesh with a P_1 continuous Galerkin discretisation, the field values are stored in the vertices of the mesh and the field is linearly interpolated inside the triangles. For a discontinuous P_{1DG} discretisation the field values are stored in the three corners of each triangle separately. The locations of these field values are referred to as nodes. As another example, the nodes of the P_2 discretisation correspond to the triangle vertices, and the midpoints of the edges.

This combination of the set of polygons (triangles, tetrahedrals, quadrilaterals, etc.) covering the domain and a choice of polynomial order and continuity between elements (P_1 , P_{1DG} , P_2 , etc.) therefore defines the discrete function space in which the approximate discrete fields are found, and the nodes in which the field values are stored. In Fluidity the term mesh is often used to refer to this specific combination, and fields are said to be defined on such meshes. In other words “meshes” in Fluidity correspond to the (polynomial) discrete function spaces.

Both simplicial meshes (triangles in 2D and tetrahedrals in 3D) and cubical meshes (quadrilaterals in 2D and hexagons in 3D) are supported. Note however that the choice between these two types of input meshes also has an influence on the function space. For simplicial meshes, a polynomial order of one means linear polynomials are used (P_1), whereas in cubical meshes this leads to bilinear (trilinear in 3D) polynomials (known as the Q_1 discretisation). Therefore to use the P_1P_1 or the $P_{1DG}P_2$ discretisation a simplicial input mesh is needed. Structured meshes in two or three dimensions can be generated by stacking two triangles in a rectangle, or six tetrahedrals in a cube respectively. Such meshes are easily obtained using Gmsh.

In mixed finite element discretisations, where different fields are discretised using different function spaces (again $P_{1DG}P_2$ is a clear example), a mesh needs to be defined for every function space that is used. In Fluidity, the input mesh is always considered to be a continuous linear mesh (P_1 for simplicial and Q_1 for cubical meshes), as its vertices correspond to the nodes of the P_1/Q_1 discretisation. For a $P_{1DG}P_2$ discretisation we therefore need two additional meshes: a P_{1DG} mesh and P_2 mesh. In section 8.3.3.2 it is explained how these can be derived from the input mesh.

In finite element discretisations the actual shape of the elements in physical space is given by a function from local coordinates on the element to coordinates in physical space, the coordinate map. This map is typically linear, resulting in elements with straight edges. For some applications, for example ocean models on the sphere, it may be desirable to use higher order polynomials however, so that large elements can be bended to fit the geometry. The mesh (function space) that is used for the coordinate map is referred to as the coordinate mesh. For standard, uncurved elements the coordinate mesh simply corresponds to the (linear) input mesh (except for periodic meshes, see section 6.5 below).

6.4 Extruded meshes

Given a 1D or 2D input mesh, Fluidity can extrude this mesh to create a layered 2D or 3D mesh, on which simulations can be performed. The extrusion is always downwards (in the direction of gravity), and the top of the domain is always flat, corresponding to the $y = 0$ -level in 2 dimension, the $z = 0$ level in 3 dimensions, or the equilibrium free surface geoid when running on the sphere.

The advantage of this approach is that the user can provide a horizontal mesh, that has been created

in the normal way (usually Gmsh), and all the configuration related to bathymetry, number of layers and layer depths can be done in the `.flml` (see section 8.4.2.4 for all options available). It also enables the application of mesh adaptivity in the horizontal and vertical independently. This means we can choose to apply adaptivity in the horizontal only and keep a fixed number of layers, or we can choose to keep the horizontal mesh fixed and dynamically adjust the vertical grid spacing (vertical adaptivity). The combination of both horizontal and vertical adaptivity is referred to as “2+1D” adaptivity. For the configuration of this kind of adaptivity see section 8.18.2.1.

6.5 Periodic meshes

Periodic meshes are those that are “virtually” connected on one side of the domain to the boundary on the opposite side. This can be done in one or more directions. To make a periodic mesh you must first create an input mesh where the edges on the two sides that will be connected can be mapped exactly by a simple transformation. For a 2d mesh that is periodic in the x -direction this simply means all the nodes on left and right boundary need to be at the same y -height. For 3d meshes, not only the nodes on the two sides need to be “periodic”, but also the edges connecting them. For a simple box geometry, this can be easily accomplished using the `create_aligned_mesh` script in the tools folder.

Alternatively, if you require a more complex periodic mesh with some structure between the periodic boundaries you can create one using `gmsh`. This can be achieved by setting up the periodic boundaries by using `extrude` and then deleting the ‘internal’ mesh.

The use of periodic domains requires additional configuration options. See section 8.4.2.3.

6.6 Meshing tools

There are a number of meshing tools in tools directory. These can be built by running `make fltools` in the top directory of the Fluidity trunk. The following binaries will then be created in the `bin/` directory (see section 9.3.3).

6.6.1 Mesh Verification

The `checkmesh` tool can be used to form a number of verification tests on a mesh in triangle mesh format. More information can be found in section 9.3.3.1.

6.6.2 Mesh creation

The `interval` tool (section 9.3.3.17) generates a 1D line mesh in triangle format.

The `gen_square_meshes` tool (section 9.3.3.14) will generate triangle files for multiple 2D square meshes with a specified number of internal nodes.

The `create_aligned_mesh` tool (section 9.3.3.3) creates the triangle files for a mesh that lines up in all directions, so that it can be made it into a singly, doubly or triply periodic mesh.

6.6.3 Mesh conversion

The `gmsh2triangle` tool (section 9.3.3.15) converts ASCII Gmsh mesh files into triangle format.

The `triangle2vtu` tool (section 9.3.3.35) can be used to convert triangle format files into `vtu` format.

The `gmsh_mesh_transform` script (section 9.3.3.16) applies a coordinate transformation to a region of a given mesh.

6.6.4 Decomposing meshes for parallel

To run in parallel one must first divide the mesh into a number of blocks, a process commonly referred to as *mesh decomposition*. Figure 6.1 shows a possible partition of an example mesh. When Fluidity is run in parallel (see section 1.4.2), each processor then reads the mesh corresponding to a single block. This section outlines how to decompose a mesh with Fluidity tools.

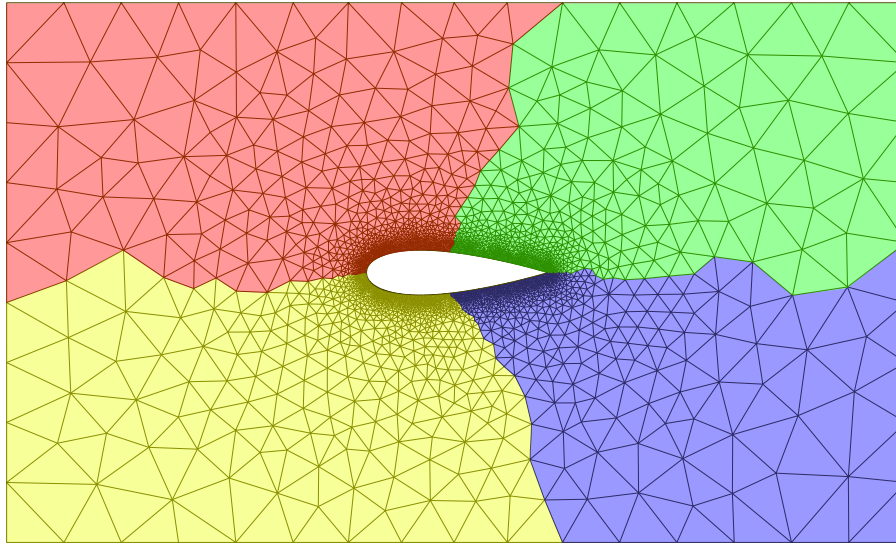


Figure 6.1: An unstructured mesh around a NACA0025 aerofoil, the coloured patches show a possible decomposition into 4 partitions.

6.6.4.1 fldecomp

`fldecomp` can be used to decompose a mesh. For example, if your mesh file is `foo.msh` and you want to decompose into four parts, type:

```
<<fluidity_source_path>>/bin/fldecomp -n 4 -m gmsh foo
```

See section 9.3.3.10 for more details.

6.6.4.2 flrecomp

`flrecomp` is a tool similar to `fldecomp` but runs in parallel. It performs a re-decomposition of a Fluidity checkpoint. For example, to decompose the serial file `foo.flml` into four parts, running on 4 processors type:

```
mpiexec -n 4 <<fluidity_source_path>>/bin/flrecomp \  
-i 1 -o 4 foo foo_flrecomp
```

The output of running `flrecomp` is a series of mesh and `vtu` files as well as the new `flml`; in this case `foo_flrecomp.flml`. Note that `flrecomp` must be run on a number of processors equal to the larger number of processors between input and output. More information can be found in section 9.3.3.12.

6.6.5 Decomposing a periodic mesh

To be able to run Fluidity on a periodic mesh in parallel you have to use two tools:

- `periodise` (section 9.3.3.22)
- `flrecomp` (section 9.3.3.12)

The input to `periodise` is your `flml` (in this case `foo.flml`). This `flml` file should already contain the mapping for the periodic boundary as described in section 8.4.2.3. `Periodise` is run with the command:

```
<<fluidity_source_path>>/bin/periodise foo.flml
```

The output is a new `flml` called `foo_periodised.flml` and the periodic meshes. Next run `flrecomp` (section 9.3.3.12) to decompose the mesh for the number of processors required. The `flml` output by `flrecomp` is then used to execute the actual simulation:

```
mpiexec -n [number of processors] \  
  <<fluidity_source_path>>/bin/fluidity [options] \  
  foo_periodised_flrecomp.flml
```

6.7 Non-Fluidity tools

In addition to the tools and capabilities of Fluidity, there are numerous tools and software packages available for mesh generation. Here, we describe two of the tools commonly used.

6.7.1 Terreno

Terreno uses a 2D anisotropic mesh optimisation algorithm to explicitly optimise for element quality and bathymetric approximation while minimising the number of mesh elements created. The shoreline used in the mesh generation process is the result of a polyline approximation algorithm that where the minimum length of the resulting edges is considered as well as the distance an edge is from a vertex on the original shoreline segment being approximated. The underlying philosophy is that meshing and approximation should be error driven and should minimise user intervention. The latter point is two pronged: usability is paramount and the user should not need to be an expert in mesh generation to generate high quality meshes for their ocean model; fundamentally there must be clearly defined objectives to the mesh generation process to ensure reproducibility of results. The result is an unstructured mesh, which may be anisotropic, which focuses resolution where it is required to optimally approximate the bathymetry of the domain. The criterion to judge the quality of the mesh is defined in terms of clearly defined objectives. An important feature of the approach is that it facilitates multi-objective mesh optimisation. This allows one to simultaneously optimise the approximation to other variables in addition to the bathymetry on the same mesh, such as backscatter data from soundings, material properties or climatology data.

See the [Terreno website](#) for more information.

6.7.2 Gmsh

Gmsh is a 3D finite element mesh generator with a build-in CAD engine and post-processor. Its design goal is to provide a fast, light and user-friendly meshing tool with parametric input and

advanced visualisation capabilities. Gmsh is built around four modules: geometry, mesh, solver and post-processing. The specification of any input to these modules is done either interactively using the graphical user interface or in ASCII text files using Gmsh's own scripting language.

For more information see the [Gmsh website](#) or the [AMCG website](#). An online manual is available at geuz.org/gmsh/doc/texinfo/gmsh.html.

6.7.3 Importing contours from bathymetric data into Gmsh

Gmsh can be used to create a mesh of a 'real' ocean domain for use with Fluidity. An online guide to using Gmsh's built in GSHHS plug-in is available at [gmsh.ocean](#). It is also possible to import contours from arbitrary bathymetry data sources into Gmsh. A guide and sample code detailing this process will in the future be available on the [AMCG website](#).

Chapter 7

Adaptive remeshing

7.1 Motivation

Historically, numerical analysts concerned themselves with *a priori* error bounds of particular numerical schemes, i.e. asymptotic analyses of the order of convergence of a discretisation with respect to some discretisation parameter such as mesh sizing h or polynomial order p . However, such *a priori* error bounds do not provide useful estimates of the simulation error of a particular physical system on a particular mesh for a specified norm: they merely describe how that error behaves as the discretisation is modified. Since such *a priori* error bounds involve the unknown exact solution, they are, in general, not computable.

In the late 1970s, the pioneering work of Babuška and Rheinboldt laid the foundations for *a posteriori* error estimates [Babuška and Rheinboldt, 1978a,b]. In contrast to *a priori* bounds, *a posteriori* error estimates involve only the approximate computed solution and data from the problem, and are thus computable (or approximately so, if they involve the solution of an auxiliary problem). These error estimates can then be used in an adaptive loop, modifying the discretisation until some user-specified error criterion is reached. Most *a posteriori* error estimation literature deals with estimating the error in the natural norm induced by the bilinear form of the problem, the energy norm. For a review of *a posteriori* error estimation with emphasis on energy norm estimation see the books of Verfürth [Verfürth, 1996] and Ainsworth and Oden [Ainsworth and Oden, 2000]. The goal-oriented adaptive framework of Rannacher and co-workers, which estimates the error in the computation of a given goal functional, is detailed in Becker and Rannacher [2001] and Bangerth and Rannacher [2003].

Once *a posteriori* estimates have been computed, there are many possible ways of modifying the discretisation to achieve some error target. These include h -adaptivity, which changes the connectivity of the mesh [Berger and Colella, 1989]; p -adaptivity, which increases the polynomial order of the approximation [Babuška and Suri, 1994]; and r -adaptivity, which relocates the vertices of the mesh while retaining the same connectivity [Budd et al., 2009]. Combinations of these methods are also possible (e.g., Houston and Süli [2001], Ledger et al. [2003]).

This review focuses on adaptive remeshing in multiple dimensions, as this is the technology available in Fluidity. This approach is the most powerful of all hr -adaptive methods, since the meshes produced are not constrained by the previous mesh; therefore, this approach allows for maximum flexibility in adapting to solution features. However, this flexibility comes at a cost: guiding the adaptive remeshing procedure (choosing what mesh to construct), executing the adaptation (constructing the chosen mesh) and data transfer of solution fields (from the previous mesh to the newly adapted mesh) become more complicated than with hierarchical refinement.

This chapter is divided as follows. Firstly, we orient the user by describing a typical adaptive loop used in Fluidity (section 7.2). Before discussing how to configure mesh adaptivity, we introduce some necessary background ideas: section 7.3 describes how a metric tensor field may be used to

encode the desired mesh, and section 7.4 describes how the mesh optimisation procedure in Fluidity uses this metric tensor to generate an adapted mesh. Finally, the chapter ends with a discussion of how to use mesh adaptivity in Fluidity, with a discussion of the various algorithms offered and their advantages and disadvantages (section 7.5).

7.2 A typical adaptive loop

This section presents a typical adaptive loop, as used in the lock–exchange example, section 10.3 (examples/lock_exchange/lock_exchange.flml).

1. The initial conditions of the prognostic variables (velocity, temperature) are applied.
2. The adaptive procedure is invoked 6 times to adapt the initial mesh to represent the desired initial conditions. This involves:
 - (a) Computing the Hessian of velocity (which is zero, as the initial velocity condition is zero).
 - (b) Converting the Hessian of velocity into a metric tensor (in this case, the tensor will request the maximum edge length size everywhere, as velocity can be represented exactly). This depends on the norm chosen and the minimum and maximum edge length sizes.
 - (c) Computing the Hessian of temperature (which is nonzero, as the initial temperature condition is a step-function).
 - (d) Converting the Hessian of temperature into a metric tensor (in this case, the tensor will request fine resolution at the step, and the maximum edge length size everywhere else).
 - (e) The size requirements of these two metrics are merged to give one metric tensor describing the desired mesh.
 - (f) The metric is then smoothed with a gradation algorithm to avoid large jumps in mesh sizing.
 - (g) The metric is scaled so that the expected number of nodes it will yield after adapting is no more than the maximum number of nodes.
 - (h) The metric is then passed to the adaptivity algorithm, which modifies the mesh until it satisfies the sizing requirements given in the metric.
 - (i) Since the simulation time has not yet advanced, the initial conditions are reapplied on the newly-generated adapted mesh.

This procedure is iterated 6 times so that the adaptive algorithm can recognise the anisotropy of the step.

3. Once the initial mesh has converged, the simulation time loop proceeds. The lock–exchange simulation invokes the adaptive algorithm every 10 timesteps to ensure that the dynamics do not extend beyond the zone of adapted resolution. Each invocation of the adaptive algorithm involves:
 - (a) Computing the Hessians of velocity and temperature (both of which will be nonzero).
 - (b) Converting the Hessians to metrics (again depending on the norms chosen and the minimum and maximum edge length sizes).
 - (c) Merging the metrics for each field to yield a combined metric which satisfies both of their size demands.
 - (d) The metric is then smoothed and scaled, as before, and passed to the adaptivity library.
 - (e) Once the adaptivity library is finished, the field data from the previous mesh is transferred to the new mesh by interpolation.

7.3 Representing meshes as metric tensors

The process of adaptive remeshing divides naturally into three parts. The first is to decide what mesh is desired; the second is to actually generate that mesh; the third is to transfer data from the old mesh to the new mesh. The form of communication between the first two stages is a *metric*: a symmetric positive-definite tensor field which encodes the desired geometric properties of the mesh. This representation was first introduced in Vallet [1990], and has proven key to the success of adaptive remeshing methods. This section describes the mathematical details of how a metric encodes the desired size and shape information: however, understanding this section is not necessary to successfully use adaptivity in Fluidity.

Firstly, consider how the size and shape information about a mesh might be encoded. A natural first attempt would be to define a size function $h(x)$ which describes the desired mesh spacing at a particular point. This works, and is used in many isotropic adaptive algorithms. However, its deficiency becomes clear when we consider the anisotropic examples that we wish to exploit: consider again the step function initial condition of the lock–exchange example. In this case, we want the mesh sizing across the step to be small (to capture the jump), but we want the mesh sizing along the step to be large (parallel to the step, the field does not change at all). Therefore, the *mesh sizing desired is not only a function of space, it is also a function of direction*. At a given point, the desired mesh sizing differs in different directions. This cannot be encoded in a scalar size function, as that associates only one number with each point in space. The solution to this problem is to increase the rank of the object returned by the size function, from a rank-0 single real value to a rank-2 tensor.

Let M be a symmetric positive-definite tensor (constant over the domain for now). M induces an inner product by

$$\langle x, y \rangle_M = x^T M y \quad (7.1)$$

and so M also induces a notion of distance between two points in the usual manner:

$$d_M(x, y) = \|x - y\|_M = \sqrt{\langle x - y, x - y \rangle}. \quad (7.2)$$

Note that the symmetric positive-definite condition is necessary for the tensor to induce a sense of distance: for otherwise, the tensor does not induce a norm.

Now let \overline{M} be a symmetric positive-definite tensor field. As long as \overline{M} is sufficiently smooth, then it also induces an inner product (see Simpson [1994] for details). We say that a mesh \mathcal{T} satisfies \overline{M} if every edge has unit edge length with respect to its inner product; \mathcal{T} is a unit mesh with respect to this metric.

The metric *warps space*, in the same manner that space is warped in General Relativity. The metric encodes a new sense of distance, and a regular unit mesh (a mesh of edge length one) is built with respect to this new sense of distance. For example, if the metric encodes that two points are “far apart”, then when a unit mesh is built, many mesh nodes will lie between them, and the mesh concentration will be dense. Conversely, if the metric encodes that two points are “close together”, then when a unit mesh is built, few mesh nodes will lie between them and the mesh concentration will be sparse.

In the adaptive procedure employed in Fluidity, the metric passed to the adaptivity library encodes the desired mesh sizing. That is, the adaptive procedure is a function that takes in a metric and returns a mesh matching that metric. Conversely, it is also possible to derive the metric for a given mesh. In order to see how the metric representing a mesh may be derived, consider the transformation from the reference element to the physical element inherent in the finite element assembly procedure [Imp, 2009]. Let K be an element of a linear simplicial mesh. K is geometrically characterised by the affine map $T_K : \hat{K} \rightarrow K$, where \hat{K} is the reference isotropic element. Since T_K is assumed affine, we can write this transformation as

$$x = T_K(\hat{x}) = J_K \hat{x} + t_K, \quad (7.3)$$

where J_K is the Jacobian of the transformation. The metric for this element is derived from the polar decomposition of the Jacobian, which yields

$$J_K = M_K Z_K \quad (7.4)$$

where M_K is a symmetric positive-definite matrix referred to as the metric, and Z_K is orthogonal [Micheletti and Perotto, 2006]. The geometric properties of a simplicial mesh can be represented as a piecewise constant (P_0) field over it. The same argument applies to nonlinear or non-simplicial elements, but here the Jacobian varies over the element, and so the metric does too.

The book of George and Borouchaki [1998] gives a thorough discussion of the role of metrics in mesh adaptivity. F. Alauzet and co-workers have recently published some thought-provoking papers advocating that metrics are in fact the continuous analogue of meshes: in the same way as the continuous equations are discretised to yield the discrete equations, a continuous metric is discretised to give a mesh on which those equations are solved. For more details, see e.g. Alauzet et al. [2006], Courty et al. [2006], Alauzet et al. [2008], Loseille and Alauzet [2011a,b].

7.4 Adaptive remeshing technology

This section briefly reviews the second problem solved by the adaptivity algorithm: given the desired size and shape information, how can a mesh satisfying this be generated?

Given a metric \overline{M} , there are three main ways to generate a mesh which satisfies that metric: global remeshing, local remeshing, and mesh optimisation. For a thorough discussion of the history of these three approaches, see Farrell [2009].

Global remeshing is the act of generating an entirely new mesh of the same domain satisfying the sizing specification, by means of an automatic mesh generator. This was first introduced in Peraire et al. [1987] by incorporating sizing and directionality inputs to an advancing front mesh generator. While this is the fastest approach if the existing mesh and the desired mesh are very different, developing a robust metric-aware mesh generator is very difficult, and to the best of the author's knowledge no open-source algorithm is available for this problem.

By contrast, local remeshing is a mechanism of adaptive remeshing in which cavities of elements are removed and the hole remeshed [Hassan et al., 1998]. These cavities are identified by measuring their conformity to the given sizing specification. Again, this relies on the availability of a robust metric-aware mesh generator to fill the cavities.

Mesh optimisation is a mechanism of adaptive remeshing which deforms the previous mesh to the adapted mesh by a sequence of local operations. The first ingredient in a mesh optimisation algorithm is the functional: how the quality of an element is determined, and how one element is deemed better than another. In the adaptive remeshing case, the functional is the point at which the metric comes in to play: the basic job of the functional is to measure the compatibility of the element to the metric specifying the size and shape of that element. Many different functionals have been considered in the literature: for a review, see Knupp [2003]. For the functional used in the 2D case, see Vasilevskii and Lipnikov [1999]; for the functional used in the 3D case, see Pain et al. [2001].

The second ingredient in a mesh optimisation algorithm is the set of operations the algorithm may perform in an attempt to improve the mesh. Example operations might include merging adjacent elements, splitting an element into two or more child elements, swapping edges or faces, and moving nodes in the mesh. For diagrams of some example optimisation operations see figure 7.1.

The mesh optimisation library provisionally executes one or more optimisation operations, and then invokes the functional to decide whether those operations improved the mesh. If the mesh was improved, then the proposed changes are committed; otherwise, they are reverted. The scheduling

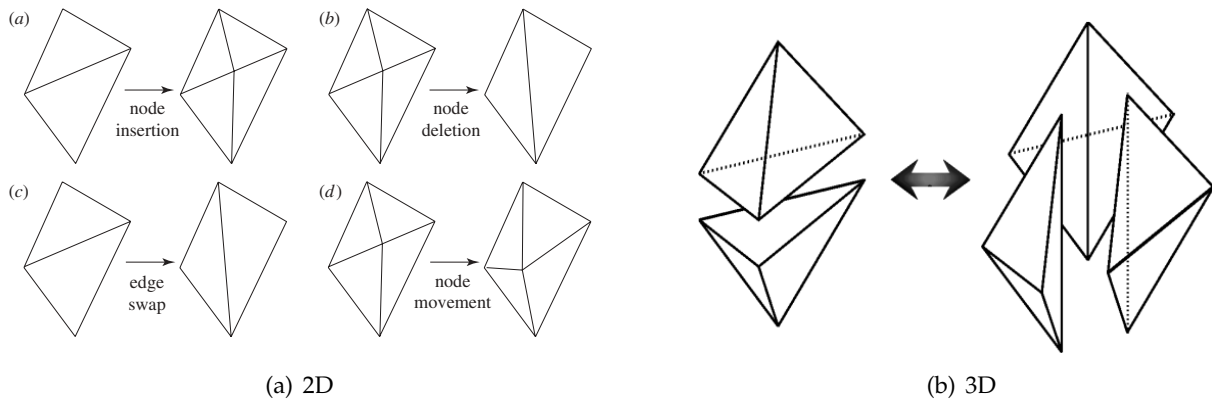


Figure 7.1: Example of mesh modification operations. (a): Node insertion or edge splitting, node deletion or edge collapse, edge swap and node movement in two dimensions [2D, from Piggott et al., 2009]. (b): Edge to face and face to edge swapping in three dimensions [3D, from Pain et al., 2001].

of the operations has a large impact on the effectiveness of the procedure [Li et al., 2005], and ensuring that the optimisation operations are robust is quite delicate [Compère et al., 2010].

Mesh optimisation is available by default in three dimensions using the algorithm of Pain et al. [2001]. It is available in two dimensions using the `mba2d` algorithm of Vasilevskii and Lipnikov [1999] if Fluidity was configured with the `--enable-2d-adaptivity` flag, section 1.3.2; this is not enabled by default as `mba2d` is licensed under the GPL, while Fluidity is licensed under the LGPL, and so its default inclusion would cause licensing complications.

7.5 Using mesh adaptivity

This section contains the practical advice for configuring and optimising adaptive simulations. Further information about the configuration options can be found in section 8.18

The first golden rule is to start with a fixed-mesh simulation that works (i.e., gives sensible answers for the resolution used). Mesh adaptivity is a complex extra nonlinear operation bolted on to the discretisation, and it will make reasoning about and debugging the simulation more difficult. Therefore, make sure that you are happy with the well-posedness of the problem, the stability of the discretisation, the convergence of the solutions, etc. before attempting to employ adaptivity.

Secondly, be aware that employing adaptivity can require a nontrivial amount of tuning to get exactly the result desired. Be prepared to iterate on the adaptivity settings, and make changes conservatively, one parameter at a time. As you gain more experience of the adaptivity options offered in Fluidity, it will become easier to identify problems and how they can be solved.

As a first step, configure Fluidity to adapt only to one field, usually the most important in the simulation. The adaptivity options for each field are found in the `.../adaptivity_options` option under each field. As the initial settings, choose `absolute_measure` (section 7.5.2), and set `p_norm` to 2 (section 7.5.1). This configures the metric to optimise for the L_2 norm of interpolation error. Under the `InterpolationErrorBound`, set the value to a constant, and as an initial guess use 10% of the L_2 norm of the field under consideration (this value is available in the `.stat` file). See section 7.5.3 for more information about this setting. Unless you are running a simulation with DG fields, leave the interpolation algorithm as `consistent_interpolation`; if you have DG fields, set their interpolation algorithm to `galerkin_projection`. This is explained in further detail in section 7.6.

Field-specific adaptivity options are configured under the field in question, and non-field-specific options are configured in `/mesh_adaptivity/hr_adaptivity`. It is recommended that the operator first configure the adaptivity options so that it is only invoked at the very start of the simulation

(using the `adapt_at_first_timestep` option), and only once this is configured satisfactorily move on to dynamic adaptivity. There are several reasons for this. Firstly, since the adaptive algorithm is invoked at the start of the simulation, the time between configuration and feedback is lessened, making it easier for the operator to get a feel for the effect of the different settings. Secondly, since the system state at the initial condition is known exactly, the complicating effects of interpolation algorithms are excised. Thirdly, accurately representing the initial condition is usually a requirement for a sensible answer at later simulation times, and so configuring the initial adaptivity is necessary anyhow.

For the initial adaptive configuration, set the period in timesteps to 20, and the maximum number of nodes to be 200000 times the number of processors you are using. (This is deliberately set high so that it does not influence the mesh produced; later on, it can be turned down to control the computational requirements of the simulation. See section 7.5.6.) Under `/mesh_adaptivity/hr_adaptivity/enable_gradation`, set the `gradation_parameter` to 2. This allows the edge length to double from element to element, and is quite a weak gradation parameter. (Again, we wish to minimise the influence of the gradation algorithm on the mesh produced; if necessary, the metric can be smoothed by reducing this value. See section 7.5.4.) Under `/mesh_adaptivity/hr_adaptivity/tensor_field::MaximumEdgeLengths/anisotropic_symmetric/constant`, set the *diagonal* entries to the diameter of your domain. (The maximum edge length can be anisotropic, in the same way that the desired edge length can be anisotropic. See section 7.5.5.) Similarly, under `.../tensor_field::MinimumEdgeLengths/anisotropic_symmetric/constant`, set the *diagonal entries* to something small (e.g., the diameter of the domain divided by 10000). Again, we wish to remove the influence of these settings by default, so that their effect can be applied by the operator if it is desired.

Activate `/mesh_adaptivity/hr_adaptivity/adapt_at_first_timestep`, and set the `number_of_adapts` to 6. Under `/mesh_adaptivity/hr_adaptivity/debug`, activate the `write_adapted_state` option. This ensures that after each invocation of the adaptivity algorithm the mesh is dumped, so that the convergence of the adaptive algorithm for the initial condition may be inspected. Finally, configure the timestepping options so that the simulation terminates after one timestep, so that the adaptivity settings may be configured for the initial condition.

Now run the problem, and inspect the `adapted_state*.vtu` that result. Each VTU is the *output* of the adaptivity loop, and hopefully convergence of the adaptive procedure towards some mesh should be observed. If the adapted mesh is satisfactory, then record the adaptivity parameters used for this field, and repeat the procedure on any other fields to which you wish to adapt that have nontrivial boundary conditions. Once you are happy with the initial mesh, run the simulation further and inspect the output of the adaptivity library, tuning the parameters to get the optimal mesh. (In this case, you may wish to cache the output of the adaptation to the initial condition with `/mesh_adaptivity/hr_adaptivity/adapt_at_first_timestep/output_adapted_mesh`.) The following sections offer guidance on the parameters that can be varied and the effect that this has on the adaptive simulation.

7.5.1 Choice of norm

The basic strategy used to compute the error metric is to *control the interpolation error* of the function represented. Let $f(x)$ be a continuous, exact field, and let $f_h(x)$ be its representation on the finite element mesh. Then, the interpolation error is defined as $|f - f_h|$, which is itself a function over the domain Ω . The “size” of this interpolation error may be quantified in different ways, using different norms. Historically, the interpolation error was first controlled in the L_∞ norm, which considers the maximum value of the interpolation error over the domain. The metric formulation which controls the L_∞ norm is the simplest, and remains the default in Fluidity. Since the L_∞ norm considers only the least accurate point in the domain, without regard to the relative size of the interpolation error there, then it can have a tendency to focus the resolution entirely on the dynamics of largest

magnitude. Other authors have instead proposed the use of the L_p norm, which incorporates more influence from dynamics of smaller magnitude [Alauzet et al., 2008, Loseille and Alauzet, 2011b]. Empirical experience indicates that choosing $p = 2$, and hence the L_2 norm, generally gives better results, and for that reason we recommend it as the default for all new adaptivity configurations.

7.5.2 Absolute, relative and p - metrics

Consider again the metric for controlling the L_∞ norm of the interpolation error. This metric takes the form

$$M = \frac{|H|}{\varepsilon}, \quad (7.5)$$

where H is the Hessian of the field under consideration and ε is the target interpolation error. As mentioned in the previous section, this metric tends to neglect smaller-scale dynamics, and so Castro-Díaz et al. [1997] proposed an alternative metric formulation to fix this. They suggested to compute

$$M = \frac{|H|}{\max(\varepsilon \cdot |f|, \varepsilon_{\min})}, \quad (7.6)$$

where f is the field under consideration, ε is now a *relative tolerance*, and ε_{\min} is a user-configurable parameter. For example, if $\varepsilon = 0.01$, then the tolerance on the denominator of the metric formulation will be 1% of the value of the field, and so it will scale the target interpolation error with the magnitude of the field. ε_{\min} is the minimum tolerance, and is employed to ensure that the denominator never becomes zero. However, empirical experience indicates that this metric formulation is very sensitive to the value of ε_{\min} , and that it generally yields poor results. The approach of using the L_p norm instead of the relative L_∞ norm is much more mathematically rigorous, and for this reason the relative metric option is deprecated.

The metric that controls the L_p norm of the interpolation error takes the form [Chen et al., 2007, Loseille and Alauzet, 2011b]

$$M = \det |H|^{-\frac{1}{2p+n}} \frac{|H|}{\varepsilon}, \quad (7.7)$$

where $p \in \mathbb{Z}$ and n is the dimension of the space.

The options for the metric are selected by field and can be found under `name_of_field/adaptivity_options`. For more information see section 8.18.1.2.

7.5.3 Weights

The target value of the norm of the interpolation error is set in the `InterpolationErrorBound` field under the `.../adaptivity_options` for a particular field. Usually, this will be constant throughout space and time, but advanced users may wish to vary this, and so it is possible to set this value as a Python field. When configuring an adaptive simulation for the first time, the general advice would be to start with a high weight and thus a high interpolation error (10% of the range of the field is a good rule of thumb), and reduce the weight as necessary to represent the desired dynamics. In all cases, the field weights should be the main parameters to vary to control the resolution of the adapted meshes.

7.5.4 Gradation parameter

In numerical simulations, a smooth transition from small elements to large elements is generally important for mesh quality. A mesh sizing function derived from error considerations may yield sudden changes in desired mesh edge length, due to the nature of the problem being resolved. Such sudden changes are undesirable in a mesh: for example, sudden changes in mesh sizing can cause

the spurious reflection of waves Bažant [1978], Bangerth and Rannacher [2001]. Therefore, a mesh gradation algorithm is applied to smooth out sudden variations in the mesh sizing function.

Various mesh gradation algorithms have been introduced to solve this problem. Löhner [1996] uses various functions of distance to point sources where edge length is specified by the user to control the isotropic sizing function for an advancing front grid generator. Owen and Saigal [2000] applies natural neighbour interpolation to smooth sudden variations in an isotropic sizing function. Persson [2006] bounds the gradient of an isotropic sizing function by solving a partial differential equation. Borouchaki et al. [1998] introduced two gradation algorithms for scalar isotropic mesh sizing functions, bounding the gradient of the sizing function or the ratio of the length of two adjacent edges, along with anisotropic generalisations of these. These algorithms have been successfully applied in many diverse application areas (e.g. Frey [2004], Alauzet et al. [2003], Laug and Borouchaki [2002], Lee [2003]).

Fluidity uses an algorithm based on that of Borouchaki et al. [1998]. It has only one user-configurable parameter, `/mesh_adaptivity/hr_adaptivity/enable_gradation/gradation_parameter`. This number constrains the rate of growth in desired edge lengths along an edge. A value of 1 would force the mesh to have constant edge length everywhere. A value of 2 would allow desired edge lengths to double from node to node. The default value is 1.5. Optionally, the algorithm may be disabled by choosing `/mesh_adaptivity/hr_adaptivity/disable_gradation` instead of `/mesh_adaptivity/hr_adaptivity/enable_gradation`.

7.5.5 Maximum and minimum edge length tensors

For robustness of the mesh adaptivity procedure, and to limit refinement/coarsening of the mesh it is possible to set maximum and minimum allowed edge length sizes. The input to these quantities are tensors allowing one to impose different limits in different directions. Assuming that these directions are aligned with the coordinate axes allows one to define diagonal tensors.

There are both good and bad reasons that one may need to impose these constraints. The good reasons are based on physics: a maximum size may be based on the size of the domain, or that it resolves a spatial scale that allows an instability to develop (the latter case would be better handled with a more advanced *a posteriori* error measure of course), or simply a consequence of the time/memory constraints of the machine the problem is running on. The bad reason is to control the mesh when the field weights have been chosen poorly. However, it is often unavoidable that the weights and max/min edge length sizes will be chosen in tandem to achieve an appropriate mesh, especially for experienced users — new users should be wary whenever maximum and particularly minimum size constraints are actually hit in the mesh and as a first stage should look to vary the weights to achieve the mesh they desire.

Finally, note that these constraints are achieved through manipulations to the metric, which in turn controls an optimisation procedure. They are therefore not hard constraints and one may observe the constraints being broken (slightly) in places.

7.5.6 Maximum and minimum numbers of nodes

Similar to the edge length size constraint above, it is possible to limit the maximum and minimum number of nodes that the mesh optimisation procedure returns. For reasons very similar to above this is potentially dangerous, but somewhat necessary.

This is effected by computing the expected number of nodes from the given metric. If the expected number of nodes is greater than the maximum number of nodes, the metric resolution is homogeneously increased so that the expected number of nodes is the maximum number of nodes. Similarly, if a minimum limit on the number of nodes is employed, the metric resolution is homogeneously

decreased if the metric would yield a mesh with fewer nodes.

For new users, altering the weights should be the primary way to control the size of the adapted mesh.

7.5.7 Metric advection

Metric advection is a technique that uses the current flow velocity to advect the metric forward in time over the period until the next mesh adapt. This allows an estimate of the mesh resolution required at each time-step before the next adapt to be obtained and incorporated into the adapted mesh [Wilson, 2009]. Each component of the metric is advected as a scalar using a control volume scheme with the time-step determined by a specified Courant number section 3.2.4.1.

With metric advection, mesh resolution is ‘pushed ahead’ of the flow such that, between mesh adapts, the dynamics of interest are less likely to propagate out of the region of higher resolution. This leads to a larger area that requires refinement and, therefore, an increase in the number of nodes. However, metric advection can allow the frequency of adapt to be reduced whilst maintaining a good representation of the dynamics

In the lock-exchange example, section 10.3, consider an adapted mesh with high resolution in the region of the interface between the two fluids. As the simulation proceeds, the gravity current fronts may propagate out of this high resolution region before the next adapt. This can lead to a more diffuse interface due to increased numerical diffusion from the advection method at coarser resolutions. Metric advection would increase the resolution in the adapted mesh not only in the interface, but also ahead of it, in the region into which the gravity current fronts will propagate.

The options for metric advection can be found under `/mesh_adaptivity/hr_adaptivity/metric_advection`, section 8.18.2.3.

7.6 Interpolation

As mentioned previously, the application of adaptive remeshing divides naturally into three sub-problems. The first is to decide what mesh is desired; the second is to actually generate that mesh. The third, discussed here, is how to interpolate any necessary data from the previous mesh to the adapted one.

This problem has received less attention from the adaptive remeshing community, with consistent interpolation (interpolation by basis function evaluation) almost universally used. Many papers in the adaptive remeshing literature do not even mention its use.

There are several good reasons for this. The drawbacks of consistent interpolation can be summarised as having suboptimal interpolation error, its unsuitability for discontinuous fields, and its lack of conservation. Firstly, for stationary problems, the interpolated solution is only used as an initial guess for the next solve, so any errors introduced in the interpolation have a minimal effect. Secondly, even for transient simulations, the interpolation error introduced is often acceptably low, provided the adapted mesh is suitable for the representation of the data. Thirdly, its unsuitability for discontinuous solutions and its loss of conservation are unimportant for the majority of applications of adaptive remeshing.

Nevertheless, there are good reasons to consider the mesh-to-mesh interpolation problem. Firstly, computing the interpolation with optimal accuracy in the L_2 norm is an interesting mathematical question in its own right. Secondly, consistent interpolation is unsuited to discontinuous Galerkin methods, which are increasingly popular. For these cases, consistent interpolation is not defined, and the averaging inherent in the pseudo-interpolation operators is diffusive and cannot exploit

discontinuous functions in the target function space. Thirdly, consistent interpolation is inherently nonconservative, which is a key requirement for the discretisation of certain problems. Without a conservative interpolation operator available, adaptive remeshing cannot be applied to these problems.

The standard method, consistent interpolation, consists of evaluating the previous solution at the locations of the nodes in the adapted mesh, and taking these values as the coefficients of the associated shape functions. As basis function evaluation is trivially available for any finite element method, the only difficulty is the problem of mesh association: the identification of which basis functions to evaluate for a given node in the adapted mesh, i.e. to identify in which element of the previous mesh each node of the adapted mesh lies. The relevant element is referred to as the parent element of the node.

[Peraire et al. \[1993\]](#) discuss interpolation between meshes in the context of non-nested multigrid methods. The authors observe that Galerkin projection is optimal in the L_2 norm, note that its assembly necessitates computing the inner products of the basis functions of both meshes, and comment that this computation is very difficult because the basis functions are defined on different supports. No mention of mesh intersection is made; however, the authors demonstrate that if the inner products are approximated with numerical quadrature on the donor mesh, the resulting approximate projection is still conservative. Despite this conservation property, the use of this procedure to compute the inner products is discouraged as it is very inaccurate.

[Löhner \[1995\]](#) discusses the mesh association problem in detail. The author discusses brute-force searching, methods of subdividing space, and develops an advancing-front vicinity searching algorithm. The algorithm exploits the connectivity of the target and donor meshes. Since adjacent nodes in the target will lie in nearby elements in the donor mesh, the algorithm uses the parenthood information for nodes which have already been interpolated to provide clues for the search for the parent of unprocessed nodes.

[George and Borouchaki \[1998\]](#) discuss the necessity of solution interpolation after adaptive remeshing, note the non-conservative character of consistent interpolation, and propose the use of the Galerkin projection from mesh to mesh by means of mesh intersection. Galerkin projection is the optimally accurate projection in the L_2 norm, and is conservative, but its implementation is very difficult. The fundamental reason for this difficulty is that the method requires the computation of the inner products of the basis functions of the two meshes. In order to compute these exactly, the supermesh of the two meshes must be constructed, which is quite involved. Although they comment that in their experience this provides a satisfactory algorithm for solution transfer, they give no examples. The reader is referred to a technical report by R. Ouachtaoui to be published in 1997 for further discussion; it appears, however, that this technical report was never published. [Geuzaine et al. \[1999\]](#) also discuss the Galerkin projection between two-dimensional meshes; however, rather than integrating over the supermesh, the integrals appear to be computed over the target mesh. This is less accurate than assembling over the supermesh, and therefore should be referred to as an approximate Galerkin projection. A similar approach is taken by [Parent et al. \[2008\]](#).

[Farrell et al. \[2009\]](#) was the first to present the application of supermeshing to adaptive remeshing, and the first to describe a bounded variant of the Galerkin projection. The supermeshing algorithm was drastically improved in [Farrell and Maddison \[2011\]](#), and this paper describes the Galerkin projection algorithm used in Fluidity.

In summary, the choice should be consistent interpolation, unless any of the following conditions hold:

- The simulation has a discontinuous prognostic field which must be interpolated.
- Conservation of some field is crucial for the dynamics.

In such cases, Galerkin projection should be applied. If both conservation and boundedness are

desired, a bounded variant of the Galerkin projection algorithm is available [Farrell et al., 2009], but this is only implemented for P_1 fields.

The default choice is consistent interpolation, as specified by `.../consistent_interpolation`. If Galerkin projection is to be used, change this to `.../galerkin_projection`. Under `galerkin_projection`, choose either `continuous` or `discontinuous`, depending on whether the field exists on a continuous or discontinuous mesh respectively. (In the continuous case, Galerkin projection involves the solution of a linear system involving the target mesh mass matrix, which is why additional options under `.../galerkin_projection/continuous/solver` are required to configure how this linear system is solved.) If Dirichlet conditions should be enforced through the Galerkin projection procedure, set the `.../galerkin_projection/honour_strong_boundary_conditions` option.

If the bounded variant of Galerkin projection is desired, activate the `.../galerkin_projection/continuous/bounded::Diffuse` option. The algorithm to bound the Galerkin projection is iterative, and the user must set a limit on the number of iterations it will perform with `boundedness_iterations`. If the bounds are known *a priori*, then these may be specified in the `bounds` option; otherwise, the bounds are derived from the field values before interpolation.

7.7 Parallel adaptivity

The approach taken to adaptivity in parallel is as follows:

- Each process adapts their local mesh, excluding halo elements.
- The mesh is re-partitioned with high edge-weighting applied to those elements below the element quality cutoff.
- Repeat the two steps above up to `adapt_iterations` (default value of 3).
- Finally re-partition without applying edge-weighting to return a load balanced mesh.

Zoltan is a parallel partitioning and data distribution library [Devine et al., 2002]. It is used within Fluidity to do the mesh re-partitioning during the parallel adaptivity. Zoltan gives access to various different partitioning libraries; ParMETIS, PT-Scotch as well as its own graph and hypergraph partitioners. For the intermediate adapt iterations (when edge-weighting is applied) the partitioner to be used by default is Zoltan graph and this can be changed using the `partitioner` option in Diamond. For the final adapt iteration where load balance is our only concern the default partitioner is ParMETIS and this can be changed using the `final_partitioner` option.

The high edge-weighting being applied to poor quality elements aims to prevent those elements being on a partition boundary after the re-partitioning. This is to prevent them being locked for the next adapt iteration. Zoltan gives priority to producing load balanced partitions so for these intermediate adapts we loosen the Zoltan parameter, load imbalance tolerance, to allow Zoltan to produce load imbalanced partitions but take into account the edge-weighting we are applying. The default load imbalance tolerance is 1.5 but this can be changed in the Zoltan options.

More information on the Zoltan options can be found in section [8.18.2.2](#)

7.8 The cost of adaptivity

The operations involved with adapting the mesh (forming the metric, remeshing, interpolation) have an associated computational cost. Whilst, in most cases, this overhead will be small, it should be

noted. In general, the cost associated with formation of the absolute, relative and p -metrics is the same as the time is spent in formation of the Hessian, which is required for all three. Metric–advection will increase the time spent in metric formation due to the additional solutions of the advection equation required. The cost will depend on the chosen CFL number. Finally, consistent interpolation is faster than Galerkin projection (due to the formation of the supermesh). Bounded Galerkin projection will also require more time than unbounded Galerkin projection due to the bounding procedure.

The percentage of time spent in the adaptivity routines for simulations of the lock–exchange are given in table 7.1, section 10.3. These values should be taken as a guide rather than a definitive value for all simulations. For example, the number of iterations used in the linear solve of the bounding procedure for Galerkin projection (and hence cost) will depend not only on the interpolation step but on how well bounded the solution is to begin with which in turn depends on the choice of discretisation method.

Adaptivity options				% of total simulation time spent in adaptivity routines		
Metric	Adapt frequency	Interpolation method	Metric advection	Metric formation	Remesh and interpolation	Total
M_∞	10	CI	no	4.0	3.5	7.5
M_∞	10	CI	yes – 5	25.7	2.2	27.9
M_∞	10	CI	yes – 10	18.5	2.6	21.1
M_∞	40	CI	yes – 5	29.5	0.7	30.1
M_∞	40	CI	yes – 10	17.8	0.9	18.7
M_∞	10	bd GP	no	3.5	14.0	17.5
M_2	10	CI	no	4.1	3.4	7.5

Table 7.1: Percentage of time spent in adaptivity routines for adaptive mesh simulations of the lock–exchange, section 10.3 [Hiester et al., 2011]. The simulations were run from $t = 0$ s until $t = 25.025$ s, the time the free–slip head nears the end wall. This is equivalent to 1001 time–steps and 100 adapts when adapting every 10 time–steps and 25 adapts when adapting every 40 time–steps. The adapt frequency is given in number of time steps. For the interpolation method CI: consistent interpolation and bd GP: bounded Galerkin projection. For those simulations that use metric advection the number given is the CFL number used to determine the time step used in the calculation of advection of the metric components.

Chapter 8

Configuring Fluidity

8.1 Overview

A Fluidity simulation is configured by creating a Fluidity options (or flml) file using Diamond, which is the Graphical User Interface (GUI). The left-hand pane of Diamond allows users to browse the *options tree*, while the right-hand pane provides brief documentation about the option and is where users enter input data.

This chapter aims to provide a detailed description of all the options in the tree. From section 8.3 onwards, Fluidity options are described in the order in which they appear in Diamond. Prior to this are some important general notes about the different types of options and, in particular, how to work with fields in Fluidity.

8.2 Options syntax

Fluidity options files, or flml files, are XML files whose grammar is defined by the fluidity options schema `fluidity_options.rng`. XML files have a tree-like structure of elements containing other elements. This structure is reflected in the left hand pane of Diamond, the GUI which is used to write flml files.

The flml file can also be edited with a standard editor and the options written in text and in code using the Spud library on which the Fluidity options system is based. A location in the options tree can be written as a path, much like the path of a file on disk. So, for example, the option controlling the physical dimension of the problem domain is written as `/geometry/dimension`. This should be read as the `dimension` option which is found under `geometry` which is in turn at the top level of the options tree. In Diamond, and in the flml file, the absolute top level element is always `fluidity_options` but this element is always discarded from paths. Figure 8.1 shows a Diamond screen shot open at the `/geometry/dimension` option. Further documentation of the Spud system is available in Ham et al. [2009] and from the Spud website.

8.2.1 Allowed Characters

Only certain characters are recognised by the options dictionary, which contains the flml input once it is read into fluidity. Therefore only the following letters are allowed in any input field:

```
/_:[ ]1234567890qwertyuioplkjhgfdaszxcvbnmMNBVCXZASDFGHJKLPOIUYTREWQ
```

Comment boxes may contain any characters.

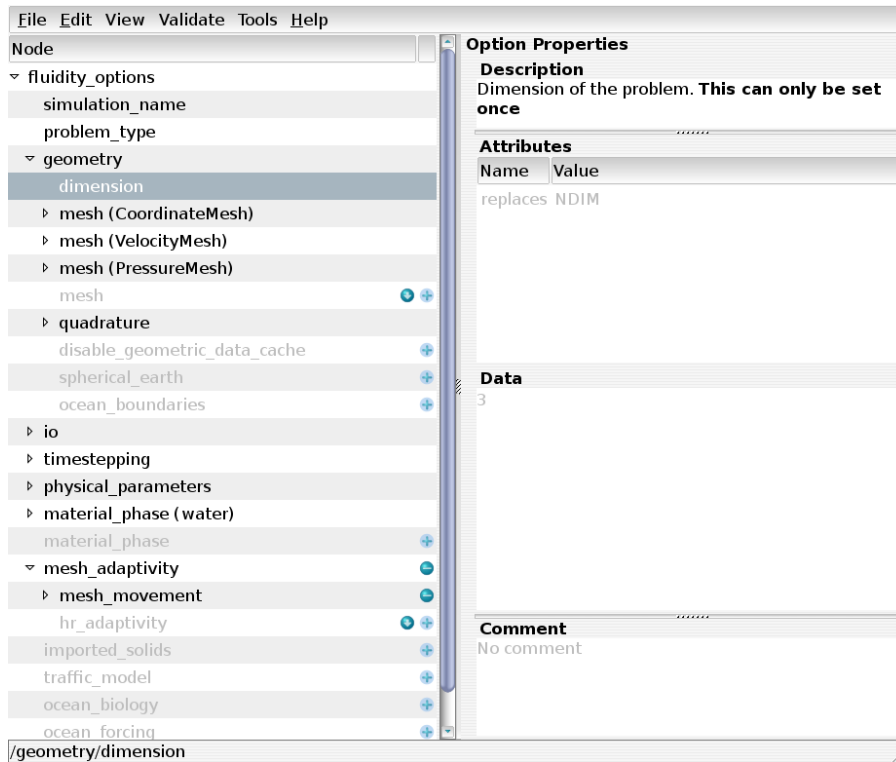


Figure 8.1: A Diamond screenshot showing the `/geometry/dimension` option. Note that the option path is displayed at the bottom of the diamond window

8.2.2 Named options

Some options in the tree, such as fields and meshes, have name attributes. The name attribute is represented in the flml file with a double colon so that, for example, the coordinate mesh has options path `/geometry/mesh::CoordinateMesh`. Note that this differs from the convention in the Diamond interface in which name attributes are given in brackets. Figure 8.2

Names of objects (fields, material phases, meshes, etc.) should be camel cased (MyOwnField) and not contain spaces or underscores. Furthermore, the characters `/` `:` `[` `]` are prohibited as these have special meanings in the options dictionary inside Fluidity.

8.3 The options tree

The top level of the options tree contains the following compulsory elements:

- Simulation Name
- Problem Type
- Geometry
- IO
- Timestepping
- Physical Parameters
- Material/Phase

The first six of these are described here.

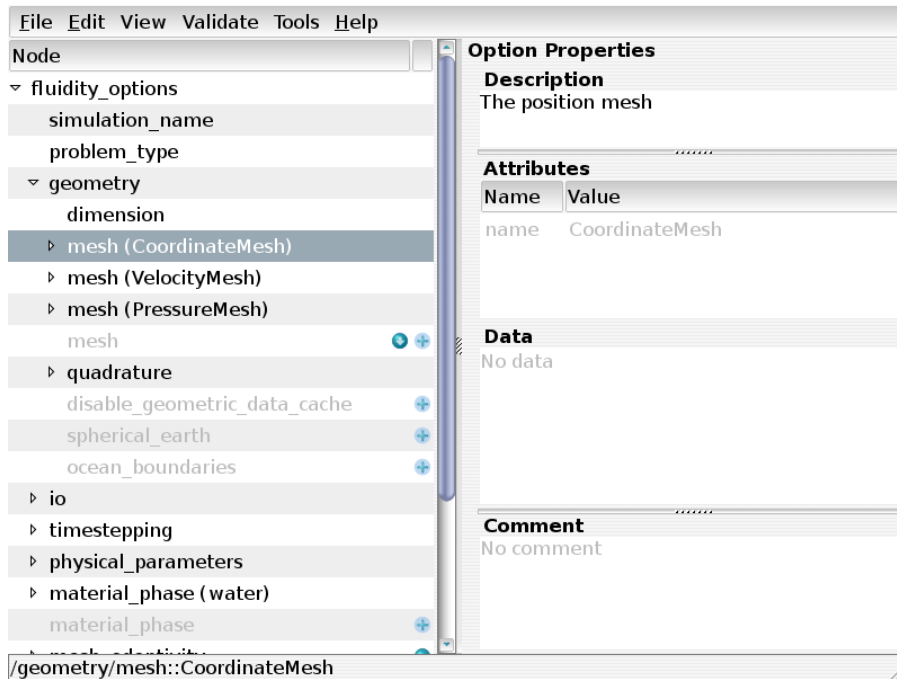


Figure 8.2: A Diamond screenshot showing the `/geometry/mesh::Coordinate` option. Note that the name is shown in brackets in the main Diamond window but after double colons in the path in the bottom bar.

8.3.1 Simulation Name

The simulation name is the base name for all output files. For example if you set the simulation name to foo then your statistics output file will be called foo.stat

8.3.2 Problem Type

Setting problem type gives fluidity a hint as to what sort of simulation you are conducting and therefore what combinations of options are likely to be valid. If you do not know which category applies to your problem, choose "fluids".

8.3.3 Geometry

This element contains all the options required to specify the geometry of the mesh and the accuracy of the finite element discretisation.

8.3.3.1 Dimension

The dimension of the domain of your problem. This can be 1, 2 or 3. Be careful, once you set the dimension you can't change it again! This is necessary to ensure that all vectors and tensors are of the correct dimension.

8.3.3.2 Meshes

Meshes are the finite element spaces on which your problem is solved. Meshes are either read from file or are derived from other meshes. Mesh options are described in detail in section 8.4. There is only

one required mesh: the `CoordinateMesh`. Some settings or fields have specific mesh requirements. These are discussed under the appropriate options.

8.3.3.3 Quadrature

Fluidity uses numerical quadrature to integrate the equations over each element. There is a performance/accuracy trade off in quadrature: the more quadrature points are employed, the more accurate the integrals are but the more expensive the assembly operation is. Quadrature rules in Fluidity are categorised by the degree of the polynomial which they will integrate exactly. The higher the degree of the quadrature rule, the more quadrature points will be employed. As a general rule of thumb, the quadrature degree employed should be at least $\max(2n_{\mathbf{u}+1}, 2n_p)$ where $n_{\mathbf{u}}$ is the degree of the elements employed for velocity and n_p is the degree of the elements employed for pressure. This means that degree 4 quadrature is sufficient for most of the fluidity configurations currently in use.

The quadrature degree is specified by setting `/geometry/quadrature/degree`.

8.3.3.4 Spherical Earth

Enabling `/geometry/spherical_earth` informs Fluidity that your simulation is being carried out in an Earth like geometry, that is, a three dimensional geometry with gravity pointing towards the centre of the coordinate system. This has implications for various options and terms such as wind forcing (see section 8.12.3.9), the calculation of buoyancy and the 'direction' of absorptions (see 8.7.3).

If this option is checked, wind forcing and *e.g.*, momentum forcing from bulk formulae (see 8.12.3.3) will automatically be rotated and applied in the direction tangential to the Earth's surface. It will also result in absorption terms set through the options tree being rotated and applied in the longitudinal, latitudinal and radial directions respectively. Additionally, in many terms such as the buoyancy density, the direction of gravity is hard coded and calculated at explicitly at Gauss points when this option is enabled. Thus, if enabling this option, the direction of gravity specified in the options tree (see 8.3.6.1) must be set via a python function representing the inward normal to the sphere. Note however that the viscosity and diffusion operators are currently not rotated automatically and thus the user must carry such rotations out themselves. An example in which the viscosity is rotated is given in section 10.11.

Under `/geometry/spherical_earth` the user must select `linear_mapping` or `superparametric_mapping`. The former results in cords between nodes being approximated as linear segments whilst the latter gives a better approximation to the Earth's shape through approximating cords with a higher order polynomial. The order of the polynomial is given by the degree of the mesh on which gravity is located.

8.3.3.5 Ocean Boundaries

These options are required if you are running an ocean simulation with a free surface or with various other ocean options which require the code to know where the ocean surface and bed lie. `/geometry/ocean_boundaries/top_surface_ids` and `/geometry/ocean_boundaries/bottom_surface_ids` are lists of boundary tags from your input mesh which lie on the ocean surface and bed respectively.

It is not usually necessary to change the settings for either of the scalar fields under this option.

8.3.4 IO

These options control the frequency and form of model outputs.

8.3.4.1 Dump format

The file format used to output fields to disk. At this stage, vtu is the only allowed format.

8.3.4.2 Dump period

This is the interval between the state fields being output to disk. You should usually start by setting this to a rather low value (possibly as short as your timestep) for testing and then increase it for production runs once you know how your configuration works. The value can be specified as either a constant or python function.

It is possible to swap `/io/dump_period` for `/io/dump_period.in.timesteps` to specify that you wish to have a dump every fixed number of timesteps.

8.3.4.3 Output mesh

All fields will be interpolated onto the same mesh for output. Usually the `CoordinateMesh` is the right choice. If you have fields that are of higher order than the selected output mesh you will lose accuracy. Interpolating all fields to a higher order mesh for output may give very large dump files however. If any of the fields in the output is discontinuous, the mesh in the output file will be a discontinuous version of the mesh selected here.

8.3.4.4 Disable dump at start

A dump is normally performed at the start of the simulation. This options disables that.

8.3.4.5 Disable dump at end

A dump is normally performed at the end of the simulation. This options disables that.

8.3.4.6 CPU dump period

This outputs dumps at specified CPU times. Not recommended.

8.3.4.7 Wall time dump period

Outputs at specified walltime (real time) periods. Not recommended.

8.3.4.8 Max dump file count

Limits the number of dumps by overwriting the previous dumps after the number specified here.

8.3.4.9 Convergence

You can check certain fields for convergence during nonlinear iterations. To do this switch on `/timestepping/nonlinear.iterations` and `/timestepping/nonlinear.iterations/tolerance` and switch on the convergence option under the fields that you want to check for convergence.

It is possible to enable the creation of a convergence file, giving details of the convergence of each field over the global nonlinear iteration loop. The `.convergence` file is in the same format as the `.stat` file. In order to do this, switch on `/io/convergence` and `/io/convergence/convergence_file`. You still need the options in the above paragraph.

8.3.4.10 Checkpointing

Enables checkpointing, which saves sufficient information (including a new flml options file) to restart a simulation, i.e., to continue the simulation after it stopped. You must specify how often to checkpoint in terms of the number of dumps. There are also options to checkpoint at the start of the simulation and at the end. This latter is useful when running on batch systems that have a time limit.

Up to five sets of files are created when checkpointing:

1. Mesh files - The `from_file` meshes, in triangle format. Surface IDs and (if present) region IDs are written to the mesh files, and adaptivity is supported. In parallel a triangle mesh is written for each process.
2. Halo files (in parallel) - Halo information for each process.
3. Field files - A `vtu` is written for each mesh with prognostic fields in each state and (in parallel) for each process.
4. Checkpointed option file - A new FLML file, with the `from_file` mesh set to read the checkpoint mesh files and the prognostic fields set to initialise from the checkpoint field files.
5. Checkpointed detector files - Two files related to checkpointing of detectors are created.

The first checkpointed detector file has the extension `.groups` and contains a header with the names of the groups of detectors in the order they were read in the simulation. It also contains information about the number of detectors in each group. This is to guarantee that when restarting from a checkpoint, the detectors will be read in the same order and consequently will be saved in that same order into the output detector file for consistency. The second checkpointed detector file has the extension `.positions.dat` and contains the last position of the detectors at the time of checkpointing, in binary format.

No checkpointed detector files will be created if only static detectors are defined in Diamond, since the position of these detectors remains always the same. This is mainly used when Lagrangian detectors are set that are advected by the flow and hence, their position changes as the simulation proceeds.

At the same time as creating the two files related to checkpointing of detectors, the detector options in the options tree or Diamond are updated so that in the new flml file the detectors are set to initialise from the checkpoint detectors files

```
(/io/detectors/detector_array/from_checkpoint_file or
/io/detectors/lagrangian_detector/from_checkpoint_file). For simplicity, the static
detectors are also read from the checkpoint file that contains the position of all detectors, static and
Lagrangian (/io/detectors/static_detector/from_checkpoint_file).
```

Checkpoint filenames all end with [dump no.]_checkpoint[-[process]].[extension], where the process number is added to the mesh, halo and field files in parallel. The checkpoint detectors filenames contain _det after [dump no.]_checkpoint.

A script is available at `scripts/rename_checkpoint.py` that can be used to easily rename these filenames and there contents to continue the naming convention of the original run. For more information see section 9.3.3.28.

To restart from a checkpoint, specify the checkpointed FLML file as input.

8.3.4.11 Stat

Contains additional options for handling stat files, for example outputting a stat file at the start (timestep zero) and outputting stat data before and after an adapt.

There are further options under individual fields, for example to exclude data from the stat file.

8.3.4.12 Detectors

Detectors are set in Diamond with the `/io/detectors` option. The detectors can be set to be static detectors, `/io/detectors/static_detector`, Lagrangian detectors `/io/detectors/lagrangian_detector` or an array of detectors, `/io/detectors/detector_array`.

When choosing to set detectors using an array, the total number of detectors needs to be specified in `/io/detectors/detector_array/number_of_detectors` and the type of detectors is indicated with the option `/io/detectors/detector_array/static` or `/io/detectors/detector_array/lagrangian`.

Examples 8.1 and 8.2 illustrate the use of a Python function to set an array of detectors, that can be static or Lagrangian.

```
def val(t):
    import math

    ret=[]
    for i in range(100):
        ret.append([-2.5, (-0.495 + i * 0.01)])

    return ret
```

Example 8.1: A Python function setting 100 detectors. This example illustrates that it is possible to use a Python function to set an array of detectors.

If one or more Lagrangian detectors are selected the option `/lagrangian_timestepping` must be set to define detector movement. The user can define the order of the Runge-Kutta method to be used by defining the Butcher tableau and timestepping weights under `/explicit_runge_kutta_guided_search`. For convenience the first- and fourth-order Runge-Kutta method (`/forward_euler_guided_search` and `/rk4_guided_search`) are available as pre-defined options. See section 3.12 for more information on Lagrangian detector advection.

The output of the detectors is an ascii file called `name_of_simulation.detectors` where the name of the simulation has been indicated in `/simulation_name`. If binary output `/io/detectors/binary_output` option is enabled then the file containing the detector data is called

```

def val(t):
    import math

    ret=[]
    for k in range(100,2000,100):
        for j in range(7000,25100,100):
            for i in range(7000,25100,100):
                ret.append([i,j,k])

    return ret

```

Example 8.2: A Python function setting 622459 detectors uniformly distributed at intervals of 100 m in the three orthogonal directions. They cover 19 z planes, from z=100 to z=1900, with 32761 detectors in each plane, from x=7000 to x=25000 and y=7000 to y=25000.

`name_of_simulation.detectors.dat` and in this case `name_of_simulation.detectors` contains only the header with the information about the detectors (name of each detector, in which column the position of each detector is stored, etc.).

Note that the algorithm used to determine the element containing a detector (of any kind) assumes that the detector is known to be within the simulation domain. The option `/fail_outside_domain` will cause Fluidity to fail if a detector is found to be outside the domain boundaries. This option should be the default choice when creating new simulations. If detectors are intended to temporarily reside outside of the domain the option `/write_nan_outside_domain` will cause Fluidity to write "NaN" values to the detector output in this case.

If `/move_with_mesh` is selected with any mesh movement algorithm the detectors will move according to the mesh displacement. That is to say that a static detector will remain static with reference to the domain boundaries rather than its true physical coordinates.

8.3.4.13 Log output

Enables additional output to the screen or log file. Usually controlled using the `-v` option when running Fluidity. However, a useful option for logging memory diagnostics can be switched on here.

8.3.5 Timestepping

These options control the start and end time of the simulation as well as options regarding timestep size.

8.3.5.1 Current time

This is the model time at the start of the simulation. In most cases this is likely to be zero. It can be non-zero when continuing a simulation from a checkpoint.

Time units

If your simulation contains real data, for example when using `ocean_forcing`, Fluidity must know how to map simulated time onto real time. This option allows the user to specify the "real-world" start time of the simulation. The input is a string of the form:

```
seconds since 1992-10-8 15:15:42.5 -6:00
```


which indicates seconds since October 8th, 1992 at 3 hours, 15 minutes and 42.5 seconds in the afternoon in the time zone which is six hours to the west of Coordinated Universal Time (i.e. Mountain Daylight Time). The time zone specification can also be written without a colon using one or two-digits (indicating hours) or three or four digits (indicating hours and minutes).

8.3.5.2 Timestep

The simulation timestep. If adaptive timestepping is not used this will define the size of the timestep used throughout the simulation. If adaptive timestepping is used this option defines only the size of the first timestep.

8.3.5.3 Finish time

The model time at which the simulation should halt. Note that the simulation may overrun slightly due to roundoff in calculating the current time or if the timestep does not divide the simulation time exactly.

8.3.5.4 Final timestep

Rather than specify a finish time, the final timestep may be specified. This is the number of timestep after which the simulation will stop.

8.3.5.5 CPU time limit

This option will stop the simulation after the CPU time reaches this limit. This option is useful when coupled with `/io/checkpointing/checkpoint_at_end` enabled.

8.3.5.6 Wall time limit

This option will stop the simulation after the wall time (real time) reaches this limit. This option is useful when coupled with `/io/checkpointing/checkpoint_at_end` enabled.

8.3.5.7 Nonlinear iterations

Nonlinear quantities in the equations are represented by their last known values. It may be necessary to solve the equations more than once to produce better approximations to those last known values for reasons of accuracy or stability. Unless there are reasons for doing this, set this value to 2.

8.3.5.8 Adaptive timestep

This option allows the timestep, ΔT , to vary throughout the run, depending on the Courant-Friedrichs-Lewy (CFL) number. There are several sub-options here. The `.../requested_cfl` is the desired upper limit of the CFL. A value of 5-10 is usual here. Fluidity will increase the timestep if the CFL number is less than this value and decrease it if the CFL is greater than this. The `.../minimum_timestep` and `.../maximum_timestep` options limit the timestep. The option `.../increase_tolerance` determines the rate of growth in the timestep. A value of 1.5 indicates the timestep can grow by at most, 50%. There is no limit on the rate of decrease. Note that if a timestep

fails to meet the CFL limit imposed it is not re-run, but the timestep is decreased for the next iteration. Finally, a desired ΔT can be calculated at the first time iteration by switching on the option: `.../at_first_timestep`

8.3.5.9 Steady state

It is possible to run Fluidity until it converges to a steady state; this is sometimes useful for initialising a problem. In order to do this, switch on `/timestepping/steady_state` and set a tolerance.

8.3.6 Physical parameters

These options control global physical quantities.

8.3.6.1 Gravity

The importance of buoyancy is discussed in section 2.4.3. This requires a gravitational field to be set and involves both its magnitude (e.g., 9.8 m s^{-2}) and a vector field specifying the direction in which gravity points. For a 3D simulation in a flat domain with gravity pointing in the negative z direction you would set `value(WholeMesh)` for this field to the vector (0.0, 0.0, -1.0). For a gravitational force with spatially varying direction, e.g. on the Earth considered in Cartesian space with gravity pointing in the negative radial direction you could use a Python function of the form

```
def val(X, t):
    from math import sqrt
    radius=sqrt(X[0]**2+X[1]**2+X[2]**2)
    rx=X[0]/radius
    ry=X[1]/radius
    rz=X[2]/radius
    return (-rx, -ry, -rz)
```

Example 8.3: A Python function returning a vector pointing in the negative radial direction.

8.3.6.2 Coriolis

Fluidity supports the specification of the Coriolis term (section 2.4.1) in a number of different ways. The following options are available:

1. `f_plane` – a single float is prescribed which corresponds to f_0 in (2.37);
2. `beta_plane` – here two floats are prescribed, f_0 and β in (2.38);
3. `sine_of_latitude` – here the Coriolis parameter from (2.35) is used and Ω , R_{earth} and latitude_0 are defined as floats with latitude calculated via $\varphi = y/R_{\text{earth}} + \text{latitude}_0$;
4. `on_sphere` – here Ω the rotation vector pointing in the inertial frame z direction (2.34) is set, note this is the direction pointing from the centre of mass to the North Pole on the Earth;
5. `python_f_plane` – time dependent python input prescribing a single float which corresponds to f_0 in (2.37) - see example 8.4.

Recall that there is a factor 2 relationship between f and Ω (2.36) — make sure you don't get caught out by this.

```

if t < 4000.0:
    omega = 3.0
elif t < 6000.0:
    omega = 2.5
elif t < 8000.0:
    omega = 2.0
elif t < 10000.0:
    omega = 1.5
elif t < 12000.0:
    omega = 1.0
elif t < 14000.0:
    omega = 0.5
else:
    omega = 0.0

return 2.0 * omega

```

Example 8.4: `python_f_plane` definition, sweeping through a number of rotation rates. Note the factor of 2 between f and Ω (see equation (2.36)).

8.4 Meshes

A mesh defines the discrete function space in which the values of one or more fields lie. For example, the mesh defines what degree of polynomials are employed on each element, whether the field is continuous or discontinuous between elements, and whether the domain is periodic in any direction.

Meshes are defined in the `fml` file by `/geometry/mesh` options. The mesh associated with each field is referred to by name in the `.../mesh` option under that field.

8.4.1 Reading meshes from file

There must always be one mesh which is read in from a set of files in triangle format. This is usually the `CoordinateMesh`. To specify the triangle files from which the coordinate mesh should be read, set the `file_name` attribute of `/geometry/dimension/mesh::CoordinateMesh/from_file` to the basename of the triangle files (that is, the filename without `.node`, `.ele`, etc.)

The coordinate mesh read in from file will always have linear elements and the `Coordinate` field is always continuous between elements.

Fluidity also has native Gmsh support, which loads in Gmsh files directly into Fluidity, and works with binary and ASCII Gmsh formats. To enable native support, Fluidity needs to be told to expect a Gmsh file, which is achieved by setting the `/geometry/mesh/from_file/format` option to `gmsh`. Fluidity will now look for a file with the extension `.msh` when it runs.

For information on generating meshes in gmsh and triangle format, see chapter 6.

8.4.2 Deriving meshes from other meshes

The alternative to reading a mesh from a file is to derive it from another mesh. This is necessary when for instance we wish to derive a mesh with different continuity or elements than the original mesh. For example, if we have a `CoordinateMesh` as our input mesh read from file, it is possible to derive a `VelocityMesh` from it by adding `/geometry/mesh::VelocityMesh`, selecting `from_mesh` under it and there selecting `mesh::CoordinateMesh`. If nothing further is specified under the new mesh, the derived mesh will be exactly the same as the mesh it is derived from.

The more interesting case occurs where we wish to derive a mesh with different continuity or elements from the original mesh. To specify a discontinuous mesh, under `.../from_mesh` enable `mesh_continuity` and select `discontinuous`.

Similarly, to specify a mesh with higher polynomial degree elements, enable `mesh_shape` under `.../from_mesh` and set `polynomial_degree`.

Meshes with any name can be added. Only the name `CoordinateMesh` is special, as it will be used to store the coordinates of the mesh. This is also the only required mesh. `VelocityMesh` and `PressureMesh` are only provided as suggested names as quite often the pressure and velocity fields need to be on a different mesh than the coordinate mesh, e.g. for $P_{1DG}P_2$. It is however not required that the velocity is defined on a mesh with the name `VelocityMesh`, nor does the pressure field have to be on a mesh with the name `PressureMesh`. If for instance the mesh needed for pressure is the same as your `CoordinateMesh`, e.g. for P_1P_1 or P_0P_1 , the pressure can be defined directly on the `CoordinateMesh` and no extra mesh is needed.

8.4.2.1 Shape function

This option is used to specify the degree of polynomial which should be used for the shape functions on each element of the mesh. If not selected, the shape functions will be the same as those on the mesh from which this mesh is derived.

8.4.2.2 Continuity

This option can be set to `discontinuous` to derive a discontinuous mesh from a continuous one. Note that it is not possible to derive a continuous mesh from a discontinuous mesh.

8.4.2.3 Periodic

To specify a periodic domain in Diamond, add a new mesh under `/geometry/mesh`. Select `.../from_mesh/mesh::CoordinateMesh` and then turn on `.../from_mesh/periodic_boundary_conditions` for each dimension that is periodic. There are three fields that need to be completed: the surface IDs of one side, the surface IDs of the opposite side and a python function (`coordinate_map`) which contains the necessary mapping function.

For example, suppose that the domain is the unit square shown in figure 8.3 which is to be periodic in the x direction. We designate surface ID 1 as the *physical boundary* and surface ID 2 as the *aliased boundary*. We therefore enter 1 in `.../physical_boundary_ids` and 2 in `.../aliased_boundary_ids`. The *coordinate map* function takes a point on the *aliased* boundary to the corresponding point on the *physical* boundary. In this case, the appropriate function is:

```
def val(X,t):
    result = list(X)
    result[0]=result[0]-1.0
    return result
```

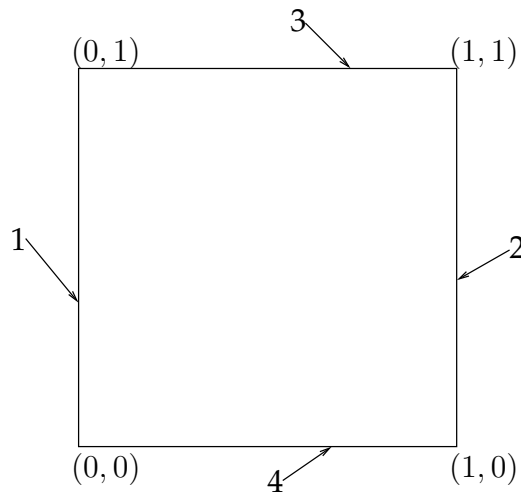


Figure 8.3: Periodic unit square with surface IDs 1-4 shown.

Meshes that are required to be periodic can now be derived from this periodic mesh. Note that the periodic mesh must be directly derived from the mesh which has been read `from_file`. It is not possible to derive a periodic mesh from a `from_mesh` mesh.

8.4.2.4 Extruded meshes

It can be advantageous to have a mesh in which all the nodes line up in vertical lines. To achieve this effect within Fluidity, it is possible to read in a mesh in $n - 1$ dimensions and extrude it along the n -th dimension. An extruded mesh is specified using the `.../from_mesh/extrude` option.

Under this option it is necessary to set the `regions/bottom_depth`. This is a scalar value which gives the depth, a positive value. The extent of the domain in n -th dimension will be $(0, -\text{bottom_depth})$. This value may be set either as a constant or as a Python function. In the latter case, function will be a function of space and time. Note in this case that the space argument `X` will be $n - 1$ -dimensional. See section 8.6.2.2 for a full explanation of the use of Python functions to prescribe field values. In this case, the depth is essentially a scalar field over the $n - 1$ dimensional parent mesh. The time argument which will be passed to the function is the simulation start time and the function will *not* be re-evaluated during the simulation.

The second option which must be set is the `.../sizing_function`. This specifies the mesh spacing along the n -th dimension. It may once again be a constant or a Python function. In the latter case, it will be a function of all n dimensions which facilitates the mesh spacing varying in depth as well as in the horizontal. Once again, the function will be evaluated only at simulation start.

It will usually be advantageous to specify the surface ID to be associated with the top and bottom boundaries, so that boundary conditions can be associated with them. This is achieved using the `.../top_surface_id` and `.../bottom_surface_id` options. The lateral boundaries of the extruded mesh will inherit the surface IDs associated with the the boundaries of the parent (non-extruded) mesh.

It is possible to specify different options for different regions of the mesh by adding multiple `.../extrude/regions` options and changing them to the generic rather than the `regions::WholeMesh` version. The new regions need to be named and the region IDs to which they apply specified.

As well as extruding 2D meshes (where only the x and y coordinates are specified in the triangle file), given a pseudo 2D mesh on a spherical shell, Fluidity can perform and extrusion in the radial direction. To perform such an extrusion simply enable the options as noted above and additionally check the `/geometry/spherical_earth` option. With this option enabled Fluidity will then

perform the specified extrusion towards the centre of the sphere.

Extrusions on the sphere can be performed such that the ‘depth’ of the extrusion conforms to bathymetric data. To extrude according to bathymetry Fluidity must be provided with a netCDF data file containing three columns of data giving the longitude, latitude and depth of each point respectively. The name and location of this data file must then be entered under `.../extrude/regions/bottom_depth/from_map`. To avoid the depth at coast dropping to zero the user may also enter a minimum depth under the `.../from_map/min_depth` option. Note however, that if a minimum depth is specified, this will be applied throughout the domain.

8.4.2.5 Extruded periodic meshes

If an extruded periodic mesh is required then the periodic mesh must first be derived from the `from_file` mesh. The extruded mesh is then derived from the periodic mesh. All other meshes are next derived from the extruded mesh. A special case is the `CoordinateMesh`. This must be derived from the extruded mesh by specifying `periodic_boundary_conditions/remove_periodicity`. At this stage it is necessary to re-specify the `physical_boundary_ids`, `aliased_boundary_ids` and `coordinate_map`. Additionally, the `inverse_coordinate_map` must be given. As the name suggests, this function must be the inverse of the original `coordinate_map`.

8.5 Material/Phase

The final compulsory element in the top level of the options tree is `/material_phase`. A `/material_phase` element groups all of the fields which pertain to one phase or one material. See section 8.19 for an explanation of the distinction between a phase and material in this context.

When configuring ocean problems (or single material/single phase fluids problems), only one `/material_phase` is required. Multi-material and multi-phase problems will require one `/material_phase` for each phase or material in the problem.

Note that you must give each of your material phases a name.

The following sections (8.7 to 8.15) describe the options below the `/material_phase` option.

8.6 Fields

8.6.1 Types of field

A field associates a value with every node in the domain. Examples of fields in a fluids simulation include the velocity and pressure. Fields in Fluidity are distinguished by the rank of the data on the field and the way in which that field is calculated.

Scalar fields have a scalar value at each node. Common examples include temperature, pressure and density.

Vector fields have a vector value, in other words a list of numbers, at each node. The rank of a vector field is 1 and the length of the vector is given by the dimension of the problem.

Tensor fields have a value given by a square matrix at each node. The side length of the matrix is the problem dimension and the rank is naturally 2. The diffusivity of a tracer is a typical example of a tensor-valued field.

Fields can also be characterised by the manner in which their value is calculated. Fluidity recognises three such categories:

Prognostic fields are the result of solving a partial differential equation. In a typical fluids simulation, the velocity and pressure are prognostic and are calculated by solving some variant of the Navier-Stokes equations. Similarly, tracers such as temperature and salinity are usually the result of solving an advection-diffusion equation. Prognostic fields typically have specified initial and boundary conditions and it will be necessary to specify spatial and temporal discretisation options. If an implicit timestepping scheme is in use (and it almost always is in Fluidity), it is also necessary to specify solver options.

Diagnostic fields are calculated from other fields without solving a partial differential equation. A typical example is the CFL number which may be calculated from the timestep, the mesh spacing and the velocity field.

Prescribed fields receive their values from sources external to Fluidity. This might be a constant or varying function specified by the user, or it might be interpolated from some external data set. Fields such as diffusivity and viscosity are often prescribed as are source and absorption terms.

An additional field type - aliased - is also available. This links the values in one field to those in another, using no extra computational resources during the simulation (i.e. it is not an independent field). This is useful when sharing fields between material_phases. For example if two material_phases share a common velocity field then only one should contain a prognostic field while the other is aliased to the other material_phase.

8.6.2 Setting field values

Field values must be specified by the user in two circumstances: the initial value of most prognostic fields and the value throughout the simulation of all prescribed fields.

The initial value of prognostic fields is set with the `.../prognostic/initial_condition` option while the value of prescribed fields is set with the `.../prescribed/value` option.

8.6.2.1 Constant fields

Fields which are constant in space and (for prescribed fields) time may be specified by simply providing a constant value in the `constant` option under `.../prognostic/initial_condition`, `.../prescribed/value`. For a scalar field, this is a single floating point (real) value while for a vector field this is a list of reals of length equal to the field dimension.

For tensor valued fields there are more options. It is possible to specify an isotropic, or rotation invariant, value by choosing `.../value/isotropic/constant` and specifying a single real which will be used for all the diagonal entries of the tensor field at all mesh nodes. The off-diagonal entries of an isotropic tensor field are always zero. A constant anisotropic field may be specified by choosing `.../value/anisotropic_asymmetric/constant` and providing the entire matrix. Finally, a constant symmetric anisotropic tensor field may be specified by selecting `.../value/anisotropic_symmetric/constant`. In this case, the user must specify all of the entries in the upper half of the matrix and those in the lower half will be filled automatically by symmetry.

8.6.2.2 Setting fields with a Python function

The value of a field which varies in space and (for prescribed fields) in time may be specified by providing an appropriate function written in Python. The Python function will be evaluated for each node in the mesh at the start of the simulation to populate the field values. For time-varying prescribed fields, the function will be evaluated again at the beginning of every timestep to update the field value. If it is known that the value of the field does not in fact vary in time, then the re-evaluation of the Python function on each timestep can be inhibited by setting the `.../prescribed/do_not_recalculate` option.

The Python function must be provided as the value of the `.../python` option which may be chosen as an alternative to the `.../constant` function. The option may contain any Python code but it must define a function `val(X,t)` where the sequence `X` is the coordinates of the point at which the field is being evaluated and `t` is the current time.

For a scalar field, the function must return a single floating point value. Similarly for a vector field, a sequence of values of length equal to the field dimension must be returned. For a tensor field, there are two cases. For an isotropic field specified with `.../value/isotropic/python`, the function must return a single float which will be used for all the diagonal entries of the tensor at that point. The off-diagonal entries will be set to zero. For the anisotropic case, the function must return a two-dimensional array (a sequence of sequences). It is the user's responsibility to ensure that the tensor is symmetric in cases where it should be.

```
def val(X,t):
    return (-X[1],X[0])
```

Example 8.5: A Python function returning a two-dimensional solid rotating vector field about the origin.

8.6.2.3 Reading fields from a file (using the `from_file` option)

A field can be populated using saved data from a file. This is intended primarily for picking up prescribed fields from previously run prognostic simulations (checkpointing) and may be specified by providing the file name in the `from_file` option under `.../prognostic/initial_condition`, `.../prescribed/value`.

For prescribed fields the format of the input file containing field data must be `vtu`, and this will only work for those prescribed fields directly underneath `.../material_phase`. For prognostic fields it is possible to select the type of input file, under `.../initial_condition/from_file/format`; the available supported formats for this include `vtu` and `NetCDF-CF 1.x`.

The file mesh must match the mesh of this field (except for piecewise constant fields which will be remapped back from the discontinuous nodal values). In parallel the process number is appended to the filename, e.g. if the file name attribute is set to `input.vtu`, process 0 reads from `input-0.vtu`.

8.6.2.4 Setting an initial condition from a NetCDF file

The initial state of certain fields can be set from external data contained within a NetCDF file. This functionality can be used by selecting the `.../initial_condition/from_netcdf/format` option. This feature is currently intended for setting, for example, the initial free surface elevation in a Tsunami simulation. This option will not currently work with multi-layered data files. Supported NetCDF file conventions include the NetCDF-CF 1.x convention.

8.6.2.5 Setting fields from NEMO data

Initial conditions of prognostic fields and the values of prescribed fields can also be set from an external NEMO data file. The external data file is in the NETCDF format and data is currently available for pressure, temperature, salinity and velocity. To set the initial condition of a prognostic field from NEMO data, chose the option `.../prognostic/initial_condition/NEMO_data` and then under `format` select the required data format. For scalar fields the formats available are ‘Temperature’, ‘Salinity’ and ‘Free-surface height’, for vector fields ‘Velocity’ is the only available format. Setting the value of prescribed fields from NEMO data works similarly. Set the option `.../prescribed/value/NEMO_data` and then proceed as above.

8.6.3 Region IDs

If the input mesh defines a number of region IDs then these may be employed to specify different field values for each region. For a prescribed field, this is achieved by changing the `.../value::WholeMesh` element to the unnamed `.../value`. The user must then specify a new name for that value element. Next, enable the `.../value/region_ids` option and set it to a list of region ids to which this value should apply. Any number of `.../value` elements may be added to allow different values to be specified in different regions. For prognostic fields, analogous behaviour for initial conditions may be achieved by switching `.../initial_condition` from `WholeMesh` to a user-specified name and specifying the region IDs appropriately.

See section [E.1.5](#) for information on including region IDs in meshes.

8.6.4 Mathematical constraints on initial conditions

For well-posedness, the initial condition of the velocity field must satisfy both continuity ([2.45b](#)) and the boundary conditions imposed on the problem. If the normal component of velocity is imposed on the entire boundary then the additional compatibility constraint of global mass conservation must be satisfied:

$$\int_{\partial\Omega} \mathbf{n} \cdot \mathbf{u} = 0.$$

8.7 Advected quantities: momentum and tracers

8.7.1 Spatial discretisations

A number of underlying finite element schemes are available for tracer fields and velocity. In each case there are restrictions on the mesh continuity which must be employed. In addition, the `conservative_advection` option is applicable to all discretisation types. See [chapter 3](#) for details.

For each field, the spatial discretisations can be selected using `.../prognostic/spatial_discretisation`. Once selected a number of other options will open underneath this option.

8.7.1.1 Continuous Galerkin

Continuous Galerkin (CG) implements the CG scheme detailed in [3.2.1.3](#). If stabilisation methods are needed, the user can either select streamline upwind or streamline upwind Petrov-Galerkin. Other options include integration of advection by part, lumping of the mass matrix, or direct exclusion of both advection and mass terms.

8.7.1.2 Control Volumes

The control volume options (`control_volumes`) implements the advection scheme described in section 3.2.4. There are several options to control the face value and how diffusion is implemented.

The face value (`face_value`) can be set to one of:

FirstOrderUpwind - see section 3.2.4.1. Note that first order upwinding does not require nonlinear advection iterations as the low order pivot solution uses first order upwinding itself. However in this case it is necessary that the implicitness factor, θ , is the same as the pivot implicitness factor, θ_p (see sections 3.4.2 and 8.7.2).

Trapezoidal - see section 3.2.4.1, should be used with the suboptions describing a face value limiter (see section 3.2.4.1) active

FiniteElement - see section 3.2.4.1, should be used with the suboptions describing a face value limiter (see section 3.2.4.1) active

FirstOrderDownwinding - see section 3.2.4.1, intended for demonstration purposes only, not recommended for general use (unconditionally unstable)

HyperC - see section 3.2.4.1

UltraC - see section 3.2.4.1

PotentialUltraC - see section 3.2.4.1

None - turns off the advective terms

For the diffusion scheme (`diffusion_scheme`) one can choose either:

ElementGradient - see section 3.2.4.2

BassiRebay - works in two configurations equal order field and diffusivity or, for fields on a linear parent mesh, with a piecewise constant (element centred) diffusivity (staggered finite volumes), see section 3.2.4.2

For steady state problems the mass terms may be disabled using `.../mass_terms/exclude_mass_terms`. Note that this also requires the suitable setting of the temporal discretisation options ($\theta = 1$).

8.7.1.3 Coupled CV

The coupled CV options (`coupled_cv`) implement another control volume discretisation with face value limits enforced in such a way to give boundedness both in the field and across the sums of fields. Section 3.2.4.1 contains more details on this method.

Options must be selected to describe the face value scheme, which include most of the algorithms described in section 8.7.1.2. Additionally it is necessary to prescribe the maximum and minimum bounds on the sum of this and the previous fields. Because the coupled scheme depends on a priority ordering a priority (outside of the spatial discretisation options at `.../scalar_field/prognostic/priority`) must also be set with higher values having the highest priority and lower values the lowest.

Related fields to be used together during coupled limiting are grouped together based on their names from successive material phases. For example, if a field called `MaterialVolumeFraction` has `coupled_cv` options then all other fields in all other material phases called `MaterialVolumeFraction` using `coupled_cv` options will be advected together in order of their priority. Spatial discretisation options within `coupled_cv` may vary between the fields but temporal discretisation options must be identical.

8.7.1.4 Discontinuous Galerkin method for the advection-diffusion equation

The Discontinuous Galerkin option implements the advection-diffusion algorithm described in Section 3.2.3.1. There are two compulsory options to set, as well as a number of non-compulsory options.

Advection scheme (`advection_scheme`) This *compulsory option* selects the approximation for the flux of scalar across a face. Select one from:

- `upwind`: Use the upwind flux as described in Section 3.2.3.1. This is the *recommended flux* for DG advection.
- `lax_friedrichs`: Use the Lax-Friedrichs flux as described in Section 3.2.3.1. This is an attempt to produce a bounded flux when the advecting velocity is discontinuous. This option is only for testing, if you have a discontinuous advecting velocity it is recommended to use the upwind flux combined with the option to project the velocity to a continuous space described below.
- `none`: This option switches off the advection term completely.

```
project_velocity_to_continuous
integrate_advection_by_parts
integrate_conservation_term_by_parts
```

Diffusion scheme (`diffusion_scheme`) This *compulsory option* selects the discretisation method used for the diffusivity term. This selection is important for performance since various different options have different stencil sizes, which affects memory signature and hence the number of elements you can use per processor.

Select one from:

- `bassi_rebay`: The classical scheme of Bassi and Rebay (see section 3.2.3.3). This scheme results in a large stencil for the diffusion matrix, which can reduce computational speed and increase memory use. If possible one should use a different option with a smaller stencil.
- `compact_discontinuous_galerkin`: The compact discontinuous Galerkin scheme (CDG) from Peraire and Persson [Peraire and Persson, 2008] (see section 3.2.3.3). This scheme has the smallest stencil of any diffusion scheme and hence is the most efficient and uses the least memory resource. *Recommended option*.

Optionally, it is possible to set the `penalty_parameter`. This optional option adds an extra term which penalises jumps across faces. You must supply a multiplicative constant which is scale independent (typical value is 10). This term is required to prove theoretical results about the CDG scheme, but we experimentally observe that it is not necessary, and hence, it is *recommended not to use this option*.

- `interior_penalty`: Symmetric interior penalty (IP) scheme. This scheme simply integrates the diffusion operator by parts in each element, averages the fluxes and adds a term which penalises jumps. You must set the `penalty_parameter` which sets the multiplicative constant, and the `edge_length_parameter` which specifies the scaling with the edge-length h . You must also select an `edge_length_option` which is either `use_face_integral` which computes a length scale from the face integral, or `use_element_centres` which uses the distance between centres of the two elements on either side of the face. Both of these options only function well for nearly isotropic meshes and hence CDG is the recommended diffusion choice since it is compact and requires no such parameters.

Slope limiter (`slope_limiter`) Need to mention about subcycling.

Mass terms (`mass_terms`)

8.7.1.5 Conservative advection

The momentum equation can be discretised conservatively by setting the BETA factor equal to 1 (corresponding to a divergence form of the equation). If BETA is set to zero, the discretisation is left non-conservative. An intermediate value can alternatively be selected. Please refer to section 3.5 for a comprehensive discussion on the influence of this parameter.

8.7.2 Temporal discretisations

Under temporal discretisation, you can set the value of theta, where 0 is explicit, 0.5 is Crank-Nicolson and 1 is implicit. For scalar fields, the control volumes option may be selected if you are using control volumes or coupled cv spatial discretisation. It contains options to set up nonlinear advection iterations, subcycling and the value of the pivot implicitness factor (see section 3.4.2). The discontinuous Galerkin option can be used if you are using discontinuous galerkin spatial discretisation to set the maximum courant number per subcycle, or the number of subcycles.

8.7.3 Source and absorption terms

The source and absorption terms allow for external forcing of the tracer and momentum equations. The source is a rate of change of the tracer which is independent of the system state while the absorption term is linear in the tracer. The source and absorption terms modify the tracer equation as follows (cf. (2.1)):

$$\frac{\partial c}{\partial t} = F(c, u, t) - \sigma c + F, \quad (8.1)$$

where $F(c, u, t)$ represents the advection and diffusion terms, σ is the absorption and F is the source. The source and absorption are usually prescribed fields supplied by the user but in some cases it may be necessary to provide a diagnostic field which will be set by a parameterisation somewhere in the model. If this is the case then this will be specified in the documentation of that parameterisation.

For tracer fields, the source and absorption are specified by the

```
.../scalar_field/prognostic/scalar_field::Source and
.../scalar_field/prognostic/scalar_field::Absorption options respectively. For
velocity, the corresponding fields are naturally vector-valued and are set by options
.../vector_field::Velocity/prognostic/vector_field::Source and
.../vector_field::Velocity/prognostic/vector_field::Absorption respectively.
```

8.7.4 Sponge regions

It is often useful to be able to relax momentum or a field variable to a given state in regions of the domain, typically in regions close to boundaries. This may be done using a combination of source and absorption terms. If F is the value of the source at a point and σ is the absorption, then the value of the scalar field c will tend to relax to a value of F/σ . The absorption, σ , has units 1/time and controls the time over which the relaxation occurs.

8.8 Solving for pressure

8.8.1 Geostrophic pressure solvers

8.8.2 First guess for poisson pressure equation

Fluidity's solution procedure for velocity and pressure can use a pressure poisson guess to speed up the convergence. In order to use a pressure guess, set `.../scalar_field::Pressure/prognostic/scheme/poisson_pressure_solution` from `never` to `only_first_timestep`.

8.8.3 Removing the null space of the pressure gradient operator

If the normal component of velocity is imposed on all boundaries then the appropriate boundary condition for pressure [see [Gresho and Sani, 1987](#)] is obtained by taking the normal component of (2.45a), this yielding a Neumann boundary condition for pressure. This only serves to define the pressure field up to an arbitrary additive constant.

There are two different and mutually exclusive options which may be used to fix the additive constant in the pressure field. The first is that the pressure at a single point may be set to 0. This is achieved by setting the `.../scalar_field::Pressure/prognostic/reference_node` option. The value of the option is the number of a node at which the pressure is to be constrained to be zero. It is an error for the node number specified here to be greater than the number of nodes in the simulation.

The second method is to instruct the linear solver to remove the null space of the pressure equation as a part of the solution procedure. This is achieved by enabling the `.../scalar_field::Pressure/prognostic/solver/remove_null_space` option. This approach often leads to better convergence rates than setting the `reference_node`.

If however there is a single location on the boundary where the normal component of velocity is not specified then there is no free constant in the pressure and neither `.../reference_node` nor `.../remove_null_space` should be set. An example may be stress free outflow or the presence of a free surface.

8.8.4 Continuous Galerkin pressure with control volume tested continuity

As described in section 3.7.1 when using a continuous Galerkin discretisation of pressure the continuity equation can be tested with the corresponding dual control volume mesh. This is achieved by including the `.../scalar_field::Pressure/prognostic/spatial_discretisation/continuous_galerkin/test_continuity_with_cv_dual` option. As described in the theory section 3.7.1 this will imply a non symmetric pressure correction matrix which must be considered when selecting the pressure matrix solver options.

The current limitations of this method are:

1. It can only be used for incompressible flow 3.6.
2. It cannot be used with the free surface model 3.9.
3. It cannot be used with the wetting and drying model 3.10.
4. It cannot be used with the implicit solids model with two way coupling.
5. It can only be used if the pressure has a mesh associated with Lagrangian shape functions.

6. It can only be used with control volume shape functions that are available, of which only P1CV are considered reliable.

8.9 Solution of linear systems

8.9.1 Iterative Method

As described in Section 3.11, for the solution of large sparse linear systems, the so called iterative methods are usually employed. These methods avoid having to explicitly construct the inverse of the matrix, which is generally dense and therefore costly to compute (both in memory and computer time). FLUIDITY is linked to PETSc: a suite of data structures and routines for the scalable (parallel) solution of scientific applications modelled by partial differential equations. It employs the MPI standard for parallelism. FLUIDITY therefore supports any iterative method provided by the PETSc library (<http://www-unix.mcs.anl.gov/petsc/petsc-2/snapshots/petsc-dev/docs/manualpages/KSP/KSPType.html> — available methods may depend on the PETSc library installed on your system). Examples include Conjugate Gradient (CG), GMRES and FGMRES (Flexible GMRES). Some options are explicitly listed under `solver/iterative_method`, for example CG: `solver/iterative_method::cg`, whereas others can be selected entering the name of the chosen scheme in `solver/iterative_method`.

8.9.2 Preconditioner

The requirement for a suitable preconditioner is described in Section 3.11.2. In a manner analogous to the selection of the iterative method, some common preconditioning options are explicitly listed under `solver/preconditioner`, for example MG: `solver/preconditioner::mg`, whereas others can be selected by entering the name of the chosen scheme in `solver/preconditioner`.

8.9.2.1 Direct Solve

Note that the option to solve a system exactly is available in FLUIDITY. For this, `solver/iterative_method::preonly` must be selected (`preonly`: preconditioner only) and the preconditioner must be set to `solver/preconditioner::LU`. A full LU decomposition of the system is then carried out.

8.9.3 Relative Error

The solver finishes if the preconditioned error becomes smaller than the original preconditioned error times this value.

8.9.4 Absolute Error

The solver finishes if the preconditioned error becomes smaller than this value.

8.9.5 Max Iterations

The maximum number of iterations allowed for the linear solver before quitting.

8.9.6 Start from Zero

Switch on to start a solve with a zero vector and not a guess from a previous solve. Note that some solves always start at zero in which case this switch will have no effect (to check this, the user should refer to the log output).

8.9.7 Remove Null Space

As documented in Section 8.8.3, this option removes the null space.

8.9.8 Solver Failures

Three options are available here:

1. Never ignore solver failures: Solver failures are always treated as fatal errors. The model stops at the end of the time step in order to allow for the latest output to be written.
2. Ignore non-convergence during spin-up: Allow for an initial period in which solver failures caused by non-convergence in the maximum number of iterations are ignored.
3. Ignore all solver failures: Ignore all solver failures. This is a dangerous option that should only be used in exceptional cases.

It is recommended that users use the first option: Never ignore solver failures, however, on occasions (e.g. challenging initial conditions) the second might also be applicable.

8.9.9 Reordering RCM

A bandwidth reduction algorithm — reverse Cuthill-McKee reordering — is used to improve cache performance.

8.9.10 Solver Diagnostics

This subsection includes a series of extra diagnostic options to help debug solver problems.

8.9.10.1 Print norms

Print out the norm of vectors and matrices before the solve, and that of the solution vector afterwards. Norms are printed at verbosity level 2, so run Fluidity with `-v2` or `-v3`.

8.9.10.2 Monitors

Options to give extra information for each iteration of the the solve. Note that some of those may really slow down your computation.

8.10 Equation of State (EoS)

The equation of state is a relation between state variables. For incompressible flows it is used to derive the density from other variables such as temperature and salinity (cf. section 2.3.3.2). For compressible flows it can be a more general relation between the state variables including density and pressure.

The following EOS are available:

- `.../equation_of_state/fluids/linear` Is a simple linear equation of state, where density is a function of temperature and salinity.
- `.../equation_of_state/fluids/ocean_pade_approximation` Is a complex EOS for ocean modelling where density is a function of temperature, salinity and pressure.
- `.../equation_of_state/compressible/miegrunneisen` Is a simple compressible material EOS.

8.10.0.3 Linear fluid EOS

The density is a linear function of temperature and salinity:

$$\rho = \rho_0 (1 - \alpha(T - T_0) + \beta(S - S_0)), \quad (8.2)$$

where ρ_0 , α , T_0 , β and S_0 are set by the following options:

- `.../linear/reference_density` sets ρ_0
- `.../linear/temperature_dependency/thermal_expansion_coefficient` sets α
- `.../linear/temperature_dependency/reference_temperature` sets T_0
- `.../linear/salinity_dependency/salinity_contraction_coefficient` sets β
- `.../linear/salinity_dependency/reference_salinity` sets S_0

Note that for Boussinesq the reference density does not influence any of the terms in the momentum equation (see (2.45)). It may influence the outcome of diagnostic fields depending on density.

The option `subtract_out_hydrostatic_level` only changes the buoyancy term. For LinearMomentum it changes to $g(\rho - \rho_0)$ and does not affect the density in the Du/Dt term. For Boussinesq it changes to $g\rho' = g(\rho - \rho_0)/\rho_0$ (again see (2.45)), and this option should always be used. In both cases the diagnostic “Density” field and all other diagnostic fields depending on density still represent the full density.

8.10.0.4 Pade ocean EOS

This EOS is described in section 2.3.3.3. This option uses mean hydrostatic pressure based on depth to calculate the pressure (hence why you need to provide the value of z on the top surface). For this option, the temp field represents potential temperature *not* in situ temperature - so beware (see McDougall et al. [2003] for a formula for converting from in-situ to potential). The units are degrees Centigrade for potential temperature, PSU for salt, kg m^{-2} for density. The reference density is 1000 kg m^{-2} and the momentum equation is Boussinesq using this reference density.

8.10.0.5 Compressible EOS

`.../equation_of_state/compressible/miegrunneisen` defines a simple compressible equation of state that can be used to describe gases, liquids and solids, known as the stiffened gas EOS.

`.../miegrunneisen/reference_density` Specifies the reference density of the material in SI units.

`.../miegrunneisen/ratio_specific_heats` Specifies the ratio of specific heats of the gas minus 1 ($c_p/c_V - 1$) in the perfect gas EOS and the Gruneisen parameter in the stiffened gas equation of state. Not activating this option simplifies the compressible EOS to that of a compressible liquid.

`.../miegrunneisen/bulk_sound_speed_squared` Specifies the bulk sound speed squared for the material c_B^2 . Not activating this option simplifies the compressible EOS to that of a perfect gas.

8.11 Sub-grid Scale Parameterisations

Fluidity contains a number of sub-grid scale parameterisations which model physical process below the resolution of the mesh.

8.11.1 GLS

This option enables the model described in section 4.1.1.1. There are a few different sub-options to configure. First, you must choose which GLS method to use from $k-\varepsilon$, $k-kl$, $k-\omega$ and gen . Next, the stability functions can be chosen. CanutoA or CanutoB are recommended. If you are running a 3D model, then switching on `.../calculate_boundaries` is recommended in order for the boundary conditions to be set correctly. Finally, you can enable a number of optional diagnostic fields.

The user can also choose to relax the diffusivity and viscosity calculated by switching on the `.../relax_diffusivity`. The value specified must be between 0 and 1.0. A value of 0 indicates no relaxation, 1.0 would indicate no changes to be made. If this option is activated, the diagnostic fields `GLSTurbulentVerticalDiffusivity` and `GLSTurbulentVerticalViscosity` must also be activated. In addition, if adaptivity is enabled, these two fields must have an interpolation method set, e.g. `.../GLSVerticalViscosity/diagnostic/consistent_interpolation`.

For each field that will be effected by the subgrid scale parameterisations, you must enable the correct diffusivity. This is done by specifying `.../subgridscale_parameterisation` in the field to GLS. Normally, this would be the temperature, salinity and any biology fields active.

Finally, fields that are altered by the GLS model, such as the Viscosity, need to be switched to a `diagnostic/algorithm::Internal`. The list of fields to switch is:

1. `GLSTurbulentKineticEnergy/Diffusivity`
2. `GLSTurbulentKineticEnergy/Source`
3. `GLSTurbulentKineticEnergy/Absorption`
4. `GLSGenericSecondQuantity/Diffusivity`
5. `GLSGenericSecondQuantity/Source`

6. GLSGenericSecondQuantity/Absorption
7. Velocity/Viscosity

If these fields are not set correctly, a user error will occur.

8.11.2 k - ϵ Turbulence Model

This option enables the turbulence model described in 4.1.1.2. It must not be confused with the $k - \epsilon$ option in the GLS model (see 4.1.1.1) which is only for oceans like problems. The user must select either implicit or explicit source and absorption terms via `.../option`. If `explicit` is chosen, the source and absorption terms are as described in 4.1.1.2. If `implicit` is chosen, the source and absorption terms are calculated as follows:

$$\begin{aligned} \text{source} &= -\min(0.0, \text{diss} - \text{prod}), \\ \text{absorption} &= \max(0.0, \text{diss} - \text{prod}), \end{aligned}$$

where ‘source’ is the explicitly calculated source term and ‘absorption’ is the explicitly calculated absorption term.

In Fluidity the k field is called `TurbulentKineticEnergy` and the ϵ field `TurbulentDissipation`. The model works well with the following spatial discretisation options for k and ϵ : `.../control_volumes/face_value::FiniteElement/limit_face_value/limiter::Sweby` with `/control_volumes/diffusion_scheme::ElementGradient`. This discretisation is significantly speeded up by selecting `/project_upwind_value_from_point/store_upwind_elements_store_upwind_quadrature`.

Fully implicit or Crank-Nicolson temporal discretisation is recommended. If using the above spatial discretisation the `control_volume` discretisation option should be selected, with `number_advection_iterations = 3`.

Fields that are altered by the k -epsilon model need to be switched to `diagnostic/algorithm::Internal`. The list of fields to switch is:

1. Velocity/Viscosity
2. TurbulentKineticEnergy/Diffusivity
3. TurbulentKineticEnergy/Source
4. TurbulentKineticEnergy/Absorption
5. TurbulentDissipation/Diffusivity
6. TurbulentDissipation/Source
7. TurbulentDissipation/Absorption

If these fields are not set correctly, a user error will occur. The eddy viscosity is available as a diagnostic tensor field and as a diagnostic scalar field. However, the molecular (or laminar) viscosity must still be prescribed under `.../k-epsilon/tensor_field::BackgroundViscosity`.

If using additional scalar fields such as temperature, salinity etc, an option is available under `.../subgridscale_parameterisation::k-epsilon` to use the eddy diffusivity scaled by a user-specified Prandtl number.

Three optional diagnostic scalar fields are available within the model: `LengthScale`, `TKEOverEpsilon` and `EpsilonOverTKE`. The first is the integral length scale of the turbulence (see (4.26)), the second is the ratio of the k and ε fields, and the third is its reciprocal, also known as ω or the specific dissipation.

8.11.2.1 Initial and Boundary Conditions

The model is sensitive to the initial conditions specified for the k and ε fields. It is recommended to use the special boundary condition type, `.../k_epsilon` on any surface of interest for these fields. Different wall functions are available for low- and high-Reynolds-number flows. The cutoff between high and low Reynolds number is at the user's discretion but is usually of order 10^4 .

In the low Reynolds number case, (4.29) are implemented by selecting `.../boundary_conditions/k_epsilon/low_Re`. In the high Reynolds number case, (4.31) and (4.32) are implemented by selecting `.../boundary_conditions/k_epsilon/high_Re`. The velocity condition, (4.30), can be selected under `.../Velocity/prognostic/boundary_conditions/type::log_law_of_wall`. However, if using a sufficiently fine near-wall mesh, roughly the correct velocity profile can be obtained with a no-slip Dirichlet BC.

If using Dirichlet conditions, it is recommended that k and ε be set to small values, chosen such that the eddy viscosity on the boundaries is of order $O(\nu)$ (background viscosity), and both fields perhaps of order $O(1/Re)$.

8.11.3 Large Eddy Simulation Models

LES models are available as options under `.../Velocity/prognostic/spatial_discretisation/continuous_galerkin/les_model`. See 4.1.2 for details of the various LES models available. These models require a prescribed viscosity (`.../Velocity/prognostic/tensor_field::Viscosity/prescribed`), to which an eddy viscosity is added to account for subgrid-scale turbulence. The LES models are currently restricted to use in incompressible flow cases, where the discrete velocity is divergence-free and the eddy viscosity tensor is traceless.

8.11.3.1 Second-Order Smagorinsky

The modified second-order Smagorinsky model of [Bentham \[2003\]](#) is available under `.../les_model/second_order`. The Smagorinsky coefficient (`.../second_order/smagorinsky_coefficient`) must be set; its value should be that suggested by the literature for a particular flow type. A reasonable all-round figure is 0.1. The eddy viscosity is available as an optional diagnostic field (`.../second_order/tensor_field::EddyViscosity`).

8.11.3.2 Fourth-Order Smagorinsky

The fourth-order Smagorinsky model of [Bentham \[2003\]](#) is available under `.../les_model/fourth_order`. The Smagorinsky coefficient (`.../second_order/smagorinsky_coefficient`) must be set; 0.1 is recommended. A fine mesh is required to get good results from this model.

8.11.3.3 WALE

The wall-adapted local eddy viscosity (WALE) model is available under `.../les_model/wale`. The Smagorinsky coefficient (`.../second_order/smagorinsky_coefficient`) must be set; 0.1 is recommended.

8.11.3.4 Dynamic LES

The Germano dynamic LES model is available under `.../les_model/dynamic_les`. The following options have to be set: first, the filter width ratio α (`.../dynamic_les/alpha`); 2 is recommended. Second, the solver options (`.../dynamic_les/solver`) are for solving the inverse Helmholtz equation for the test-filtered velocity; `cg/SOR` is recommended.

Optional options:

- `.../dynamic_les/enable_lilly`: use the Lilly modification to the Germano model. It is recommended.
- `.../dynamic_les/enable_backscatter`: allows negative eddy viscosity, which may result in more realistic turbulent flow if the mesh resolution is fine enough.

Several diagnostic fields are available if desired:

- `.../dynamic_les/vector_field::FilteredVelocity`: the velocity field filtered with the test filter.
- `.../dynamic_les/tensor_field::FilterWidth`: the mesh size tensor
- `.../dynamic_les/tensor_field::StrainRate`: the strain rate \overline{S}_{ij} .
- `.../dynamic_les/tensor_field::FilteredStrainRate`: the filtered strain rate \widetilde{S}_{ij} .
- `.../dynamic_les/tensor_field::EddyViscosity`: the eddy viscosity ν_T .

8.12 Boundary conditions

The simulated system requires suitable boundary conditions for full closure. An example could be the amount of sunlight at the ocean surface, a specified value of temperature heating material from below, or a momentum stress in the form of wind for velocity. It is also possible to leave boundary conditions undefined, in which case "stress-free" conditions are applied. See section 2.2.2 for further details.

8.12.1 Adding a boundary condition

Boundary conditions are set for each field contained in state under `.../boundary_conditions`. Multiple boundary conditions can be set for each field, such that the sides, surface and bottom can have different conditions. A boundary conditions is added by clicking the "+" symbol in the appropriate field

8.12.2 Selecting surfaces

To each boundary condition a set of domain surfaces is assigned on which it is applied to. The surfaces are identified by a surface ID specified during the mesh generation procedure (see section E.1.4). For example if the top and bottom of your mesh is defined as surface 1, then simply add a 1 to `.../boundary_conditions/surface.ids`. Multiple surfaces can be added, separated by a space.

8.12.3 Boundary condition types

Fluidity supports a wide range of boundary conditions which will be introduced in the next sections.

8.12.3.1 Dirichlet

A Dirichlet condition sets the value of the field (c) at each location over the surface $\partial\Omega$:

$$c(\mathbf{x}) = f(\mathbf{x}) \quad \text{on } \partial\Omega.$$

Dirichlet boundary conditions can also be applied weakly by selecting the `.../apply_weakly` option. Unlike the strong form of the Dirichlet conditions, weak Dirichlet conditions do not force the solution on the boundary to be pointwise equal to the boundary condition.

8.12.3.2 Neumann

A Neumann boundary condition sets a flux term q to the normal (\mathbf{n}) of the surface $\partial\Omega$:

$$\int_{\partial\Omega} \varphi(\bar{\kappa}\nabla c) \cdot \mathbf{n} \, d\Gamma,$$

where φ is a test function (see section 3). The Neumann condition is specified by assigning a value to the q , where

$$q = (\bar{\kappa}\nabla c) \cdot \mathbf{n}, \quad \text{on } \partial\Omega.$$

8.12.3.3 Bulk formulae

These boundary conditions can be used on:

- Salinity
- Temperature
- Velocity
- PhotosyntheticRadiation

They use meteorological data and convert it into a Neumann or Dirichlet boundary condition as appropriate for the fields above. You do not need to have all the above fields; only velocity and temperature are required. More information can be found in section 8.12.5.1.

8.12.3.4 Zero flux

For control volume discretisations only, this option prevents the field fluxing from the boundary.

8.12.3.5 Flux

For control volume discretisations only, this option allows a given flux h of field c through the boundary. In other words, we have

$$\frac{\partial c}{\partial t} = h$$

8.12.3.6 Free surface

The `.../free_surface` option allows the height of upper surface to vary according to the pressure and velocity fields. This boundary condition is available on the velocity field only. It is recommended you activate a diagnostic Free Surface field also.

By default, the mesh geometry is not influenced by the free-surface calculation, however Fluidity can deform the mesh according to the free-surface elevation. This option is available at `/mesh_adaptivity/mesh_movement/free_surface`.

8.12.3.7 Wetting and drying

In order to use wetting and drying, first switch on the mesh deformation as described in [8.12.3.6](#).

Secondly, if the mesh is extruded within fluidity, the extrusion parameters have to be changed such that areas above sea level are included. For example if a bathymetry map file is used for the extrusion, the option `/geometry/mesh/from_mesh/extrude/regions/bottom_depth/from_map/surface_height` can be used to shift down the domain such that the whole bathymetry is below zero. A non-zero initial pressure together with the relationship between pressure and free-surface elevation $p = \rho\eta$ can be used to shift the initial free-surface down accordingly as well.

Finally, wetting and drying is activated under `/mesh_adaptivity/mesh_movement/free_surface/wetting_and_drying`. The only required parameter is the wetting and drying threshold value d_0 , which specifies the minimum layer-thickness that is retained in dry areas. Following equation can be used to determine the threshold value:

$$d_0 = \frac{l\Delta x}{r},$$

where Δx and l are the maximum horizontal element size and number of mesh layers in the dry areas, respectively and r is the maximum aspect ratio. A typical value for latter is between 500 – 1000.

8.12.3.8 Drag

This option applies a quadratic or linear drag to the Velocity field. Both the value and the type of drag need to be set. A Manning-Strickler drag can be used by activating `.../quadratic_drag/manning_strickler`

8.12.3.9 Wind forcing

A wind forcing can be applied to the Velocity field as either a stress or velocity. For stress values, the physical units should match those of the simulation, so for example, if you use the non-dimensional value of ρ as 1.0, your stresses (in $\text{kgm}^{-1}\text{s}^{-2}$) should be divided by the reference density. If using wind velocity (at 10m height) the density of the air needs to be specified in the same units, i.e. $\rho_{\text{air}} = 1.3 \times 10^{-3}$.

Alternatively `.../Velocity/boundary_conditions/wind_forcing/wind_stress` sets the value of wind forcing from a NETCDF file. The NETCDF file must contain East-West and North-South components, along with times locations (latitude/longitude) for the values. In addition, one must set `/timestepping/current_time/time_units` in order for the simulated time to be matched to the NETCDF data.

8.12.3.10 No normal flow

When using `.../control_volumes` under Pressure `.../spatial_discretisation` or when using `.../integrate_continuity_by_parts` with CG Pressure and Velocity this boundary condition type imposes a weak no normal flow boundary condition on the surfaces specified.

8.12.3.11 Near-wall treatment

This option implements a penalty function for the near wall region, negating the need to use fine meshes near walls [Bazilevs et al., 2007]. This option should be used in conjunction with a `.../no_normal_flow` boundary on the same surface.

8.12.3.12 Log-law of wall

This option sets the velocity to proportional to the logarithm of the distance from the boundary. A surface roughness needs to be specified which is the thickness of laminar sublayer.

8.12.4 Special input data for boundary conditions

When running free surface simulations the surface elevation at the boundary is specified by applying a pressure Dirichlet condition. Since the free surface elevation is often measured data, there are some special possibilities to specify a pressure Dirichlet condition:

`.../from_file` allows the specification of a single file containing something useful. This option is available on the Pressure (Free Surface) field. Tidal boundary conditions can be applied by setting this option and referencing a relevant NetCDF file containing appropriate amplitude and phase data for the desired tidal constituent(s). The file is referenced under:

- `.../tidal/file_name,`

with the amplitude and phase names (as specified in the NetCDF file) set under:

- `.../tidal/variable_name_amplitude,`
- `.../tidal/variable_name_phase`

respectively. Finally, the constituent should be selected from the list under:

- `.../tidal/name.`

A separate tidal boundary condition needs to be set for each constituent.

`.../NEMO_data` will set the field according to a specified NEMO input file. This option is available for the Pressure (Free Surface) field. In order to use this option a prescribed field containing NEMO pressure field data must first be created. See section 8.6.2.5 for information on setting prescribed

fields from NEMO data. Then, under `.../NEMO_data/field_name`, set the string to that of the prescribed field containing the NEMO pressure data to enable this option.

`.../synthetic_eddy_method` Available for velocity. This generates statistically realistic turbulent flow at an inflow using a statistical method (for a full explanation see [Jarrin et al. \[2006\]](#)). The user specifies a mean velocity (e.g. `python profile`), turbulence lengthscale, Reynolds stress profile and number of samples. This is useful for high-Reynolds-number industrial CFD flow, and/or if using an LES model.

8.12.5 Special cases

There are a few special cases of boundary conditions that are not applied using the methods described above. These include ocean surface forcing and the boundary conditions on the General Length Scale (GLS) turbulence model.

8.12.5.1 Ocean surface forcing

Ocean surface forcing takes parameters from ERA40 datasets, passes them through bulk formulae and gives a boundary condition for the salinity, temperature, photosynthetic radiation and velocity fields. The settings for these options are in `/ocean_forcing/bulk_formulae`. However, you must also set up `/timestepping/current_time/time_units`.

Under `/ocean_forcing/bulk_formulae` an input file must be defined. The fields on which bulk formulae are to be imposed should have their upper surface set to the correct boundary condition type (`bulk_formulae`). The input file must contain the following ERA40 parameters for the duration of the simulated time:

- 10 metre U wind component (m s^{-1})
- 10 metre V wind component (m s^{-1})
- 2 m temperature (K)
- Surface solar radiation downward (Wm^{-2}s)
- Surface thermal radiation downward (Wm^{-2}s)
- Total precipitation (ms)
- Run off (ms)
- 2 m dew point temperature (K)
- Mean sea-level pressure (Pa)

These variables are surface variables as defined by data files from the ERA40 website. Note that some parameters are accumulated values and as such are required to be divided by the ERA40 temporal resolution - Fluidity assumes 6 hour temporal resolution. These parameters are used as input to the default bulk forcing formulae of [Large and Yeager \[2004\]](#) included in Fluidity. Other formulae are available: COARE 3.0 [[Fairall et al., 2003](#)] and those of [Kara et al. \[2005\]](#) which are based on the COARE data.

Other options under ocean surface forcing include specifying a latitude and longitude, and using a single position for the forcing data. These options are only really useful when simulating pseudo-1D columns (see the `gls-StationPapa` test for an example of a pseudo-1D column). Enabling the `position` option allows the user to specify a latitude and longitude as two real numbers (e.g. 50.0

-145.0 for 50° N and 145° W). These co-ordinates are translated into cartesian co-ordinates, which are then added to the positions of the surface of the mesh. This allows the use of simple mesh geometries and co-ordinates, whilst still specifying where the forcing data should originate. Moreover, the `single_location` option forces *all* surface nodes to receive the same forcing.

Finally, it is possible to output the fluxes that are imposed on the ocean surface, by enabling the `output_fluxes_diagnostic` option. Here, the user can enable diagnostic fields for momentum, heat, salinity and photosynthetic radiation downwards. The fluxes will then be included in the output as normal scalars or vectors, but with values confined to the upper surface.

8.12.5.2 GLS sub-grid scale parameterisation

The GLS model (see section 4.1.1.1) requires that Neumann boundary conditions are set for stability, however, the boundary conditions on the Generic Second Quantity (Ψ) depend on other modelled variables. In order for the boundary conditions to be set correctly, enable the `.../subgridscale_parameterisations/GLS/calculate_boundaries` option.

8.12.5.3 k-epsilon sub-grid scale parameterisation

The k-epsilon turbulence model (see section 4.1.1.2) should apply zero Dirichlet boundary conditions to the TurbulentKineticEnergy (k) field. The TurbulentDissipation (ε) field should use the special type of Dirichlet condition called `k_epsilon` which is calculated in the k-epsilon module. To enable calculation of the boundary conditions on both fields, set the `.../subgridscale_parameterisations/k-epsilon/calculate_boundaries` option.

8.13 Astronomical tidal forcing

Astronomical tidal forcing can be switched on for 11 different constituents under:

- `/ocean_forcing/tidal_forcing,`

(see Wells, 2008 for descriptions of the different constituents). These can be switched either individually or in combination. In addition, a body tide correction can be stipulated under:

- `.../tidal_forcing/love_number,`

for which the suggested value is 0.3 (assuming Love numbers of $k=0.3$ and $h=0.61$; see section 2.4.5.3).

Note that for many cases, specifically those involving open boundaries, it is often desirable to combine astronomical tidal forcing with a co-oscillating boundary tide condition (see section 8.12.4).

8.14 Ocean biology

Enabling this turns on the ocean biology model. In addition you also need to add several scalar fields in the first material phase:

- Phytoplankton
- Zooplankton

- Nutrient
- Detritus
- Primary production

There are several items that need configuring before biology can be used. First a relationship between sources and sinks needs encoding. This is best done by importing `fluidity.ocean.biology` into `/ocean_biology/pznd/source_and_sink_algorithm` and calling the models from there. An example is given below.

```
import fluidity.ocean_biology as biology

day=1./(3600*24)

p={}
p["alpha"]=0.015*day
p["beta"]=0.75
p["gamma"]=0.5
p["g"]=1*day
p["k_N"]=0.5
p["k"]=0.5
p["mu_P"]=0.1*day
p["mu_Z"]=0.2*day
p["mu_D"]=0.05*day
p["p_P"]=0.75
p["v"]=1.5*day

biology.pznd(state, p)
```

Example 8.6: A Python function that imports the biology module and sets the algorithm to use.

The final thing to change is to add absorption coefficients in the photosynthetic radiation field for water and plankton concentration.

8.15 Sediment model

Fluidity contains a basic sediment model in which sediment is treated as a tracer with a settling velocity and the amount of sediment falling through the bottom boundary is recorded. Enabling sediments creates two template fields: one for sediment concentration in the fluid and one for the flux of sediment over the bottom boundary. These can be found at `.../sediment/scalar_field::SedimentTemplate` and `.../sediment/scalar_field::SedimentFluxTemplate` respectively. It is also possible to specify multiple sediment classes, each of which can contain a different settling velocity and hence when coupled with the density, represent different sediment grain sizes. At least one class must be present.

The sediment classes can use either the template initial concentration or set their own.

Note: To use sediment, a linear equation of state must also be enabled `.../equation_of_state/fluids/linear`

8.16 Large scale low aspect ratio ocean simulations

This section contains advice for running a large scale ocean simulation with a large aspect ratio. This section is split into options that must be used and options that are recommended.

8.16.1 Options that must be switched on

These options are almost always recommended for large scale ocean problems.

8.16.1.1 Meshes

The mesh that you use must be a two plus one mesh which is unstructured in the horizontal and structured in the vertical. It can either be constructed in gmsh and read into fluidity, or a two-dimensional mesh can be made in gmsh and extruded within fluidity (see 6.4). In addition, these mesh settings are recommended:

- The Velocity mesh must be discontinuous galerkin polynomial order one, so set `.../geometry/mesh (VelocityMesh) /from_mesh/mesh_shape/mesh_continuity` to `discontinuous`.
- The Pressure mesh must be of polynomial order two, so set `.../geometry/mesh (PressureMesh) /from_mesh/mesh_shape/polynomial_degree` to `2`.
- The temperature and salinity are solved on a continuous galerkin, polynomial order one mesh (no special mesh options).
- Also under geometry, the `.../geometry/ocean_boundaries` option must be switched on, with the surface id's specified.

8.16.1.2 Time stepping

- `.../timestepping/nonlinear_iterations` option must be more than 1, normally 2 is a good choice.
- `.../material_phase/equation_of_state/subtract_out_hydrostatic_level` must be on.

8.16.1.3 Velocity options

The velocity is discontinuous galerkin. The required options are listed below.

- `.../equation` must be set to `Boussinesq`
- `.../spatial_discretisation` must be `discontinuous galerkin`
- `.../spatial_discretisation/advection` is `upwind`
- `.../spatial_discretisation/advection/integrate_advection_by_parts` is `twice`

8.16.1.4 Advected scalar fields (temperature, salinity etc)

The temperature and salinity are continuous galerkin.

- `.../spatial_discretisation` must be Continuous Galerkin

8.16.1.5 Pressure options

This is continuous Galerkin discretisation, with a mesh of polynomial order two (already specified above).

- `.../spatial_discretisation` must be Continuous Galerkin
- `.../solver/vertical_lumping` must be on

8.16.2 Recommended or optional settings

These settings may be recommended, but this section is not intended to be a list of instructions.

8.16.2.1 Meshes

The quadrature degree is usually four in these cases.

8.16.2.2 Velocity options

- `.../spatial_discretisation/discontinuous_galerkin/lump_mass_matrix` is **off**
- `.../spatial_discretisation/viscosity_scheme` can be Bassi Rebay or compact discontinuous galerkin
- `.../spatial_discretisation/advection/conservative_advection` is set to 0.0
- `.../temporal_discretisation/theta` is 0.5 (Crank-Nicolson), or 1 if the advection term is switched off (e.g. during spinning up)
- `.../temporal_discretisation/conservative_advection` is set to 0.0
- `.../solvers` normally gmres - or cg if the advection term is switched off (e.g. during spinning up)
- `.../solvers/preconditioner` eisenstat - or try mg if using compact discontinuous Galerkin vorticity

You might want to create a Viscosity field under Velocity and set it to the required value.

An Absorption field may need to be added under Velocity to allow larger time steps to be taken, otherwise your time steps will be limited by the scale of the baroclinic waves. This term should have a vertical component equal to $\frac{1}{\rho_0}\theta\Delta t g \frac{\partial \rho}{\partial z}$ and the other components are zero. ρ_0 is the reference density, θ is the value set under

`.../Velocity/temporal_discretisation/theta`, Δt is the timestep, g is the acceleration due to gravity and $\frac{\partial \rho}{\partial z}$ is the background density stratification. The absorption term can be a constant if the background stratification is constant. Otherwise, set it with a python function. Also turn on the `.../Absorption/include_pressure_correction` option.

8.16.2.3 Free Surface Field

This can be added if required. You should also select a free surface boundary condition under Velocity.

8.16.2.4 Pressure options

- `.../spatial_discretisation/remove_stabilisation_term` is switched on
- `.../spatial_discretisation/integrate_continuity_by_parts` is switched on
- `.../scheme/poisson_pressure_solution` is only first time step
- `.../scheme/use_projection_method` is on
- `.../solver` is normally `cg` or `gmres`
- `.../solver/preconditioner` is normally `mg`

8.17 Geophysical fluid dynamics problems

This section contains advice for running Geophysical Fluid Dynamics (GFD) problems, such as laboratory-scale flows e.g. the lock-exchange and the annulus or smaller-scale ocean problems e.g. a gravity current on an incline. This section is arranged by options for the different levels of the options tree. Both continuous-Galerkin (P1-P1) and discontinuous-Galerkin (P1DG-P2) discretisations may be used, chapter 3, and different options choices are distinguished where necessary.

8.17.1 Problem type

The `.../problem_type` option should be set to `fluids` or `ocean`.

8.17.2 Geometry

For both P1-P1 and P1DG-P2 `.../geometry/mesh::CoordinateMesh` is required. For P1-P1 this can then be used for both the velocity and pressure fields so `.../geometry/mesh::VelocityMesh` and `.../geometry/mesh::PressureMesh` do not need to be set.

For P1DG-P2 the velocity mesh requires discontinuous Galerkin to be selected for continuity and the pressure mesh must have polynomial order two. To do this set:

- `.../geometry/mesh::VelocityMesh/from_mesh/mesh::CoordinateMesh`
- `.../geometry/mesh::VelocityMesh/from_mesh/mesh_continuity` to
discontinuous
- `.../geometry/mesh::PressureMesh/from_mesh/mesh::CoordinateMesh`
- `.../geometry/mesh::PressureMesh/from_mesh/mesh_shape/polynomial_degree`
to 2

If `.../scalar_field::GeostrophicPressure` is to be included, cf. 8.8.1, 8.17.4.5, then a further mesh needs to be added. This must have polynomial degree one order greater than the mesh used for the velocity field. To demonstrate let us call this mesh 'GeostrophicPressureMesh', derive this

mesh from the `CoordinateMesh` and assume the mesh used for the velocity field has polynomial order 1, then to include this option:

- select a new `.../geometry/mesh` and set the name attribute as `GeostrophicPressureMesh`
- select `.../from_mesh::CoordinateMesh`
- set `.../from_mesh::CoordinateMesh/mesh_shape/polynomial_degree` to 2

8.17.3 Timestepping

2 non-linear iterations are recommended and can be specified by setting `.../timestepping/nonlinear_iterations` to 2.

8.17.4 Material/phase

8.17.4.1 Equation of state

For most problems the linear equation of state is appropriate, cf. 2.3.3. This is selected with `.../equation_of_state/fluids/linear` and other values such as the thermal contraction coefficient can be set in the options that appear below.

It is generally recommended to subtract out the hydrostatic pressure level from the equation of state by setting the option `.../equation_of_state/fluids/linear/subtract_out_hydrostatic_level`. This will allow increased accuracy for lower-order element pairs, cf. 3.8.

8.17.4.2 Pressure

The specified mesh should be `CoordinateMesh` for P1-P1 and `PressureMesh` for P1DG-P2.

- `.../spatial_discretisation/continuous_galerkin` is recommended for the spatial discretisation.
- `.../scheme/poisson_pressure_solution` can be chosen as either `never` or `only_first_timestep`, cf. 8.8.2.

Note, if the normal component of the velocity is imposed on all boundaries, then either `.../prognostic/reference_node` or `.../solver/remove_null_space` need to be set, cf. 8.8.3.

8.17.4.3 Velocity

The specified mesh should be `CoordinateMesh` for P1-P1 and `PressureMesh` for P1DG-P2.

The equation used to solve for Velocity is set under `.../equation`, cf. 3.5. For GFD problems that require the Boussinesq approximation selecting `.../equation::Boussinesq` will ensure that this correct formulation used, cf. 2.4.4, 2.4.4.1, 3.5.

A Crank-Nicolson temporal discretisation with a non-linear relaxation is recommended for the temporal discretisation cf. 3.3, 3.4. This is selected by setting:

- .../temporal_discretisation/theta to 0.5
- .../temporal_discretisation/relaxation to 0.5

If using P1-P1, the following are recommended for the spatial discretisation:

- .../spatial_discretisation/continuous_galerkin/stabilisation/no_stabilisation, cf. 3.2.1.3
- .../spatial_discretisation/continuous_galerkin/mass_terms/lump_mass_matrix, cf. 3.6.1.1
- .../spatial_discretisation/conservative_advection set to 0 (non-conservative), 8.7.1

If using P1DG-P2, the following are recommended for the spatial discretisation:

- .../spatial_discretisation/discontinuous_galerkin/viscosity_scheme/compact_discontinuous_galerkin, cf. 3.2.3.3
- .../spatial_discretisation/discontinuous_galerkin/advection_scheme/upwind, cf. 3.2.3.1
- .../spatial_discretisation/discontinuous_galerkin/advection_scheme/integrate_advection_by_parts/twice, cf. 3.2.3.1
- .../spatial_discretisation/conservative_advection set to 0 (non-conservative), 8.7.1

8.17.4.4 Advected scalar fields

The recommended options for scalar fields are considered for P1-P1 and P1DG-P2 separately.

P_1P_1

The mesh used should be the `CoordinateMesh` and the equation type `AdvectionDiffusion`, cf. 3.2.

For the spatial discretisation a control-volumes discretisation with a finite-element face value discretisation and Sweby limiter are recommended which are selected with the options, cf. 8.7.1.2:

- .../spatial_discretisation/control_volumes/face_value::FiniteElement
- .../prognostic/spatial_discretisation/control_volumes/face_value::FiniteElement/limit_face_value/limiter::Sweby

To help increase speed it is possible to store upwind elements so they do not have to be recalculated every time step (only after adapts). To do this activate the option:

- .../prognostic/spatial_discretisation/control_volumes/face_value::FiniteElement/limit_face_value/limiter::Sweby/project_upwind_value_from_point/store_upwind_elements

An Element Gradient diffusion scheme is also generally recommended, selected under `.../spatial_discretisation/control_volumes/diffusion_scheme::ElementGradient`

For the temporal discretisation a Crank-Nicolson scheme is recommended, with the control volume options of 3 advection iterations and limit theta, cf. 8.7.1.2. These are set with the options:

- `.../temporal_discretisation/theta` set to 0.5
- `.../temporal_discretisation/control_volumes/number_advection_iterations`
- `.../prognostic/temporal_discretisation/control_volumes/limit_theta`

$P_{1DG}P_2$

The mesh used should be the `VelocityMesh` and the equation type `AdvectionDiffusion`, cf. 3.2.

For the spatial discretisation a discontinuous-Galerkin discretisation is recommended with a Lax-Friedrichs advection scheme, velocity projected to continuous space, advection integrated by parts once and a compact-discontinuous-Galerkin diffusion scheme with a vertex-based slope limiter. These options are selected with:

- `.../spatial_discretisation/discontinuous_galerkin/advection_scheme/lax_friedrichs`, cf. 3.2.3.1
- `.../spatial_discretisation/discontinuous_galerkin/advection_scheme/project_velocity_to_continuous/mesh::CoordinateMesh`
- `.../spatial_discretisation/discontinuous_galerkin/advection_scheme/integrate_advection_by_parts/once`, cf. 3.2.3.1
- `.../spatial_discretisation/discontinuous_galerkin/diffusion_scheme/compact_discontinuous_galerkin`, cf. 3.2.3.3
- `.../spatial_discretisation/discontinuous_galerkin/slope_limiter::Vertex_Based`, cf. 3.2.3.2.

For the temporal discretisation a Crank-Nicolson scheme with subcycling is recommended. This can be set with:

- `.../temporal_discretisation/theta` set to 0.5
- `.../temporal_discretisation/discontinuous_galerkin/maximum_courant_number_per_subcycle` set to an appropriate value.

8.17.4.5 Geostrophic Pressure

If enabled a 'geopressure' solver is used, 8.8.1.

- The specified mesh should be the `GeostrophicPressureMesh`, cf. 8.17.2.
- The terms included in the right-hand side of the geopressure solver are selected under: `.../spatial_discretisation/geostrophic_pressure_option`.
- A reference node must be set as geopressure uses Neumann boundary conditions on all boundaries (cf. 8.8.3), for example: `.../reference_node::node_1`

8.18 Mesh adaptivity

The configuration on mesh adaptivity occurs in two places: under `mesh_adaptivity` where the overall adaptive settings are configured, and on a per-field basis where both the interpolation method is set and if that field should be considered when creating the error metric. See chapter 7 for the background to adaptivity and more detailed information.

8.18.1 Field settings

For each field present in the simulation there are up to two options that should be set. The first is the interpolation method that should be used to transfer the values of a field from the old to the new mesh, section 7.6. Second, in order to form the error metric by which the mesh is adapted, section 7.5.1, the user must set which fields should form the error metric and how the error for that field should be calculated.

8.18.1.1 Interpolation method

For each prognostic field in the current state, an interpolation type, section 7.6, must be set. These can be set by selecting an option `.../prognostic/<interpolation type>` where `<interpolation type>` is one of:

- Consistent interpolation - the default and quick interpolation method, but is non-conservative and dissipative.
- Pseudo-consistent interpolation - not recommended at present.
- Galerkin interpolation - Conservative and non-dissipative, but requires the construction of a supermesh [Farrell et al., 2009, Farrell and Maddison, 2010]
- Grandy interpolation - Conservative, but highly diffusive. See Grandy [1999].

For some fields, such as Pressure and Velocity other interpolation methods are available.

For diagnostic and prescribed fields an interpolation method is not required. However, if an output dump occurs immediately following a mesh adapt, diagnostic fields may not have correct values depending on the method by which they are calculated. In these instances, it is worth setting an interpolation type for these fields which will ensure that the values are set correctly before an output dump occurs.

The Galerkin projection also requires some further settings depending on the mesh type. For discontinuous meshes there are no other required settings. For continuous meshes a solver is required in order to perform the supermesh projection. The solver settings are configured as with any other solver, see section 8.9 for more details.

With piecewise linear continuous fields additional options are available to bound the result following a Galerkin projection:

```
.../galerkin_projection/continuous/bounded::Diffuse
```

and

```
.../galerkin_projection/continuous/bounded::Algencan
```

The latter uses the `algencan` optimisation library to bound the field and requires Fluidity to be configured with `--enable-algencan`. The `Diffuse` bounding algorithm is internal to Fluidity and the most frequently used.

To use the `Diffuse` bounding algorithm [Farrell et al., 2009] the number of iterations the algorithm is allowed to take must be specified. Additionally an optional tolerance can be specified to terminate

this iteration loop early. Furthermore if the bounds on the field are known in advance then these can be specified through:

```
.../bounded::Diffuse/bounds/upper_bound
and
```

```
.../bounded::Diffuse/bounds/lower_bound.
```

If the diffusion bounding algorithm fails to locally redistribute the unboundedness then a conservative but non-local redistribution can be activated using:

```
.../bounded::Diffuse/repair_deviations
```

again with an optional tolerance:

```
.../bounded::Diffuse/repair_deviations/tolerance.
```

8.18.1.2 Creating an error metric

The second step for configuring adaptivity is to set up the fields that are to form the error metric used to adapt the mesh. For each field that should be considered when forming the metric the option `.../adaptivity_options` needs to be enabled. The type or error norm on which the metric is based (absolute or relative) is set with `.../adaptivity_options/absolute_measure` or `.../adaptivity_options/relative_measure`. For a p -norm `.../adaptivity_options/absolute_measure` should be selected and the value of p set with the option `.../adaptivity_options/absolute_measure/p_norm` ($p = 2$ is recommended).

The `InterpolationErrorBound` field must be set and as with any other prescribed field can take a constant value or vary in space and time (by prescribing a python function for example). The error bound is set as separate fields within state, so for Temperature, for example, the acceptable error is stored in a field called `TemperatureInterpolationErrorBound`. This field is output as any other field too.

For relative interpolation error bounds a tolerance value also has to be set under

```
.../adaptivity_options/relative_measure/tolerance.
```

This value prevents division by zero and should be set to a small enough number that the field can effectively be considered zero at this value.

For discussion of the different metrics and error norms see section [7.5.1](#).

8.18.2 General adaptivity options

These are found under `mesh_adaptivity`. Here the user can specify whether to use mesh movement methods, prescribed adaptivity (serial only) or hr adaptivity. hr adaptivity is the normal method for most applications.

Under `/mesh_adaptivity/hr_adaptivity` there are a number of mandatory options, which are:

- `period` - how often should the mesh be adapted. This can be set in number of simulation seconds, or in number of timesteps. It is recommended that adapt happen every 10-20 timesteps.
- `maximum_number_of_nodes` - sets the maximum possible number of nodes in the domain. In parallel this is the global maximum number, but can be altered to be the number per process. If the maximum number of nodes is reached the mesh is coarsened everywhere until this is achieved.
- `enable_gradation` - is on by default and set to a value of 1.5. This constrains the jump in desired edge lengths along an edge and therefore controls how fast the mesh size may change.
- `tensor_field::MinimumEdgeLengths` - a tensor specifying the minimum edge length of an element.

- `tensor_field::MaximumEdgeLengths` - a tensor specifying the maximum edge length of an element.

In addition to these mandatory settings, there are a number of other configuration options.

- `cpu_period` - sets the time interval for the mesh adapt in cpu time.
- `minimum_number_of_nodes` - sets the minimum possible number of nodes in the domain. In parallel this is the global minimum number. The mesh is refined until this is achieved.
- `adaptive_timestep_at_adapt` - used in conjunction with adaptive timestep (see section 8.3.5.8), this option resets the timestep back to the minimum value under `.../adaptive_timestep/minimum_timestep` immediately following a mesh adapt.
- `maximum_node_increase` - the maximum ratio by which the number of nodes is allowed to increase. A value of 1.1 indicates the number of nodes can increase by at most 10%.
- `node_locking` - allows the locking of nodes via a python function that cannot be moved by adaptivity.
- `functional_tolerance` - specifies the minimum element functional value for which elements are considered for adaptivity, section 7.4. Default value is 0.15.
- `geometric_constraints` - this applies geometric constraints to the metric formation which aims to prevent the metric demanding edge length that are inappropriately large in comparison to the resolution required to preserve the geometric accuracy of the boundaries. If you get 'knife elements' near the boundaries try turning this option on. This only works in 3D.
- `bounding_box_factor` - this option bounds the edge lengths requested by the metric by bounding box of the domain, multiplied by the specified factor. The default value is 2.
- `reference_mesh` - supply a reference mesh which supplies the minimum or maximum edge length to the metric.
- `aspect_ratio_bound` - maximum aspect ration of elements in the adapted mesh.
- `adapt at first timestep` - perform mesh adaptivity before the first timestep occurs. This can occur a specified number of times.
- `preserve_mesh_regions` - ensures that regions in your mesh, specified by region IDs, are preserved through adaptivity. is adapted, then the mesh is extruded using the adaptivity metric in the 3rd dimension. You must use an extruded mesh with this option, section 6.4.
- `adaptivity_library` - choose which adaptivity library to use. In 2D you are restricted to `libmba2d`. In 3D you can choose either `libmba3D` or `libadaptivity` (default).
- `adapt_iterations` - this options controls the number of intermediate adapt iteration during parallel adaptive simulations, section 7.7. The default value is 3. Higher values may give you better meshes, especially when the number of elements per process is low.
- `debug` - options for output that is useful for debugging adaptivity.

8.18.2.1 Vertically structured and 2+1D adaptivity

For some problems it can be advantageous to apply adaptivity in the horizontal and vertical as separate steps. This means a horizontal (surface) mesh is adapted first after which a column of nodes is created under each surface node. The resolution in the vertical columns is either specified under

the extrusion options, or determined via a vertical adaptivity step. This functionality is switched on using the `vertically_structured_adaptivity` option. An extruded initial mesh is required for this option (see section 6.4, and section 8.4.2.4 for its configuration). The horizontal adaptivity stage is then applied to the horizontal input mesh, and the `bottom_depth` and `sizing_function` extrusion options are reapplied for the creation of the vertical columns. If an extruded initial mesh is provided *without* `vertically_structured_adaptivity`, the extruded mesh is simply adapted in all directions, resulting in a fully unstructured mesh and the extrusion options are no longer applicable. Further options under `vertically_structured_adaptivity` are:

- `inhomogenous_vertical_resolution` - This option switches on vertical adaptivity. This means it will no longer create layers based on the `sizing_function` option. Instead, the distance between the nodes in the vertical columns is based on the vertical component of the error metric. The vertical resolution will therefore vary over the depth and in each column independently. With the combination of `vertically_structured_adaptivity` and `inhomogenous_vertical_resolution`, adaptivity can thus focus resolution in both horizontal and vertical, while maintaining a columnar nodal structure. This combination is referred to as 2+1D adaptivity.
 - `adapt_in_vertical_only` - With this option vertical adaptivity is applied, but the horizontal mesh is kept fixed.
- `split_gradation` - Instead of applying gradation to the full metric before splitting into a horizontal and vertical metric, with this option the gradation is applied after the split. Thus in particular when specifying anisotropic gradation, the gradation in the horizontal and vertical is applied completely independently.
- `vertically_align_metric` or `use_full_metric` - The metric applied in the horizontal adaptivity stage is assembled by merging the 3D metric in each column and then projecting to the horizontal plane. Typically the 3D metric for large aspect ratio problems already decomposes in an (almost) vertical eigenvector and 2 horizontal ones. However, even the slightest tilt causes vertical error bounds to be “leaked” into the merged horizontal metric, leading to unexpected small horizontal edge lengths. Therefore for large aspect ratio problems the option `vertically_align_metric`, which decomposes the metric *before* merging in the horizontal, is recommended.
- `include_bottom_metric` - When constructing the horizontal metric incorporate the components of the full metric tangential to the bottom boundary. For example, this is useful when horizontal contours of a field intersect the bathymetry and this information is not automatically incorporated into the horizontal metric leading to the contact point being underresolved.

8.18.2.2 Zoltan options

There are a number of options available for controlling Zoltan’s behaviour when re-partitioning the mesh during and after adaptivity, which can be found under `mesh_adaptivity/zoltan_options`. The options are:

- `partitioner` - this is the partitioner used in the intermediate adapt iterations. It can be one of Scotch, ParMetis, Zoltan, or Zoltan Hypergraph. Default is Zoltan.
- `final_partitioner` - the partitioner used for the final adapt iteration where load balancing is important. Same choices as above. Default is ParMetis.
- `element_quality_cutoff` - at what value of element quality is an element deemed “bad”. Default is 0.6.

- load imbalance tolerance - a value of 1 means each processor will have exactly the same number of elements. However, smaller numbers here mean that the intermediate adapts may not be able to move the mesh sufficiently to get a good quality mesh from adaptivity.
- additional adapt iterations - increases the number of intermediate adapt iterations during parallel adaptive simulations.
- zoltan debug - debugging options.

For more information on the approach to parallel adaptivity adopted in Fluidity see section 7.7.

8.18.2.3 Metric advection

Metric advection advects the metric along with the flow, ensuring the resolution can be pushed ahead of any flow, rather than lagging behind, section 7.5.7. The advection equation is discretised with a control volume method, section 3.2.4.1. For spatial discretisation a first order upwind scheme for calculation the face values (the default) and non-conservative form are generally recommended. These are selected with options

- `/mesh_adaptivity/hr_adaptivity/metric_advection/spatial_discretisation/control_volumes/face_value::FirstOrderUpwind`
- `/mesh_adaptivity/hr_adaptivity/metric_advection/spatial_discretisation/conservative_advection = 0.0`

For temporal discretisation a semi-implicit discretisation in time is recommended, section 3.4, with option

- `/mesh_adaptivity/hr_adaptivity/metric_advection/temporal_discretisation/theta = 0.5`

The time step is controlled by the choice of CFL number, specified in `/mesh_adaptivity/hr_adaptivity/metric_advection/temporal_discretisation/maximum_courant_number_per_subcycle`. The metric is advected over the time period between the current and the next adapt. This time period can be scaled with the option `/mesh_adaptivity/hr_adaptivity/metric_advection/temporal_discretisation/scale_advection_time` which has a default value of 1.1.

8.19 Multiple material/phase models

This section contains advice on setting up simulations with multiple `material_phase` options. This enables related fields to be grouped together into related materials or phases. For example a prognostic scalar field in one `material_phase` will be advected using the Velocity field from that `material_phase`, while a prognostic scalar field in another `material_phase` will be advected according to the Velocity field in its `material_phase`.

We refer to two typical scenarios: a *multiple material* model and a *multiple phase* model. A multiple phase model is one in which the Velocity field in each `material_phase` is in some way independent of the velocities in the other `material_phases`. This means that scalar fields (for example phase volume fractions) in each `material_phase` are advected independently. A multiple material model is one in which the Velocity field is shared between all `material_phases` so that all scalar fields are advected similarly.

8.19.1 Multiple material models

Models with a single prognostic velocity field that is shared between material_phases (using the `aliased` field type) are referred to as multiple material models. These are generally used to describe systems of nearly immiscible materials with different material properties contained within the same domain. This section focusses on this type of multiple material_phase simulation.

In a multiple material simulation each material_phase requires:

- an equation of state, and
- a `MaterialVolumeFraction` scalar field.

The equation of state provides the density properties of the material described in the current material_phase. For incompressible simulations a linear equation of state is used, which only requires a reference density:

```
.../equation_of_state/fluids/linear/reference_density
```

to be set. For Boussinesq multimaterial simulations, where a material's density depends on temperature and/or salinity, then the same dependencies exist between the equation of state and these fields as in single material simulations. For example a `Temperature` field must be present in the material_phase where it is needed (although it may be aliased between material_phases). If the `subtract_out_hydrostatic_level` option is selected, it must only be select in a single material_phase. Fully compressible multimaterial simulations are not supported.

The `MaterialVolumeFraction` field describes the location of the material, varying from 1 in regions where the cells are entirely the current material to 0 where none of this material is present. As the materials are generally treated as being nearly immiscible, the prognostic `MaterialVolumeFraction` field should be discretised using a control volume spatial discretisation with one of the face value schemes designed for advecting step functions:

- `.../spatial_discretisation/control_volumes/face_value::HyperC`
- `.../spatial_discretisation/control_volumes/face_value::UltraC`
- `.../spatial_discretisation/control_volumes/face_value::PotentialUltraC`

as described in sections 3.2.4.1–3.2.4.1. These schemes are only guaranteed to be bounded for explicit advection so the implicitness factor, θ :

```
.../temporal_discretisation/theta
```

and the pivot implicitness factor, θ_p :

```
.../temporal_discretisation/control_volumes/pivot_theta
```

should be set to zero and, for high Courant number flows, advection subcycling should be used:

```
.../temporal_discretisation/control_volumes/maximum_courant_number_per_subcycle
```

or

```
.../temporal_discretisation/control_volumes/number_advection_subcycles.
```

For an N material problem, N material_phases are required and hence N `MaterialVolumeFractions`, $c^i, i = 1, \dots, N$, and N equations of state. However, only $N - 1$ of the `MaterialVolumeFraction` fields, $c^i, i = 1, \dots, N - 1$, need be prognostic. The final volume fraction field, c^N , should always be set to diagnostic, as it can be recovered using the internal algorithm:

$$c^N = 1 - \sum_{i=1}^{N-1} c^i. \quad (8.3)$$

So, for example, in the case when $N = 2$ there need only be a single prognostic `MaterialVolumeFraction` field and a single diagnostic `MaterialVolumeFraction` field. In this case it makes no difference

which material phase contains the prognostic volume fraction and which contains the diagnostic field. In more complicated scenarios with $N > 2$ a coupled control volume discretisation (see section 3.2.4.1) becomes necessary to ensure that not only each of the $N - 1$ prognostic MaterialVolumeFractions remain bounded but also that their sum, $\sum_{i=1}^{N-1} c^i$, is bounded. This ensures, through equation 8.3, that the final diagnostic MaterialVolumeFraction is also bounded. As discussed in section 3.2.4.1 this process requires a priority ordering for the fields, which must be specified at:

```
.../scalar_field::MaterialVolumeFraction/prognostic/priority.
```

The diagnostic field is always treated as the lowest priority volume fraction so in this case the choice of priority ordering and diagnostic field may affect the results if the interfaces between the materials are in the vicinity of one another. Priority ordering and coupled limiting do not affect the advection process if the material interfaces are separated from each other.

If adaptive remeshing is being used then the bounded and minimally dissipative behaviour of the above advection must be preserved through the interpolation between successive meshes. As discussed in section 8.18 several interpolation algorithms are available. We discuss them again here in terms of their suitability for multiple material modelling. Consistent interpolation on piecewise linear parent meshes guarantees boundedness of the interpolated volume fraction field and of the sum of the volume fractions. However it tends to introduce excessive amounts of numerical diffusion and it is not conservative. Galerkin projection guarantees conservation of the field and is not excessively dissipative. However it does not guarantee boundedness.

To ensure minimal dissipation, conservation and boundedness it is necessary to use a bounding algorithm following the Galerkin projection. The Diffuse bounding algorithm (see section 8.18.1.1) is generally used. This redistributes unbounded values in the field locally, guaranteeing boundedness of each volume fraction individually. It does not however guarantee boundedness of the sum of the volume fractions and this must be enforced by coupling each MaterialVolumeFraction field together through the interpolation with the option:

```
.../bounded::Diffuse/bounds/upper_bound/coupled
```

under all $N - 1$ prognostic MaterialVolumeFractions. As with coupled control volume advection this uses the priority numbering of the fields to determine the order in which they are bounded. The local bounds enforced on successive fields are then modified to ensure boundedness of their sum. This redistribution of materials during the bounding procedure introduces some relative movement between materials, which, by equation 8.3, is filled in by the diagnostic MaterialVolumeFraction. Despite this problem bounded Galerkin projection is recommended to transfer field data during mesh adaptivity.

The advected (and interpolated) volume fractions describe volume averaged the locations of the materials. In combination with the equation of state they can therefore be used to define global bulk values for the density using:

$$\rho = \sum_{i=1}^N \rho^i c^i \quad (8.4)$$

where ρ is the bulk density and ρ^i are the individual material densities, given by their respective equations of state. This bulk density can be seen in the diagnostic Density field in whichever material_phase has the prognostic Velocity field in it.

In addition to the density the volume fractions may be used to specify a bulk Viscosity that varies between the materials according to:

$$\bar{\mu} = \sum_{i=1}^N \bar{\mu}^i c^i \quad (8.5)$$

where $\bar{\mu}$ is the bulk viscosity and $\bar{\mu}^i$ is the individual material's viscosity. To use this it is necessary to activate a MaterialViscosity field in every material_phase with a nonzero viscosity. Additionally the Viscosity field underneath the prognostic Velocity must be activated and set to the bulk_viscosity diagnostic algorithm.

8.19.2 Multiple phase models

Models with one prognostic velocity field per `material_phase` are referred to as multiple phase models. The use of these multiple velocity fields permits the inter-penetration and interaction between different phases.

8.19.2.1 Simulation requirements

In a multi-phase simulation, each `material_phase` requires:

- an equation of state, and
- a `PhaseVolumeFraction` scalar field.

As per multi-material simulations, the equation of state provides the density properties of the phase described in the current `material_phase`. For incompressible simulations a linear equation of state is used, which only requires a reference density:

`.../equation_of_state/fluids/linear/reference_density`
to be set.

For an N phase problem, N `material_phases` are required and hence N `PhaseVolumeFractions`, $\alpha_i, i = 1, \dots, N$, and N equations of state. Just as in multi-material simulations, only $N - 1$ of the `PhaseVolumeFraction` fields, $\alpha_i, i = 1, \dots, N - 1$, need be prognostic. The final `PhaseVolumeFraction` field, α_N , should always be set to `diagnostic`, as it can be recovered using the internal algorithm:

$$\alpha_N = 1 - \sum_{i=1}^{N-1} \alpha_i. \quad (8.6)$$

8.19.2.2 Inter-phase interactions

Inter-phase interactions can be included under `../multiphase_properties` in `Diamond`. Currently, `Fluidity` only supports fluid-particle drag between the continuous phase and dispersed phase(s), given by:

$$\mathbf{F}_D = \frac{3 \alpha_p C_D \alpha_f \rho_f |\mathbf{u}_f - \mathbf{u}_p| (\mathbf{u}_f - \mathbf{u}_p)}{4 d}, \quad (8.7)$$

where f and p denote the fluid (i.e. continuous) and particle (i.e. dispersed) phases respectively, and d is the diameter of a single particle in the dispersed phase. The drag coefficient C_D is defined as:

$$C_D = \frac{24}{\text{Re}}, \quad (8.8)$$

with

$$\text{Re} = \frac{\alpha_f \rho_f d |\mathbf{u}_f - \mathbf{u}_p|}{\mu_f}. \quad (8.9)$$

Note that μ_f denotes the isotropic viscosity of the fluid (i.e. continuous) phase.

Within a dispersed phase, entering a value for d in the `../multiphase_properties/particle_diameter` option enables fluid-particle interaction between itself and the continuous phase.

8.19.2.3 Current limitations

- Boussinesq and fully compressible multi-phase simulations are not yet supported.
- The momentum equation for each `material_phase` can only be discretised in non-conservative form.
- The stress term must be in tensor form, and only works with an isotropic viscosity.
- `bassi_rebay` and `compact_discontinuous_galerkin` are currently the only DG viscosity schemes available for multi-phase flow simulations.
- Discontinuous `PhaseVolumeFraction` fields are not yet supported.
- The Pressure field only supports `continuous_galerkin` and `control_volume` discretisations.
- Prescribed velocity fields cannot yet be used in multi-phase simulations.
- Fluid-particle drag can currently only support one continuous (i.e. fluid) phase.

8.20 Compressible fluid model

Enabling `.../material_phase/equation_of_state/compressible` allows the compressible equations described in sections 2.3.2.1 and 2.3.2.2 to be solved. At the moment there is one available option for the required compressible equation of state: Mie-Grunneisen (see section 8.10.0.5). Compressible functionality is not yet fully supported and this is intended as a stub upon which further developments will be described.

This section contains advice for running a compressible simulation, by describing the necessary options to set up the problem. The options required for prognostic fields are:

8.20.1 Pressure options

- The value for the atmospheric pressure can be added by switching on `.../atmospheric_pressure`, otherwise a default of zero is used.
- A Poisson pressure equation should not be used to calculate a first guess, therefore `.../scheme/poisson_pressure_solution` should be set to *never*.
- `.../scheme/use_compressible_projection_method` should be selected, so the calculated pressure satisfies the continuity equation and the EOS.

8.20.2 Density options

`.../prognostic/equation` and `.../prognostic/solver` do not need to be enabled. If the equation type is not turned on, the density will make use of the pressure solve, so no solver options are needed either. By having an equation type turned on, the density is not only incorporated into the pressure solve but also an Advection-Diffusion equation is solved (and solver options need to be specified).

8.20.3 Velocity options

- for continuous velocities `.../spatial_discretisation/.../lump_mass_matrix` should be turned on.
- in the presence of viscosity `.../spatial_discretisation/continuous_galerkin/stress_terms/stress_form` is required and all components of the anisotropic symmetric Viscosity tensor should be filled out. This functionality is only available with continuous Galerkin velocities.

8.20.4 Restrictions: discretisation options and element pairs

Either continuous Galerkin or control volumes can be used as discretisation options for pressure and density (both fields need to have the same option). When using control volumes pressure and density have to be on the same order of parent mesh.

Chapter 9

Visualisation and Diagnostics

9.1 Visualisation

Output files containing simulation data are managed using the Visualization Toolkit (VTK - please refer to the website <http://www.vtk.org/>). VTK tools adopts the .vtu file format for unstructured grids: for each N dump times, the `simulationname_N.vtu` file is created, containing the output data of the simulation. This file contains a snapshot of the run at the timestep immediately proceeding the dump time. For example, if the timestep is set to three seconds, and the dump period to 10 seconds, the first dump will occur at 12 seconds.

When running a simulation in parallel, the data are stored both in both .vtu and .pvtu files. The .vtu files contain each the output data for each partition of the parallelised mesh, with the filename `simulationName_P.N.vtu` where P is the processor number and N is the dump number. The .pvtu files contain the general output data for the whole mesh, with numbering is still ordered by dump number.

Visualisation of the .vtu and .pvtu files can be done using paraview (<http://www.paraview.org/paraviewindex.html>) and/or mayavi (<http://mayavi.sourceforge.net/>). See the [AMCG website](#) for more information.

9.2 Online diagnostics

9.2.1 Fields

9.2.1.1 Internal diagnostic fields

Fluidity has a set of predefined diagnostic fields called internal diagnostic fields. These diagnostic fields can be scalar, vector and tensor fields. Common used internal diagnostic field are the CFLNumber or the Gradient of a specified `scalar_field`.

Each internal diagnostic field has a unique identifier and are classified by their field type: `scalar_field`, `vector_field` or `tensor_field`. To configure a internal diagnostic field, add a new field of the appropriate type and select the identifier. A description of the available diagnostic fields is given below. For example, to add the CFLNumber (which is a `scalar_field`), one would add a new `scalar_field` and select `scalar_field (CFLNumber)`.

Some internal diagnostics contain a `.../diagnostic/fieldname` attribute defining the field used to compute the diagnostic (for example the field used to compute a gradient). The internal diagnostics do not have a dependency resolution, that is if this source field is itself a diagnostic field

it may happen that the source field is not computed yet. In such a case, one should try and use diagnostic algorithms instead, see next section.

In the following, a description of the internal diagnostics available in Fluidity is given.

Internal `scalar_field` diagnostics:

AbsoluteDifference: Absolute Difference between two scalar fields. Both fields and this diagnostic `scalar_field` must be in the same `material_phase`. It also assumes both fields are on the same mesh as the `AbsoluteDifference` field.

BackgroundPotentialEnergyDensity: Background potential energy density: $PE_b = \rho z_{star}$ where ρ is the density, z_{star} is the isopycnal coordinate (which is calculated in the diagnostic `scalar_field::IsopycnalCoordinate`).

Limitations:

- Requires a constant gravity direction.
- The `Density` and `GravitationalPotentialEnergyDensity` fields must be on the same mesh.

Limitations: Requires the diagnostic `scalar_field::IsopycnalCoordinate` and (therefore) is not parallelised.

BulkMaterialPressure: Calculates the bulk material pressure based on the `MaterialDensity` and `MaterialVolumeFraction` (and `MaterialInternalEnergy` if appropriate) for the equation of state of all materials.

CFLNumber: CFLNumber as defined on the co-ordinate mesh. It is calculated as $\Delta t \mathbf{u} J^{-1}$ where Δt is the timestep, \mathbf{u} the velocity and J the Jacobian.

ControlVolumeCFLNumber: CFL Number as defined on a control volume mesh. It is calculated as $\Delta t \frac{1}{V} \int_{cv_{face}} \mathbf{u}$ where Δt is the timestep and \mathbf{u} the velocity. The integral is taken over the faces of the control volume and V is the volume of the control volume.

ControlVolumeDivergence: Divergence of the velocity field where the divergence operator is defined using the control volume C^T matrix. This assumes that the test space is discontinuous control volumes.

CVMaterialDensityCFLNumber: Courant Number as defined on a control volume mesh and incorporating the `MaterialDensity`. Requires a `MaterialDensity` field!

DG.CourantNumber: CFLNumber as defined on a DG mesh. It is calculated as $\Delta t \frac{1}{V} \int_{element} \mathbf{u}$ where Δt is the timestep and \mathbf{u} the velocity. The integral is taken over the faces of the element V is the volume of the element.

DiffusiveDissipation: The rate at which internal energy is converted to potential energy: $-g \frac{\partial \rho}{\partial y}$, where ρ is the Density. Note the actual diffusive dissipation is $-2g \bar{\kappa} \frac{\partial \rho}{\partial y}$ (2 subject to definition) where $\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho = \bar{\kappa} \nabla^2 \rho$. This should be taken into account when post-processing. It also assumes kappa is isotropic and constant, cf. [Winters et al. \[1995\]](#).

DiscontinuityDetector: This field detects the discontinuities in a discontinuous Galerkin field, the larger the discontinuity, the larger the value it takes. The discontinuity field is used by the HWENO slope limiter, [3.2.3.2](#).

FiniteElementDivergence: Divergence of the velocity field where the divergence operator is defined using the finite element C^T matrix.

FreeSurface: Computes the free surface. Note: the diagnostic `FreeSurface` field only works in combination with the `free_surface` boundary condition applied to the `Velocity` field. It gives you a 3D field (constant over the vertical) of the free surface elevation.

FreeSurfaceHistory: The free surface history diagnostics saves snapshots of the free surface field. The regularity and amount of snapshots can be specified in `diamond`. The main usage of this diagnostic is for harmonic analysis, see 9.2.1.2

FunctionalBegin: Add a field to be used by `Explicit_ALE` to visualise functional values before iterations start.

FunctionalIter: Add a field to be used by `Explicit_ALE` to visualise functional values at each iteration.

GalerkinProjection: Galerkin projection of one field onto another mesh. The field must be in the same `material_phase` as this diagnostic `scalar_field`.

GravitationalPotentialEnergyDensity: Gravitational potential energy density: $\rho(g \cdot (r - r_0))$ where ρ is the density (taken from `scalar_field::Density`), and r_0 is the potential energy zero point.

Limitations:

- Requires a constant gravity direction.
- The `Density` and `GravitationalPotentialEnergyDensity` fields must be on the same mesh.

GridPecletNumber: The GridPecletNumber: $Pe = U\Delta x/\kappa$, where κ is the diffusivity. It is calculated as $\bar{\kappa}^{-1}\mathbf{u}J$ where $\bar{\kappa}$ is the diffusivity tensor, \mathbf{u} the velocity and J the Jacobian.

GridReynoldsNumber: Grid Reynolds number: $Re = U\Delta x/\nu$, where ν is the viscosity. It is calculated as $\bar{\nu}^{-1}\mathbf{u}J$ where $\bar{\nu}$ is the viscosity tensor, \mathbf{u} the velocity and J the Jacobian.

HorizontalStreamFunction: Calculate the horizontal stream function `psi` where: $\frac{\partial\psi}{\partial} = -v$ and $\frac{\partial\psi}{\partial y} = u$ where u and v are the velocity components perpendicular to the gravity direction. A strong Dirichlet boundary condition of 0 is applied on all boundaries.

HorizontalVelocityDivergence: Horizontal velocity divergence: $div_H\mathbf{u}$. The horizontal plane is determined from the gravity field direction.

IsopycnalCoordinate: Isopycnal coordinate $z_{star}(x,t) = \frac{1}{A} \int_{V'} H(\rho(x',t) - \rho(x,t))dV'$ where ρ is the density, A is the width/area of the domain.

Limitations:

- You need to specify a (fine) mesh to redistribute the `Density` onto.
- Requires a constant gravity direction.
- The `Density` and `GravitationalPotentialEnergyDensity` fields must be on the same mesh.
- Not parallelised

KineticEnergyDensity: Kinetic energy density: $\frac{1}{2}\rho|\mathbf{u}|^2$ where ρ is the density taken from the `scalar_field::Density`.

Limitations: The `Density`, `KineticEnergyDensity` and `Velocity` fields must be on the same mesh.

MaterialDensity: The density of the material in multimaterial simulations. Required in compressible multimaterial simulations. Can be `diagnostic` if using a linear equation of state, or `prognostic` if a compressible simulation. (Note that if you set a multimaterial equation of state and this field is prognostic then its initial condition will be overwritten by the density that satisfies the initial pressure and the equation of state).

MaterialEOSDensity: Calculates the material density based on the bulk `Pressure` (and `MaterialInternalEnergy` if appropriate) for the equation of state of this material.

MaterialMass: Add a `MaterialMass` `scalar_field` to calculate the spatially varying mass of a material.

- MaterialPressure:** Calculates the material pressure based on the MaterialDensity (and MaterialInternalEnergy if appropriate) for the equation of state of this material.
- MaterialVolume:** Add a MaterialVolume scalar field to calculate the spatially varying volume of a material.
- MaterialVolumeFraction:** Volume fraction c^N of material N in multimaterial simulations. Required in compressible multimaterial simulations. If `diagnostic`, this computes $c^N = 1 - \sum_{i=1}^{N-1} c^i$.
- MaxEdgeWeightOnNodes:** An estimate of the edge weights whilst adapting using Zoltan.
- MultiplyConnectedStreamFunction:** Calculate the stream function of 2D incompressible flow for multiply connected domains. Note that this only makes sense for proper 2D (not pseudo-2D) simulations. Requires a continuous mesh.
- NodeOwner:** Output the processors which own the nodes of the mesh on which this field is based.
- PhytoplanktonGrazing:** Grazing rate of Phytoplankton by Zooplankton. This is calculated by the ocean biology module and will not be calculated unless ocean biology is being simulated. See section 5.1 for more details.
- PerturbationDensity:** Calculates the perturbation of the density from the reference density.
- PhaseVolumeFraction:** Volume fraction α^N of phase N in multimaterial simulations. Required in multiphase simulations. If `diagnostic`, this computes $\alpha^N = 1 - \sum_{i=1}^{N-1} \alpha^i$.
- PotentialVorticity:** Ertel potential vorticity: $(\mathbf{f} + \nabla \times \mathbf{u}) \cdot \nabla \rho'$ where \mathbf{f} is the magnitude of the Coriolis force, \mathbf{u} the velocity and ρ' the perturbation density as calculated in `.../scalar_field::PerturbationDensity`. Limitations: Requires a geometry dimension of 3.
- PrimaryProduction:** Primary production rate of Phytoplankton. This is calculated by the ocean biology module and will not be calculated unless ocean biology is being simulated. See section 5.1 for more details.
- RelativePotentialVorticity:** Relative potential vorticity: $\nabla \times \mathbf{u} \cdot \nabla \rho'$, where \mathbf{u} is the velocity and ρ' the perturbation density as calculated in `.../scalar_field::PerturbationDensity`
- RichardsonNumber:** Returns the Richardson number: $\frac{N^2}{(\frac{\partial u}{\partial z})^2 + (\frac{\partial v}{\partial z})^2}$ where $N^2 = \frac{g}{\rho_0} \frac{\partial \rho}{\partial z}$ is the buoyancy frequency, z is the vertical direction, g is the magnitude of gravity (u, v) is the horizontal velocity, ρ_0 is the reference density and ρ' the perturbation density. (In 2D $z \rightarrow y$ and $\frac{\partial v}{\partial z}$ is not included).
- ScalarAbsoluteDifference:** Absolute Difference between two scalar fields. Both fields and this diagnostic `scalar_field` must be in the same `material_phase`. Assumes both fields are on the same mesh as the `AbsoluteDifference` field.
- Speed:** Speed: $|\mathbf{u}|$ Limitations: The Speed and Velocity fields must be on the same mesh.
- StreamFunction:** Calculate the stream function of 2D incompressible flow. Note that this only makes sense for proper 2D (not pseudo-2D) simulations. Requires a continuous mesh.
- SumMaterialVolumeFractions:** Sums up the prognostic MaterialVolumeFraction fields (i.e. computes $\sum_{i=1}^{N-1} c^i$, where N is the current material and c^i is the MaterialVolumeFraction of material i)
- SumVelocityDivergence:** Sums up the divergence of each phase's apparent velocity, i.e. $\sum_{i=1}^N \nabla \cdot (\alpha_i \mathbf{u}_i)$. Used in multiphase simulations.
- UniversalNumber:** Output the universal numbering of the mesh on which this field is based.

VelocityDivergence: Velocity divergence: $\text{div } \mathbf{u}$

ViscousDissipation: $\nabla \mathbf{u} : \nabla \mathbf{u} = \sum_{ij} \frac{\partial u_i}{\partial x_j} \frac{\partial u_i}{\partial x_j}$. The actual viscous dissipation for a Boussinesq fluid with isotropic viscosity, ν , is $\nu \rho_0 (\nabla \mathbf{u}) : (\nabla \mathbf{u})$ where ρ_0 is the reference density in the equation of state. This should be taken into account when post-processing, cf. [Winters et al. \[1995\]](#).
Limitations: Only coded for 2D.

CopyofDensity: This scalar field is meant to replace DENTRAF. Basically, if you use new options, DENTRAF is no longer needed. No repointing is done from this field to DENTRAF.

ParticleScalar: Add a field to be used by Solid_configuration to map the solid_Concentration from particle mesh to the fluid mesh.

SolidConcentration: The volume fraction of the solid phase in FEMDEM.

SolidPhase: Zero everywhere except for the boundary of the solid phase in FEMDEM.

VisualizeSolid: Add a field to be used by Solid_configuration to visualise the solid_Concentration

VisualizeSolidFluid: Add a field to be used by Solid_configuration to visualise the solids and MaterialVolumeFraction together.

WettingDryingAlpha: Wetting and drying alpha coefficient. Alpha is 1 in dry and 0 in wet regions. Note: the diagnostic WettingDryingAlpha only works in combination with the free_surface boundary condition applied to the Velocity field. It gives you a 3D field (constant over the vertical) of the wetting and drying alpha coefficient.

The available internal `vector_field` diagnostics are:

AbsoluteDifference: Absolute Difference between two vector fields. Both fields and `vector_field::AbsoluteDifference` must be in the same material_phase. Assumes both fields are on the same mesh as `vector_field::AbsoluteDifference`.

AbsoluteVorticity: Absolute vorticity: $f \mathbf{k} + \nabla \times \mathbf{u}$, where f is the magnitude of the Coriolis force and \mathbf{u} the velocity. Limitations: Requires a geometry dimension of 3.

Buoyancy: Computes the buoyancy term $\mathbf{b} = -\rho \mathbf{g}$.

BedShearStress: Returns the (vector) bed shear stress, $b_{ss} = \rho C_D |\mathbf{u}| \mathbf{u}$, with ρ the density, C_D the drag coefficient and \mathbf{u} the velocity. The density and drag coefficients have to be given and are assumed to be constant. The field is only calculated over surface elements/nodes and interior nodes will have zero value.

ControlVolumeDivergenceTransposed: Gradient of a scalar field evaluated using the transpose of the C^T matrix constructed using control volumes. The related field must be in the same material_phase as `vector_field::ControlVolumeDivergenceTransposed`

Coriolis: Projects the Coriolis term onto the mesh of this diagnostic field. lump mass matrix?

DiagnosticCoordinate: Coordinate field remapped to the specified mesh.

DgMappedVelocity: The continuous solution mapped to a discontinuous mesh. Limitations: Requires a geometry dimension of 3. Requires inner element active for momentum.

DgMappedVorticity: Vorticity of the DG mapped Velocity. Note vorticity is actually calculated over a DG field. Limitations: Requires a geometry dimension of 3. Requires inner element active for momentum.

ElectricalConductivity: A spontaneous potentials diagnostic to compute electrical conductivity.

- FiniteElementDivergenceTransposed:** Gradient of a scalar field evaluated using the transpose of the C^T divergence matrix constructed using finite elements. The field must be in the same material_phase as `vector_field::FiniteElementDivergenceTransposed`.
- FiniteElementGradient:** Gradient of a scalar field evaluated using the C gradient matrix constructed using finite elements. The field must be in the same material_phase as `vector_field::FiniteElementGradient`.
- FunctionalGradient:** Same as `vector_field::SolidVelocity` but it is on the Particle mesh. It is used to map the velocities coming from an external program like FEMDEM or DEM to the fluid mesh.
- GalerkinProjection:** Galerkin projection of one field onto another mesh. The field must be in the same material_phase as `vector_field::GalerkinProjection`
- InnerElementFullVelocity:** Full velocity in an inner element SGS treatment of momentum. Limitations: Requires a geometry dimension of 3. Requires inner element active for momentum.
- InnerElementFullVorticity:** Vorticity of the full velocity in an inner element SGS treatment of momentum. Limitations: Requires a geometry dimension of 3. Requires inner element active for momentum.
- InnerElementVorticity:** Vorticity of the SGS velocity in an inner element SGS treatment of momentum. Limitations: Requires a geometry dimension of 3. Requires inner element active for momentum.
- LinearMomentum:** LinearMomentum field: $p = \rho u$ (where p is the linear momentum, ρ the density and u the velocity)
- MaxBedShearStress:** Max Bed Shear Stress. Note that you need `vector_field::BedShearStress` turned on for this to work.
- ParticleForce:** Same as Solid Velocity field but it is on the Particle mesh. It is used to map the velocities coming from an external program like FEMDEM or DEM to the fluid mesh.
- ParticleVector:** Same as Solid Velocity field but it is on the Particle mesh. It is used to map the velocities coming from an external program like FEMDEM or DEM to the fluid mesh.
- PlanetaryVorticity:** Planetary vorticity. Limitations: Requires geometry dimension of 3.
- SolidForce:** Same as Solid Velocity field but it is on the Particle mesh. It is used to map the velocities coming from an external program like FEMDEM or DEM to the fluid mesh.
- SolidVelocity:** Solid Velocity field. Used to generate the momentum source
- TemperatureGradient:** Temperature gradient
- VectorAbsoluteDifference:** Absolute Difference between two vector fields. Assumes both fields are on the same mesh as `vector_field::AbsoluteDifference`. Both fields and `vector_field::AbsoluteDifference` must be in the same material_phase.
- VelocityPlotForSolids:** Implicit solids related field for the velocity of the solid phase.
- Vorticity:** (Relative vorticity field) - (curl of the velocity field)

9.2.1.2 Diagnostic algorithms

The name of each field for each material / phase in the options tree must be unique. Hence there can only be one field named “Gradient” in a single material / phase. The concept of a diagnostic algorithm is designed to solve this issue - multiple fields, each with their own name, can share the same algorithm. This, for example, allows the gradient of multiple fields to be calculated for a single material / phase.

To configure a diagnostic field using a diagnostic algorithm, select the `.../diagnostic` option for a generic `scalar_field`, `vector_field` or `tensor_field`. This contains a `.../diagnostic/algorithm` choice element from which you can select the diagnostic algorithm.

Some diagnostic algorithms contain a `.../diagnostic/algorithm/source_field` attribute defining the field used to compute the diagnostic (for example the field used to compute a gradient). If this source field is itself a diagnostic field defined in terms of a diagnostic algorithm then the source field is computed first (dependency resolution). In the majority of cases, a `scalar_source_field`, `vector_source_field`, `tensor_source_field` or `component_source_field` attribute is defined. This identifies the expected type of input field. `component_source_field` denotes scalar field input, but for which vector or tensor field components of the form `field_name%comp` can be used. The attribute `.../diagnostic/algorithm/material_phase_support`, which may take the value “single” or “multiple”, defines if the diagnostic algorithm may access fields in other material / phases. For multiple `material_phase_support` diagnostic fields, a source field in another material / phase may be defined by a `::` delimited “state_name :field_name” string.

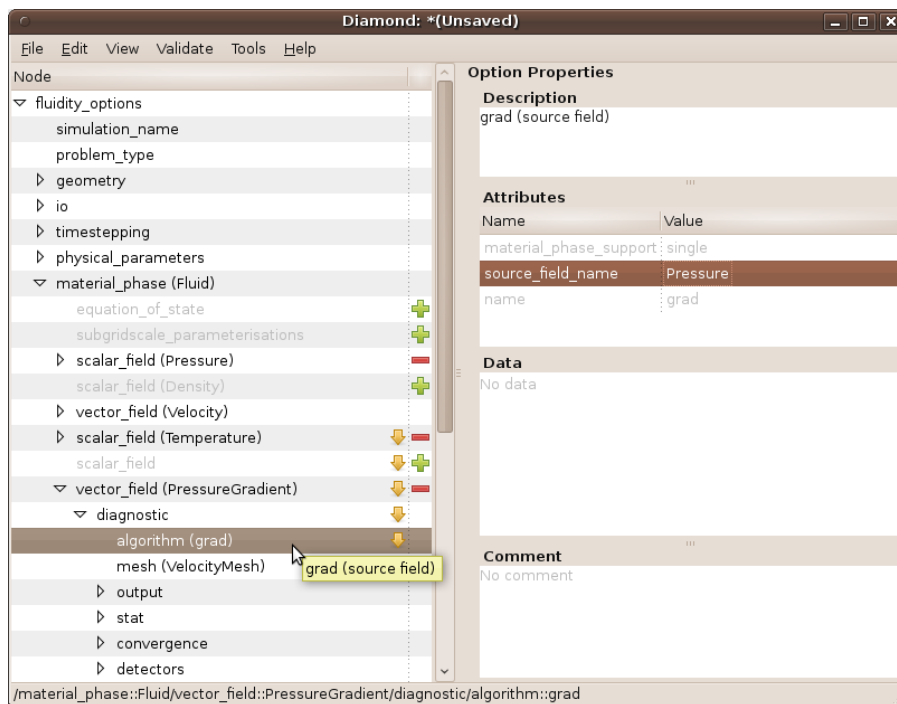


Figure 9.1: Configuration of a diagnostic field using a diagnostic algorithm in Diamond. Here a pressure gradient diagnostic is defined.

The available internal diagnostics are:

temporalmin: Writes the (nodewise) minimum scalar value over all previous timesteps.

temporalmax: Writes the (nodewise) maximum scalar value over all previous timesteps.

l2norm: Calculates nodewise l2norm of a vector field source.

time_averaged_scalar: Calculates the time average of a scalar field over the duration of a simulation.

- period_averaged_scalar:** Calculates the time average of a scalar field over a defined period, e.g. daily.
- time_averaged_scalar_squared:** Calculates the time average of squared scalar fields.
- free_surface_history:** Records the history of a free surface field.
- tidal_harmonics:** Calculates the (tidal) harmonics of the FreeSurface field. Note that "ocean boundaries" (see 8.3.3.5) have to be set and the FreeSurface and FreeSurfaceHistory diagnostic field have to be switched on (see 9.2.1.1).
- div:** Computes the divergence of a vector field.
- grad:** Computes the gradient of a scalar field.
- finite_element_divergence:** Computes the divergence of a field, where the divergence operator is defined using the finite element C^T matrix.
- curl_2d:** [curl (source field)]_z. Valid in 2D only.
- scalar_advection:** Computes $\mathbf{u} \cdot \nabla(s)$, where s is the source field.
- scalar_laplacian:** Computes $\nabla^2(s)$, where s is the source field. Applies natural boundary conditions on all boundaries.
- tensor_second_invariant:** Algorithm for the second invariant of a tensor field.
- scalar_potential:** Computes the scalar potential φ where: $F = \nabla\varphi + \nabla A + H$ with a Neumann boundary condition of $\nabla\varphi \cdot \mathbf{n} = F \cdot \mathbf{n}$ on all boundaries.
- projection_scalar_potential:** Computes the scalar potential φ where: $F = \nabla\varphi + \nabla A + H$ using a pressure projection method.
- node_halo:** Parallel diagnostic for the nodal halos.
- universal_numbering:** Writes the nodal universal numbering.
- element_halo:** Paints the element halo. Requires a degree 0 mesh.
- element_ownership:** Writes the element halo ownership. Requires a degree 0 mesh.
- element_universal_numbering:** Writes the element halo universal numbering. Requires a degree 0 mesh.
- scalar_sum:** Computes the sum of two scalar fields.
- scalar_difference:** Computes the difference between two scalar fields (i.e. field1 - field2)
- scalar_edge_lengths:** Computes the edge lengths of the Coordinate mesh.
- grad_normal:** Computes $\int (\nabla s) d\mathbf{n}$. The output is P0 on the surface. Errors will occur at domain edges - this is a limitation of current output formats.
- column_ids:** Convert the column_ids to a field. The mesh must be directly extruded.
- universal_column_ids:** Convert the universal column_ids to a field. The mesh must be directly extruded.
- scalar_copy:** Copies the scalar field. This is intended for testing purposes only.
- scalar_galerkin_projection:** Galerkin projects the scalar field.
- helmholtz_smoothed_scalar:** Smooth a scalar field by inverting a Helmholtz operator.

helmholtz_anisotropic_smoothed_scalar: Smooth a scalar field by inverting a Helmholtz operator.

lumped_mass_smoothed_scalar: Smooth a scalar field by inverting the lumped mass: $ML_{S_smooth} = M S$.

particle_reynolds_number: Diagnostic algorithm used in multiphase flow simulations. Calculates the particle Reynolds number, $(\alpha_f \rho_f |\mathbf{u}_f - \mathbf{u}_p| d) / \mu_f$, where the subscripts f and p denote the fluid (i.e. continuous) and particle (i.e. dispersed) phases respectively, and d is the particle diameter.

apparent_density: Diagnostic algorithm used in multiphase flow simulations. Calculates the apparent density of the material_phase, i.e. Density multiplied by the PhaseVolumeFraction field.

9.2.1.3 Python diagnostic algorithms

A python diagnostic algorithm, chosen via `.../diagnostic/algorithm::scalar_python_diagnostic` (or similar equivalents for vector and tensor fields) allows direct access to the internal Fluidity data structures in the computation of a diagnostic field. The python code entered at `.../diagnostic/algorithm::scalar_python_diagnostic` can access three variables: the simulation timestep `dt`, the diagnostic field `field`, and the simulation state `state`. `field` and `state` are python representations of the internal Fluidity data structures - see appendix B for more complete documentation of the Python state interface.

```
deltaT = 4.0

t = state.scalar_fields["Temperature"]
assert(t.node_count == field.node_count)

for i in range(field.node_count):
    tMinusT0 = t.node_val(i) * deltaT
    diagVisc = 1.620e-2 * (1.0 - 2.79e-2 * tMinusT0 + 6.73e-4 \
        * tMinusT0 * tMinusT0)
    visc = numpy.zeros((3, 3))
    for j in range(3):
        visc[j][j] = diagVisc
    field.set(i, visc)
```

Example 9.1: A tensor python diagnostic algorithm defining a temperature varying viscosity used in a baroclinic annulus simulation, configured as in [Hignett et al. \[1985\]](#) table 1 (main comparison).

9.2.1.4 Other diagnostic algorithms

scalar_field diagnostic algorithms:

extract_scalar_component: Extracts a Cartesian component of a named vector field (see attributes). `Element component` sets which Cartesian component is extracted.

9.3 Offline diagnostics

There are three main types of offline diagnostics:

- `fltools`, section 9.3.3: programs written in Fortran that are compiled by running `make fltools` on the command line in the top directory of the Fluidity source tree. The binaries are built in `bin`. The F90 source files can be found in `tools`
- python scripts, section 9.3.3: run `make fltools` in the top directory of the Fluidity source tree and these will be found in the `bin`. The source code can be found in `tools`.
- python modules: modules that can be imported e.g. for use in a test case. They are found in `python/fluidity/diagnostics` and are imported with `import python.fluidity.diagnostics.modulename`

9.3.1 vtktools

`vtktools.py` is a set of python tools that allows you to analyse data from a `vtu` or `pvtu`. The tools are based on python `vtk` and the module can be found in `python/` directory of the Fluidity trunk. To extract the pressure and velocity fields from a `vtu`, for example, use

```
import sys
sys.path.append('fluidity\_source\_path/python/')

import vtktools
data = vtktools.vtu('example.vtu')
p = data.GetScalarField('Pressure')
uvw = data.GetVectorField('Velocity')
u = uvw[:,0]
```

This performs the following steps

- imports the python `sys` module and then appends the directory which `vtktools.py` is in to the system path;
- imports the `vtktools` module;
- makes an object called `data` which contains the information about the `vtu` `example.vtu` (if running in parallel this would be `example.pvtu`);
- the pressure field is then extracted using `GetScalarField` which returns an array;
- similarly the full velocity field is extracted using `GetVectorField` which returns an array;
- finally the horizontal velocity field is obtained by picking the relevant values from the full velocity field.

A full list of the available tools is given in table 9.1. A summary can also be obtained by typing `help(vtktools)` in an ipython session. Further examples of use can be found in the online at [Cook Book](#) in the python scripts used for postprocessing of the examples and in the test cases.

method	arguments	use
AddField	name, array	Adds the values in array (the entries of which may have an arbitrary number of components) as a field called name
AddFieldtoField	fieldname, array, newFieldName=None	Adds the values in array to the field fieldname. If newFieldName is specified then a new field with that name is created with the new values, otherwise the original field is replaced.
AddScalarField	name, array	Adds a scalar field called name using the values in array
AddVectorField	name, array	Adds a vector field called name using the values in array
ApplyCoordinateTransformation	f	Applies the coordinate transformation specified in the function f to the grid coordinates. It will overwrite the existing coordinate values. An example for f def f(X,t=0): return [X[0]*t,X[1],X[2]]
ApplyEarthProjection		assumes the input geometry is Cartesian and projects to longitude, latitude and depth. It will overwrite the existing coordinate values.
ApplyProjection	projection_x, projection_y, projection_z	Applies a projection to the grid coordinates. It will overwrite the existing values. projection_x, projection_y and projection_z should all be strings that contain the projection to be evaluated, with x, y and z for the x, y and z coordinates respectively. For example, projection_x='x2'+ will translate all the values of x by 2.
CellDataToPointData		transforms all cell-wise fields to point-wise fields. All existing fields will remain,
Crop	min_x, max_x, min_y, max_y, min_z, max_z	Crops the edges defined by the bounding box given by the arguments
CrossFieldWithField	fieldName, array, newFieldName=None, postMultiply=True	Calculates the cross product, $\mathbf{a} \times \mathbf{b}$, where \mathbf{a} is the field fieldName and \mathbf{b} is array. If postMultiply \neq True then $\mathbf{b} \times \mathbf{a}$ will be calculated. If newFieldName is specified then a new field with that name is created that takes the values of the cross product, otherwise the original field is replaced.
DotFieldWithField	fieldName, array, newFieldName	Calculates the dot product of the field called fieldName and array. If newFieldName is specified then a new field with that name is created that takes the values of the dot product, otherwise the original field is replaced.
GetCellPoints	id	Returns an array with the node numbers of the cell (mesh element) number id.
GetCellVolume	id	Returns the volume of the cell (mesh element) with number id

GetDerivative	name	Returns the derivative of the field called name. Each component of the returned array has form $\frac{\partial T}{\partial x}, \frac{\partial T}{\partial y}, \frac{\partial T}{\partial z}$ for a scalar field and $\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \frac{\partial w}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}, \frac{\partial w}{\partial y}, \frac{\partial u}{\partial z}, \frac{\partial v}{\partial z}, \frac{\partial w}{\partial z}$ for a vector field where T is a scalar field, (u, v, w) a vector field and (x, y, z) the spatial coordinate vector field. The field name has to be point-wise data. The returned array gives a cell-wise derivative. (To obtain the point-wise derivative add the field to the vtk object and use CellDataToPointData.)
GetDistance	x, y	Returns the distance in physical space between x and y
GetField	name	Returns an array with the values of the field called name.
GetFieldIntegral	name	Returns the integral over the domain of the field called name.
GetFieldNames		Returns the name of the available fields.
GetFieldRank	name	Returns the rank of the field called name.
GetFieldRms	name	Returns the root mean square (RMS) of the supplies scalar or vector field called name.
GetLocations		Returns an array with the locations of the nodes.
GetPointCells	id	Returns an array with the elements which contain the node id.
GetPointPoints	id	Returns the nodes that connect to the node id.
GetScalarField	name	Returns an array with the values of the scalar field called name.
GetScalarRange	name	Returns the range (min, max) of the scalar field called name.
GetVectorField	name	Returns an array with the values of the vector field called name.
GetVectorNorm	name	Returns an array with the norm of the vector field called name.
GetVorticity	name	Returns the vorticity of the vector field called name. The vector field name has to be point-wise data. The returned array gives a cell-wise derivative. (To obtain the point-wise derivative use CellDataToPointData.)
IntegrateField	field	Returns the integral of the field called field assuming a linear representation on a tetrahedral mesh.
ManipulateField	fieldName, manipFunc, newFieldName=None	Generic field manipulation method. Applies the supplied manipulation function, manipFunc, to the field called fieldName. manipFunc must have form def manipFunc(field, index): ... return fieldValAtIndex. If newFieldName is specified then a new field with that name is created that takes the calculated values, otherwise the original field is replaced.

MatMulFieldWithField	fieldName, array, newFieldName=None, postMultiply=True.	Multiplies two matrices $\bar{A}\bar{B}$ where \bar{A} is the field fieldName, and \bar{B} is array. If postMultiply \neq True then $\bar{A}\bar{B}$ will be calculated. If newFieldName is specified then a new field with that name is created that takes the values of the product, otherwise the original field is replaced.
ProbeData	coordinates, name	Returns an array of values of the field called name at the positions given in coordinates. The values are calculated by interpolation of the field to the positions given. coordinates can be created using vtktools.arr() e.g. coordinates=vtktools.arr([[1,1,1],[1,1,2]]).
RemoveField	name	Removes the field called name.
StructuredPointProbe	nx, ny, nz, bounding_box=None	Returns a vtk structured object. nx, ny, nz are the number of points in the x, y, z directions respectively. If bounding_box is not specified the bounding box of the domain is calculated automatically. If specified bounding_box = [xmin, xmax, ymin, ymax, zmin, zmax].
SubFieldFromField	fieldName, array, newFieldName=None	Subtracts array from the field called fieldName. If newFieldName is specified then a new field with that name is created that takes the values of the product, otherwise the original field is replaced.
Write	filename = []	Writes the data to a vtu file. If filename is not specified the name of the file originally read in will be used and therefore the input file will be overwritten.
functions	arguments	use
VtuDiff	vtu1, vtu2, filename=None	Generates a vtu with fields that are the difference between the field values in the two supplied vtus, vtu1 and vtu2. Fields that are not common between the two vtus are neglected. If the cell points of the vtus do not match then the fields of vtu2 are projected onto the cell points of vtu1.
VtuMatchLocations	vtu1, vtu2, tolerance= 9.9999999999999995e-07	Checks that the locations in the supplied vtus, vtu1 and vtu2 match to within the value of tolerance. The locations must be in the same order.
VtuMatchLocationsArbitrary	vtu1, vtu2, tolerance= 9.9999999999999995e-07	Checks that the locations in the supplied vtus, vtu1 and vtu2 match to within the value of tolerance. The locations may be in a different order.
arr	object, dtype=None, copy=True, order=None, subok=False, ndim=True	Creates an array from object, where object must be an array, any object exposing the array interface, an object whose _array_ method returns an array, or any (nested) sequence.

`dtype`: data-type, optional. The desired data-type for the array. If not given (the default), then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting use the `.astype(t)` method.
`copy`: bool, optional. If `True` (default), then the object is copied. Otherwise, a copy will only be made if `_array_` returns a copy, if `object` is a nested sequence, or if a copy is needed to satisfy any of the other requirements.
`order`: {'C', 'F', 'A'}, optional. Specify the order of the array. If order is 'C' (default), then the array will be in 'C'-contiguous order (last-index varies the fastest). If order is 'F' then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontinuous).
`subok`: bool, optional. If `True`, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).
`ndim`: int, optional. Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.
 For examples see the module help information.

Table 9.1: Table of tools available in `vtktools` for extraction and analysis of data in `vtu` and `pvtu` files. The tools which are methods can be applied to an object created from a `vtu` or `pvtu` with `data = vtktools.vtu('example.vtu')`. The methods can then be used with `data.method(arguments)`. Note only the `write` method can change the original `vtu` or `pvtu`. The functions are used as usual `python` functions. See section 9.3.1, the [Cook Book](#) the postprocessing `python` scripts in the examples, the test cases and the source code for the `python` tools in `tools` for further examples of use. The list can also be obtained in a `ipython` session with the command `help(vtktools)`.

9.3.2 Diagnostic output

9.3.3 fltools

An extended set of Fluidity tools exist that supplement the main Fluidity program. Table 9.2 lists them and descriptions can be found by referring to the relevant section. The tools can be built by running `make fltools` in the top directory of the Fluidity trunk. The programs generated can then be found in the `bin/` directory.

9.3.3.1 checkmesh

`checkmesh` can be used to form a number of verification tests on a mesh in triangle mesh format. It is run from the command line:

```
checkmesh input_basename
```

where `input_basename` is the base name of an input triangle file. `checkmesh` tests for:

- Degenerate volume elements
- Inverted tetrahedra
- Degenerate surface elements
- Mesh tangling

`checkmesh` is parallelised. If running in parallel, it should be launched on a number of processes equal to that in the mesh decomposition. In parallel `checkmesh` output is written to `checkmesh.log-[process]` and `checkmesh.err-[process]` log files.

```
Checking volume elements for tangling ...
In intersection_finder
In advancing_front_intersection_finder
Exiting advancing_front_intersection_finder
Exiting intersection_finder
Tangled volume element found:
Element: 1
Coordinates:
  0.10000000000000000E+001  0.00000000000000000E+000
  0.11666666666700001E+001  0.00000000000000000E+000
  0.83867056794499995E+000  0.54463903501499999E+000
Numbering:
      1
     14
      2
```

Example 9.2: `checkmesh` reporting a mesh tangling error.

9.3.3.2 clean_mayavi_mesh

This program cleans up vector eps mesh images output by Mayavi2. The program removes pointy line joins which are unsightly on very anisotropic meshes and calculates an accurate bounding box, replacing the incorrect one produced by Mayavi2.

Program	Section
checkmesh	9.3.3.1
clean_mayavi_mesh	9.3.3.2
create_aligned_mesh	9.3.3.3
create_climatology_atlas	9.3.3.4
create_param_sweep	9.3.3.5
differentiate_vtu	9.3.3.6
edge_length_distribution	9.3.3.7
encode	9.3.3.8
fladapt	9.3.3.9
fldecomp	9.3.3.10
fldiagnostics	9.3.3.11
flrecomp	9.3.3.12
genpvtu	9.3.3.13
gen_square_meshes	9.3.3.14
gmsh2triangle	9.3.3.15
gmsh_mesh_transform	9.3.3.16
interval	9.3.3.17
linear_interpolation	9.3.3.18
mean_flow	9.3.3.19
mms_tracer_error	9.3.3.20
nodecount	9.3.3.21
periodise	9.3.3.22
petsc_readnsolve	9.3.3.23
probe_vtu	9.3.3.24
project_to_continuous	9.3.3.25
project_vtu	9.3.3.26
pvtu2vtu	9.3.3.27
rename_checkpoint	9.3.3.28
runut	9.3.3.29
stat2csv	9.3.3.30
statplot	9.3.3.31
streamfunction_2d	9.3.3.32
supermesh_difference	9.3.3.33
transform_mesh	9.3.3.34
triangle2vtu	9.3.3.35
unifiedmesh	9.3.3.36
update_options	9.3.3.37
vertical_integration	9.3.3.38
visualise_elements	9.3.3.39
vtkdiagnostic	9.3.3.40
vtk_projection	9.3.3.41
vtudecomp	9.3.3.42
vtudiff	9.3.3.43
vtu_bins	9.3.3.44

Table 9.2: Table of fltools. On running `make fltools` they can be found in the `bin/` directory of the Fluidity trunk.

The input file should be an eps file output by the “save scene as” “Vector PS/EPS/PDF/TeX” option in Mayavi2.

```
clean_mayavi_mesh [--margin=n] <input_filename> <output_filename>
```

If present, `--margin` sets the margin between the figure and the bounding box.

9.3.3.3 create_aligned_mesh

Creates the triangle files for a mesh that lines up in all directions, so that it can be made it into a singly, doubly or triply periodic mesh. It is run from the command line:

```
create_aligned_mesh newmesh Lx Ly Lz Nx Ny Nz [Ox Oy Oz]
```

This creates a box that is $L_x \times L_y \times L_z$ with N_x, N_y, N_z layers in the x, y, z directions respectively. The optional arguments O_x, O_y, O_z specify the position of the origin. Information can be found by running `create_aligned_mesh --help` on the command line. Note the mesh will always be 3d.

9.3.3.4 create_climatology_atlas

This creates a climatology atlas, for use with ICOM, using “High resolution ($1/4^\circ$) Temperature and Salinity Analyses of the World’s Oceans. Version 2”

The $1/4^\circ$ grid climatological mean fields of in situ temperature (degrees Celsius) and salinity (practical salinity scale) for the annual, seasonal, and monthly time periods were calculated by Boyer et al. [2005] using objective analysis techniques. The data and associated metadata was obtained from the NODC, http://www.nodc.noaa.gov/OC5/WOA01/qd_ts01.html. All the data files are gzipped ASCII $1/4$ gridded data files (‘Girdded Fields’) and contain the DOS end-of-line character (M). There are 12 monthly averages of temperature and 12 monthly averages of salinity. In addition, there 4 seasonal averages of temperature and 4 seasonal averages of salinity corresponding to winter (defined as January, February and March), spring summer and autumn. Further information can be found online at the above address.

For use with ICOM (relaxing to climatology at the boundaries), a NetCDF data was created containing the monthly means, and the additional 9 standard levels from the seasonal means in order to provide information below the 1500m level. In addition, Killworth correction (Killworth 1995) is applied to the data in order to facilitate accurate time interpolation.

To use this just execute it in the directory which contains all the files listed above gunzipped.

9.3.3.5 create_param_sweep

It is sometimes useful to vary certain parameters of your simulation to assess the effect of that parameter on some part of the output. This can be a tedious thing to set up, especially if you wish to cover a number of parameters and their interacting effects. A script (`create_param_sweep`) allows this to be done very easily.

Usage

Take a directory which contains everything you need for a simulation; mesh files, flml, any input files, python scripts, etc. Next generate a parameter file in the following format:

```
NAME; spud_path; colon:seperated:parameter:values
NAME2; spud_path2; colon:seperated:parameter:values
```

The values should be in "Python" format, e.g. a tensor is `[[0, 0, 1], [3, 5, 23], [34, 6, 2]]`. The spud path can be copied from diamond (i.e. click on the parameter you wish to vary and click "copy path"). The name is a human readable name that will be used in the output files.

You can then run the script:

```
create_param_sweep template/ sens_test/ param_file.txt
```

where `template` is your set up directory, `sens_test` is where you want all the files to be created (doesn't need to exist already) and `param_file.txt` is your parameter space file. When complete you will have the following directory structure:

```
output_dir/
  template (copied in if not already here)
  runs/
    1/
      run.flml
      other files
    2
    3
  directory_listing.csv
```

`directory_listing.csv` will contain the directory numbers and which parameter set they contain. Each run is contained in a separate, numbered, directory.

9.3.3.6 differentiate_vtu

`differentiate_vtu` takes the gradient of every scalar field and vector field component within an `vtu`. It is primarily intended for cases where the derivatives of large numbers of fields are required. `differentiate_vtu` is built as part of the `fltools` build target (see section 9.3.3), and is used via:

```
differentiate_vtu [-v] input_filename output_filename [input_fieldname]
```

where `input_filename` is the input `vtu` and `output_filename` is the output `vtu`. If `input_fieldname` is supplied then only that field is differentiated. The `-v` flag enables verbose output.

9.3.3.7 edge_length_distribution

`edge_length_distribution` is a python script that creates histograms of the edge lengths of the elements in the mesh at each time step. It also produces line graphs of the maximum and minimum edge lengths as a function of time. This is useful for analysing adaptive runs to see how the edge lengths compare to a fixed run of the same process. It is run from the command line:

```
edge_length_distribution [options] vtu_filename
```

where `vtu_filename` is the input `vtu` file. The options available are:

```
-s START_VTU      The first dump id of files to be included, default = 0
-e END_VTU        The last dump id of files to be included, if not used
                  all vtus with id >= start_vtu will be included
-b NO_BINS        Number of bins for the histogram plots, default = 10
-m               Plots the maximum and minimum edge lengths over time
-c               Plots a histogram of the cumulative total of edge
                  lengths for all vtus
-p               Will allow plots to be made from the data in the log
```

files 'time.log' and 'edge_lengths.log' rather than extracting the information from the vtus as this can take a while. Note: you must have run the script once WITHOUT this option otherwise the log files will not exist

--pvtu Uses pvtus instead of vtus

9.3.3.8 encode

Script to encode .avi movies playable on Windows machines. Requires arguments:

1st image file type (e.g. jpg,png,etc...)
 2nd path to folder containing images, and for movie output.
 3rd name of movie output file (without the file extension).
 4th OPTIONAL - enter characters here to limit the image files used as frames (e.g. 'image1' would only select images which satisfy 'image1*.jpg').

For example, `encode.sh jpg /data/movie_files my_movie image002` will encode a movie called `my_movie` using all jpg files from the directory `/data/movie_files/` starting with the characters 'image002'.

9.3.3.9 fladapt

fladapt performs a single mesh adapt based on the input options file (which may be a checkpoint) and outputs the resulting mesh. It is run from the command line:

```
fladapt [options] INPUT OUTPUT
```

where INPUT is the name of the input options file and OUTPUT is the name of the generated mesh. The flag `-v` flag enables verbose output and the flag `-h` flag displays the help message.

9.3.3.10 fldecomp

fldecomp is used to decompose a mesh into multiple regions, one per process. In order to run fldecomp, if your mesh files have the base name `foo` and you want to decompose into four parts, type:

```
fldecomp -n 4 -m mesh_format mesh_file
```

Where:

`mesh_file` is the base name of your mesh file(s). For example, `foo` for `foo.msh` with Gmsh format, or `foo.face/node/ele` with triangle format.
`mesh_format` is the format of the mesh file you wish to decompose. It can take two values: `gmsh` or `triangle`. If you omit the `-m` option, fldecomp will default to `triangle`.

For performance reasons, fldecomp supports only binary Gmsh files. These are generated by passing the `-bin` argument to Gmsh, for example:

```
gmsh -3 -bin project.geo -o project.msh
```

This creates a 3D binary Gmsh mesh called `project.msh` from the geometry file.

9.3.3.11 fldiagnostics

fldiagnostics is an offline diagnostic tool. It is run from the command line:

```
fldiagnostics ACTION [OPTIONS] INPUT OUTPUT [FIRST] [LAST]
```

If `FIRST` is supplied, treats `INPUT` and `OUTPUT` as project names, and processes the specified range of project files. The options are:

```
add[-diag]  Add a diagnostics field. Options:
             -m NAME      Field from which to extract a mesh to use with the
                       diagnostic field (default "Velocity")
             -o NAME      Diagnostic field name
             -r RANK      Specify the rank of the diagnostic field (will try all
                       ranks if not supplied, but will also suppress useful
                       error messages)
             -s STATE     Name of the state from which to read options in the
                       options file (defaults to the first state)
             -x FLML      Model options file (not always required)
```

and the `-h` flag displays the help message. To add the diagnostic field `GridReynoldsNumber` to a set of vtus run:

```
fldiagnostics add -o GridReynoldsNumber -r 0 ...
                ... lock_exchange.vtu lock_exchange_out 5 8
```

9.3.3.12 flredecomp

flredecomp is a parallel tool that performs a re-decomposition of a Fluidity checkpoint. It is invoked as follows:

```
mpiexec -n [target number of processors] flredecomp \
        -i [input number of processors] \
        -o [target number of processors] [input flml] [output flml]
```

For example, to decompose the serial file `foo.flml` into four parts, type:

```
mpiexec -n 4 flredecomp \
        -i 1 -o 4 foo foo_flredecomp
```

The output of running `flredecomp` is a series of mesh and vtu files as well as the new flml; in this case `foo_flredecomp.flml`. Note that `flredecomp` must be run on a number of processors equal to the larger number of processors between input and output.

9.3.3.13 genpvtu

genpvtu creates a set of pvtus from a base set of vtus. It is run from the command line:

```
genpvtu basename
```

with `basename` that of the vtu set e.g. `{example_0.0.vtu, example_0.1.vtu, example_0.2.vtu; example_1.0.vtu, example_1.1.vtu, example_1.2.vtu}` has `basename` 'example'. It will produce pvtus with names `example_0.pvtu, example_1.pvtu`.

9.3.3.14 `gen_square_meshes`

`gen_square_meshes` will generate triangle files for 2D square meshes. It is run with python from the command line:

```
gen_square_meshes [OPTIONS] NODES MESHES
```

NODES is the number of nodes in the mesh and MESHES is the number of meshes to be created. [OPTIONS] are -h for help and -v for Verbose mode.

9.3.3.15 `gmsh2triangle`

`gmsh2triangle` converts ASCII Gmsh mesh files into triangle format. Whilst Fluidity can read in Gmsh files directly as noted in section 6.1, this tool should be used in cases where native Gmsh support does not work. It is run from the command line:

```
gmsh2triangle [--2d] input
```

where `input` is the input `.msh` file. The `--2d` flag can be used to instruct `gmsh2triangle` to process a 2D input `.msh` file. Otherwise, 3D input is assumed.

9.3.3.16 `gmsh_mesh_transform`

`gmsh_mesh_transform` applies a coordinate transformation to a region of a given mesh. It is run from the command line:

```
gmsh_mesh_transform [constants] region transformation mesh
```

where `constants` is an optional list of constant associations separated by commas, for use in region and transformation, `region` is a python expression which evaluates to true over the region to be transformed (use 'True' for whole domain), `transformation` is a python expression giving the coordinate transformation and `mesh` is the name of the gmsh mesh file. Note: This script creates a backup of the original mesh file with a '.bak' extension. Examples:

To rescale the z-dimension by a factor of 1000:

```
gmsh_mesh_transform '(x,y,1000*z)' mesh.msh
```

To project all points that lie within a circle of centre (`xcentre`, `ycentre`) in z by a distance `zprojection`:

```
gmsh_mesh_transform 'xcentre=50, ycentre=50, radius=20, zprojection=50'
'(x-xcentre)**2 + (y-ycentre)**2 < radius**2' '(x, y, z+zprojection)' mesh.msh
```

To project all points that lie within a circle of centre (`xcentre`, `ycentre`) in z in the shape of a cone, by a distance `zprojection` at the centre:

```
gmsh_mesh_transform 'xcentre=50, ycentre=50, radius=20, zprojection=50'
'(x-xcentre)**2 + (y-ycentre)**2 < radius**2' '(x, y, z + zprojection *
(1 - sqrt((x-xcentre)**2 + (y-ycentre)**2) / radius ) )' mesh.msh
```

To add an ice shelf to a meshed box for x in `[0,shelflength]` and z in `[0,shelfslopeheight+minoceandepth]`. Note this applies to both 2d and 3d domains and the ocean domain can extend further:

```
gmshtool mesh_transform 'shelflength = 550, shelveslopeheight = 800,
minoceandepth = 100' 'x < shelflength' '(x, y, (z/(shelveslopeheight
+ minoceandepth)) * ((x/shelflength) * shelveslopeheight + minoceandepth))'
mesh.msh
```

9.3.3.17 interval

This is a one-dimensional mesh generator. It is run from the command line:

```
interval [options] left right name
```

where left and right define the range of the line. It has a number of user defined input options:

```
--dx          constant interval spacing
--variable_dx interval spacing defined with a python function
--region_ids  python function defining the region ID at each point
--reverse     reverse order of mesh
```

9.3.3.18 linear_interpolation

linear_interpolation linearly interpolates all the fields of a set of vtus on to the mesh of a target vtu. It is run with python from the command line:

```
linear_interpolation TARGET VTU1 VTU2 VTU3
```

TARGET is the name of the mesh that the fields will be interpolated on to. VTU1, VTU2, VTU3 are the vtus from which the fields will be interpolated (the number of vtus can be 1 or more, 3 are used here for illustration). An output vtu called interpolation_output.vtu will be generated that will contain all the interpolated fields from all the vtus.

9.3.3.19 mean_flow

mean_flow calculates the mean of the fields in a set of vtus. It is run with python from the command line:

```
mean_flow [options] vtu_basename first_id last_id
```

where the range of vtu files is defined by vtu_basename first_id last_id. The options include specification of an area of the domain to sample and the number of sampling planes in each direction:

```
-b, --bbox xmin/xmax/ymin/ymax/zmin/zmax
      bounding box of sampling window
-i, --intervals i/j/k
      number of sampling planes in each direction
-v, --verbose
      verbose output
```

The option -h provides further information on how to use these.

9.3.3.20 mms_tracer_error

mms_tracer_error evaluates the error between two fields using either an L_2 or L_∞ control-volume norm. It is called from a python script e.g.:


```
import mms_tracer_error as error
l2_error = error.l2("MMS.vtu", "field1", "field2")
inf_error = error.inf("MMS.vtu", "field1", "field2")
```

where `MMS.vtu` is the name of the `vtu` and `field1`, `field2` are two scalar fields in the `vtu` to be compared.

9.3.3.21 nodecount

`nodecount` will return the number of nodes in a list of `vtus`. It is run from the command line:

```
nodecount [vtulist]
```

If `vtulist` is not supplied it will run on any `vtu` in the working directory.

9.3.3.22 periodise

`periodise` is used to create a periodic mesh. The input to `periodise` is your `flml` (in this case `foo.flml`). This `flml` file should already contain the mapping for the periodic boundary as described in section 8.4.2.3. `Periodise` is run with the command:

```
<<fluidity_source_path>>/bin/periodise foo.flml
```

The output is a new `flml` called `foo_periodised.flml` and the periodic meshes. Next run `flredec` (section 6.6.4.2) to decompose the mesh for the number of processors required. The `flml` output by `flredec` is then used to execute the actual simulation:

```
mpiexec -n [number of processors] \
  <<fluidity_source_path>>/bin/fluidity [options] \
  foo_periodised_flredec.flml
```

9.3.3.23 petsc_readnsolve

Whenever in Fluidity a linear solve has not completed successfully the equation is dumped out in a file called `matrixdump`. This file can be used to analyse the behaviour of this solve without having to rerun the model with `petsc_readnsolve`. It reads in the `matrixdump` and tries to solve it with PETSc options set in the `flml` file. It is advisable to first reproduce the failing solve with the same options as it happened in Fluidity. After that the solver options in the `flml` file can be changed to see if that fixes the problem. The options under `.../solver/diagnostics` are particularly useful to diagnose the problem. `petsc_readnsolve` is run from the command line:

```
petsc_readnsolve FILENAME FIELDNAME [OPTIONS]
```

where `FILENAME` is the relevant `flml` file and `FIELDNAME` is the field for which the solve was failing. The `OPTIONS` available are:

<code>-l</code>	Write the information that is normally printed in the terminal to a log file called <code>petsc_readnsolve.log-0</code> .
<code>-prns_verbosity N</code>	Determines the amount of information that is printed to the terminal. By default <code>petsc_readnsolve</code> uses the maximum verbosity (3), this can be lowered with this option.
<code>-prns_filename file</code>	reads from the specified file instead of

```

                                'matrixdump'
-prns_zero_init_guess          no initial guess is read from matrixdump
                                instead the initial guess is zeroed
-prns_write_solution file      writes solution vector to specified file
                                so that it can be used for comparison in
                                next runs of petsc_readnsolve (provided
                                we are sufficiently confident in the
                                accuracy of the obtained solution).
-prns_read_solution file       reads solution vector from the specified file,
                                so that exact errors can be calculated. For
                                small matrices a good approximation of the
                                exact solution can be found using a direct
                                method: select iterative_method "preonly"
                                and preconditioner "lu" in the .flml. Note
                                however that for ill-conditioned matrices
                                direct methods are very sensitive to round
                                off errors
-prns_scipy                     writes out several files that can be read in scipy
-prns_random_rhs               Instead of the rhs in the matrixdump, use a
                                random rhs vector.

```

Additionally all options that are available from the PETSc library may be added to the command line. Options specified in the flml file always take precedence however. Some PETSc useful options:

```

-ksp_view           Information on all the solver settings.
-mat_view_info     Information on matrix size
-mat_view_draw     Draw the matrix nonzero structure
-help             Gives an overview of all PETSc options that can be given for
                 the selected solver/preconditioner combination.

```

Parallel

When a solve fails in a parallel run, a single matrixdump file is written. petsc_readnsolve can be run in serial on this matrixdump but owing to the usual large size of a parallel run and that the behaviour of a solver in parallel is often different than in serial, it is generally better to run petsc_readnsolve in parallel as well. This is done by prepending `mpirun -np N` on the command line (where N is the number of processes).

petsc_readnsolve in parallel requires the mesh files of the same mesh as the one used by Fluidity during the failing solve. Therefore, for adaptive runs, a checkpoint at the point of the failing solve is required and then the checkpoint flml is used for petsc_readnsolve. In most cases the mesh files are not needed for serial runs of petsc_readnsolve, even if the Fluidity run was parallel.

9.3.3.24 probe_vtu

Returns the value of a field at a specified coordinate in a vtu file. probe_vtu is built as part of the fltools build target (see section 9.3.3), and is used via:

```
probe_vtu [-v] input_filename field_name x [y z]
```

where input_filename is the input vtu and field_name is the field to be probed. x, y and z are the coordinates at which the field is to be evaluated. The -v flag enables verbose output.

9.3.3.25 `project_to_continuous`

`project_to_continuous`, given a `vtu` file containing fields on a discontinuous mesh and triangle files for the corresponding continuous mesh, will produce a `vtu` with its fields projected onto the continuous mesh. It is run from the command line:

```
project_to_continuous [OPTIONS] vtufilename trianglename
```

where `vtufilename` is the name of the discontinuous `vtu` and `trianglename` is the base name of the triangle files. The flag `-h` prints out the help message and the flag `-v` enables verbose output.

9.3.3.26 `project_vtu`

`project_vtu` performs a Galerkin projection on a given `vtu` file from a specified donor mesh to a target mesh. It is run from the command line:

```
project_vtu [OPTIONS] input donor_basename target_basename output
```

where `input` is the name of the `vtu` file to be projected and `donor_basename` is the base name of the triangle files defining the mesh that corresponds to the input `vtu` file. `target_basename` is the base name of the triangle files defining the output mesh and `output` is the name of the output `vtu` file. The flag `-h` prints out the help message and the flag `-v` enables verbose output.

9.3.3.27 `pvtu2vtu`

`pvtu2vtu` combines `pvtus` into `vtus`. It is run from the command line:

```
pvtu2vtu [OPTIONS] PROJECT FIRSTID [LASTID]
```

with `PROJECT` the basename of the `pvtus` and `FIRSTID` and `LASTID` the first and last id numbers respectively of the `pvtus` to be included. `LASTID` is optional and defaults to `FIRSTID`. Running with the option `-h` will give further information on the other options available. Note, this may not always work with `vtus` from adaptive runs.

9.3.3.28 `rename_checkpoint`

`rename_checkpoint` takes a list of `vtu` files in the working directory produced from a serial checkpointed `flml` file with names `base_filename_checkpoint.i.vtu` for all `i` and renames them as `base_filename_index+i.vtu`. Additionally it may take a list of `vtu` and `pvtu` files in the current directory produced from a checkpointed parallel `flml` file with names `base_filename_checkpoint.i.j.vtu` and `base_filename_checkpoint.i.pvtu` for all `i` (index) and `j` (processor number) and renames them as `base_filename_index+i.j.vtu` and `base_filename_index+i.pvtu`. It is run from the command line with:

```
rename_checkpoint [options] base_filename index
```

9.3.3.29 `runut`

Run a specified unit test with:

```
runut UNIT\_TEST\_NAME
```

where `UNIT_TEST_NAME` is the name of the unit test.

9.3.3.30 stat2csv

stat2csv converts a Fluidity stat file into a csv file. It is run from the command line:

```
stat2csv [OPTIONS] PROJECT
```

with PROJECT the base name of the stat file. The default output is to PROJECT.csv. Running with the option -h will provide further information on the output file name, type and format.

9.3.3.31 statplot

statplot is a graphical program for previewing files in the .stat file format. statplot can be launched via:

```
statplot filename [filename2 filename3 ...]
```

This generates a graphical user interface displaying a plot of one statistic in the .stat file against another. The ordinate and abscissa statistics can be selected via combo boxes immediately beneath the plot. The plot itself can be navigated using the pylab navigation toolbar - see http://matplotlib.sourceforge.net/users/navigation_toolbar.html for more complete documentation.

If multiple .stat files are supplied, the data are combined. This is useful for visualising output for simulations that are checkpointed and resumed.

Additional keyboard commands:

Key	Function
s	Switch to scatter plot mode
l	Switch to line plot mode
r	Re-load the input file
R	Re-load the input file without changing the plot bounds
q	Quit
x	Toggle x-axis linear / log
y	Toggle y-axis linear / log

9.3.3.32 streamfunction_2d

streamfunction_2d solves the Poisson equation for the 2D streamfunction Ψ :

$$\nabla^2 \Psi = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}, \quad (9.1)$$

using a continuous Galerkin formulation. It applies the strong Dirichlet boundary condition of:

$$\Psi = 0 \quad \text{on } \partial\Omega, \quad (9.2)$$

for all surfaces, and hence is only suitable for problems with a no normal flow boundary condition on all surfaces.

streamfunction_2d is built as part of the fltools build target (see section 9.3.3), and is used via:

```
streamfunction_2d [-v] input output
```

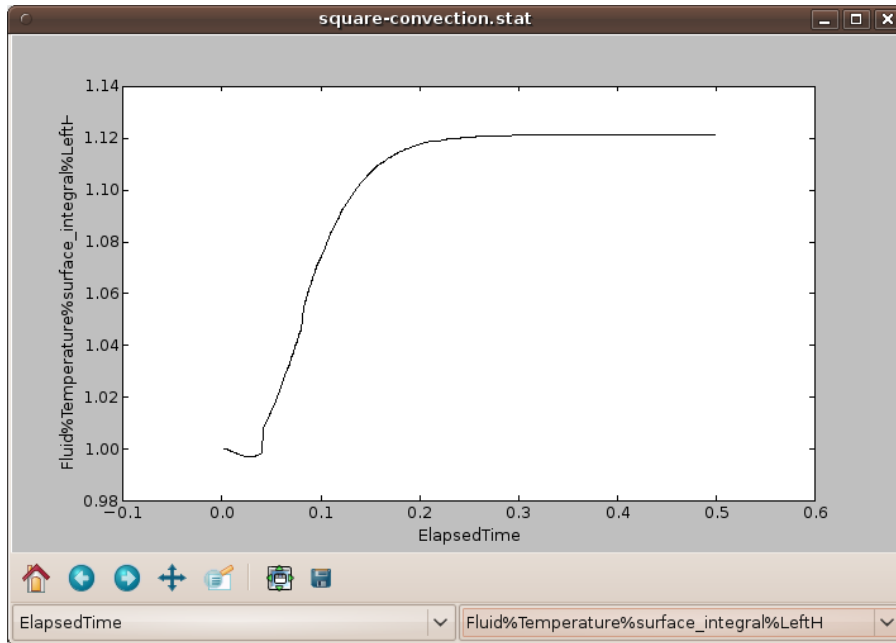


Figure 9.2: Visualisation of a heat flux diagnostic in a 2D cavity convection simulation using statplot.

where `input` is a `vtu` file containing a continuous vector field “Velocity”, and `output` is an output `vtu` filename. The `-v` flag enables verbose output.

`streamfunction_2d` can only be used in serial.

9.3.3.33 `supermesh_difference`

The `supermesh_difference` tool computes the difference between two `vtu` files using supermesh construction. `supermesh_difference` is built as part of the `fltools` build target (see section 9.3.3), and is used via:

```
supermesh_difference vtu_1 vtu_2 output_vtu
```

The output `vtu` (with filename `output_vtu`) is equal to the first `vtu` (with filename `vtu_1`) minus the second `vtu` (with filename `vtu_2`). The `-v` flag enables verbose output.

Be aware that `supermesh_difference` can generate extremely large output files, particularly in three dimensions. `supermesh_difference` can only be used in serial.

9.3.3.34 `transform_mesh`

`transform_mesh` applies a given coordinate transformation to the given mesh in triangle format. It is run from the command line:

```
transform_mesh transformation mesh
```

with `mesh` the base name of the triangle mesh files. A `mesh.node`, `mesh.face` and `mesh.ele` file are required. `transformation` is a python expression giving the coordinate transformation. For example to rescale the `z`-dimension by a factor of 1000 run:

```
transform_mesh '(x,y,1000*z)' mesh
```

It can be used for 2D or 3D meshes. Note that a similar script exists to work with `gmsh` files (section 9.3.3.16).

9.3.3.35 triangle2vtu

This converts triangle format files into vtu format. It is run from the command line:

```
triangle2vtu input
```

where `input` is the triangle file base name.

9.3.3.36 unifiedmesh

`unifiedmesh` dumps the supermesh constructed from two input meshes (see Farrell et al. [2009], Farrell and Maddison [2010]) to discontinuous triangle mesh files and a vtu file. It is run from the command line:

```
unifiedmesh <triangle-file-1> <triangle-file-2> <output-file-name>
```

No file names should have extensions and both the triangle mesh files and the vtu file will have the name `output-file-name`

9.3.3.37 update_options

`update_options` is a developer tool that bulk updates `flml`, `bml`, `swml` and `adml` files after schema changes. It is run from the command line:

```
update_options [OPTIONS] [FILES]
```

where `FILES` is a list of files to be updated. If this argument is not provided all files in `tests/*/*.`, `tests/*/*/*.`, `longtests/*/*.`, `longtests/*/*/*.` and `examples/*/*.` will be updated. The flag `-h` prints out the help message and the flag `-v` enables verbose output.

9.3.3.38 vertical_integration

The `vertical_integration` tool computes the Galerkin projection of the vertical integral of some field onto a specified target surface mesh, via supermeshing (see Farrell et al. [2009], Farrell and Maddison [2010]) of the source mesh with a vertical extrusion of the target mesh. This can be used to compute vertically integrated quantities for arbitrary unstructured meshes (with no columnar vertical structure).

`vertical_integration` is built as part of the `fltools` build target (see section 9.3.3), and is used via:

```
vertical_integration -b bottom -t top -s sizing [-d -p degree -v]
  target source output
```

where `target` is the target surface mesh triangle base name, `source` is the source vtu, and `output` is an output vtu filename.

The compulsory flags `-b bottom` and `-t top` define the lower and upper bounds of the vertical integral to be `bottom` and `top` respectively, and `-s sizing` sets the thickness of layers used in the computation of the vertical integral (the thickness of the layers in the vertical extrusion of the target mesh through the source mesh). The optional flag `-p degree` sets the polynomial degree of the output integral. If `-d` is supplied, the integral is output on a discontinuous mesh. Otherwise, it is output on a continuous mesh for non-zero polynomial degree, and a discontinuous mesh for a P_0 output mesh. By default the output field is P_1 (continuous). The `-v` flag enables verbose output.

`vertical_integration` can only be used in serial.

9.3.3.39 visualise_elements

`visualise_elements` produces VTK output which approximates the shape functions of higher order two dimensional elements which cannot be visualised directly by outputting `vtu` files. This is not useful for simulation output but is very useful for producing images of shape functions for presentations and publications.

`visualise_elements` uses Spud for options handling and is therefore driven by an xml input file rather like that used for Fluidity. This uses its own schema so the file is edited using:

```
diamond -s <<fluidity_source_path>>/schemas/visualise_elements.rng \
  <<myfile>>.xml
```

The program is then invoked with:

```
visualise_elements <<myfile>>.xml
```

There are two modes of operation for `visualise_elements`. The first produces an illustration of all the basis functions for a given function space on a single element. Figure 9.3 illustrates this mode of usage. The input file used to create this figure is available at `<<fluidity_source_path>>/tools/data/visualise_quadratic.xml`. Table 9.3 shows the options which are relevant to this mode of operation.

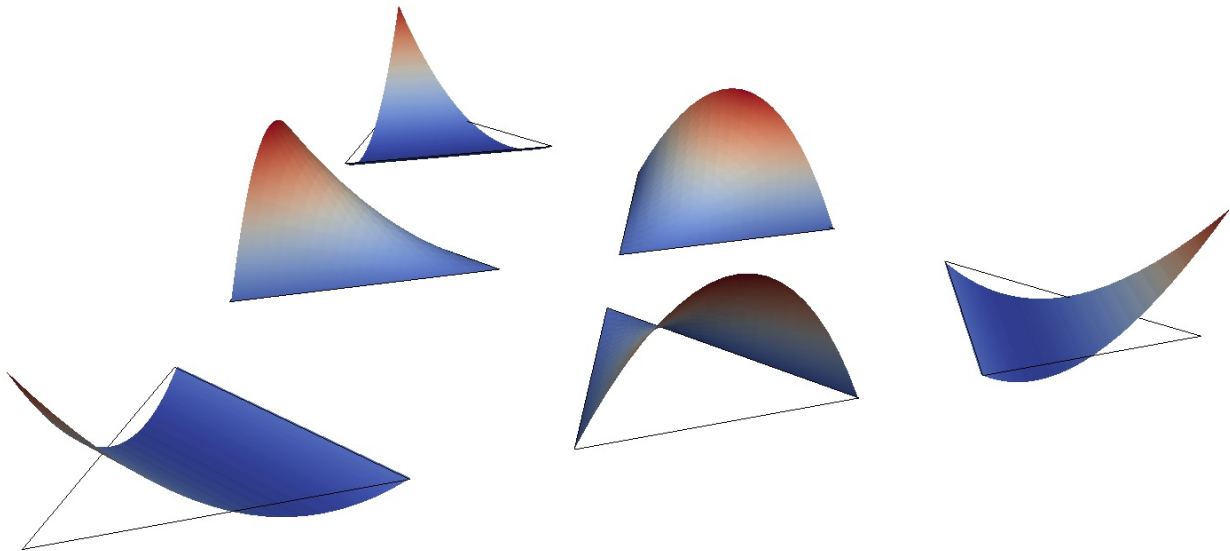


Figure 9.3: Visualisation of the six basis functions of the quadratic triangle generated by `visualise_elements`.

In the second mode of operation, a `CoordinateMesh` and a `TracerMesh` are specified. The `/material_phase` option is enabled and a prescribed tracer field is set up. In this mode, the output is a higher order visualisation of the specified tracer field. `<<fluidity_source_path>>/tools/data/visualise_sin.xml` is an input file for this mode of operation which visualises a sin function using quadratic elements. In this mode of operation, the `/geometry/element_vertices`, `/geometry/element_degree` and `/geometry/element_family` options are ignored in favour of the values found in the meshes. `/geometry/visualisation_degree` is still employed as the degree of the elements used to construct the visualisation.

Option	Meaning
/project_name	Base name for output files.
/geometry/element_vertices	3 for triangles, 4 for quads.
/geometry/element_degree	Degree of elements to visualise.
/geometry/quadrature/degree	Set to at least /geometry/element_degree.
/geometry/visualisation_degree	Higher values produce smoother visualisations.
/geometry/mesh	Do not enable.
/material_phase	Do not enable.

Table 9.3: Options relevant to the use of `visualise_elements` to visualise the basis functions of a given function space.

The output of `visualise_elements` is two `.vtu` files. The one whose name commences with the project name from the input file contains the actual rendered elements using a large number of visualisation elements per input element. The second, with the name commencing “outline”, contains no fields but has the original elements. The most effective visualisation mechanism is to load both files into paraview. Set the representation of the outline field to “Wireframe” and apply the “Warp By Scalar” filter to the visualisation field.

9.3.3.40 `vtkdiagnostic`

`vtkdiagnostic` runs diagnostics on a given `vtu` file. It is run from the command line:

```
vtkdiagnostic -i example.vtu [OPTIONS]
```

The `OPTIONS` are:

General options:

```
-i, --input=FILENAME
    VTU file to use as input
-p, --node-positions
    Print out node XYZ positions
-b, --buff-body-volume
    Volume of buff body
-c, --clip <scalar name>/<scalar value>/<orientation>
-e, --element-volumes
    Print out element volumes
--vtk-arrays=array1[,array2,...,arrayN]
    Print out contents of specified VTK arrays
    e.g. --vtk-arrays=Velocity,Temperature
-g, --debug
    Print debugging information
-o, --offset <scalar name>/<offset value>
-r, --radial-scaling=scale_factor
    Assume spherical Earth geometry and scale the radius
-h, --help
    Print this usage information
```

Vorticity integral diagnostic options:

```
-v, --vorticity-integral
    Perform vorticity integral diagnostic
-2, --2d
    Force treatment as a 2d problem
```



```
-d, --dump-vtu=FILENAME
    Dumps a VTU file containing velocity and vorticity to FILENAME
-w, --debug-vorticity
    Imposes artificial (sinusoidal) velocity field
    and dumps debugging mesh to vorticityDebugMesh.vtu
```

9.3.3.41 vtk_projection

`vtk_projection` projects the co-ordinates of an input `vtu` to a specified type of output co-ordinates. It is run from the command line:

```
vtk_projection[OPTIONS] -I <in-coordinates> -O <out-coordinates> infile.vtu
```

The `OPTIONS` are:

```
-h, --help
    Print this usage information

-I, --incoords
    Coordinates of input file. Valid types are:
    type      | description
    -----
    cart       | Cartesian (meters)e
    spherical  | Longitude/Latitude
    stereo     | Stereographic Projection

-O, --outcoords
    Coordinates of output file. Valid types are:
    type      | description
    -----
    cart       | Cartesian (meters)
    spherical  | Longitude/Latitude
    stereo     | Stereographic Projection

-o, --output=FILENAME.vtu
    File that the result is outputed to.
    **The default is to overwrite the input file**

-v, --verbose
    Be verbose
```

9.3.3.42 vtudecomp

`vtudecomp` decomposes a `vtu` given a decomposed triangle mesh. It is run from the command line:

```
vtudecomp [OPTIONS] MESH VTU
```

with `MESH` the base name of the decomposed triangle mesh and `VTU` the name of the `vtu` file to be decomposed. Running with the option `-h` provides further information on the other options. Note, `genpvtu`, section 9.3.3.13 can be used to create a `pvtu` from the decomposed `vtus`.

9.3.3.43 vtudiff

`vtudiff` generates `vtus` with fields equal to the difference between the corresponding fields in two input `vtus` (`INPUT1 - INPUT2`). The fields of `INPUT2` are projected onto the cell points of `INPUT1`. It is run from the command line:

```
vtudiff [OPTIONS] INPUT1 INPUT2 OUTPUT [FIRST] [LAST]
```

with `OUTPUT` the name of the output `vtu`. If `FIRST` is supplied, treats `INPUT1` and `INPUT2` as project names, and generates a different `vtu` for the specified range of output files `FIRST - LAST`. If not supplied `LAST` defaults to `FIRST`. The option `-s` if supplied together with `FIRST` and `LAST`, only `INPUT1` is treated as a project name. This allows a range of `vtus` to be diffed against a single `vtu`.

9.3.3.44 vtu_bins

`vtu_bins` returns the fraction of the domain contained within given values for a given scalar field in a `vtu`. `vtu_bins` is built as part of the `fltools` build target (see section 9.3.3), and is used via:

```
vtu_bins [-v] input_filename input_fieldname BOUND1 [BOUND2 BOUND3]
```

where `input_filename` is a `vtu` file containing a scalar field `input_fieldname`, and `BOUND1`, `BOUND2`, ... are the boundary values. The output is sent to standard output. The `-v` flag enables verbose output.

If negative boundary values are required, add two `--`'s before the boundary values on the command line:

```
vtu_bins [-v] input_filename input_fieldname -- BOUND1 [BOUND2 BOUND3]
```

`vtu_bins` requires a linear simplex mesh for the field `input_fieldname`.

```
$ vtu_bins annulus_1564.vtu Temperature 0.0 1.0
          -inf - 0.000000000000000000E+000 :
0.23071069007104538E-004
  0.000000000000000000E+000 - 0.100000000000000000E+001 :
0.99951353813554100E+000
  0.100000000000000000E+001 - inf :
0.46339079545184387E-003
```

Example 9.3: Using `vtu_bins` to compute the volume of under- and over-shoot errors in a DG annulus simulation.

9.4 The stat file

The `stat` file contains information about the simulation, collected at run time. These diagnostics can be extracted from it using the `stat_parser`, section 9.3.2 and it can be quickly and easily visualised with `statplot`, section 9.3.3.1. Note, for parallel runs, unless otherwise stated the values have been calculated in a 'parallel-safe' manner.

The diagnostics that are recorded in the `stat` file for each field are selected by the user. What is included should be considered carefully as including a lot of information can make a notable increase in the simulation run time. To configure the `stat` file locate the `.../stat` element for a generic `scalar_field`, `vector_field` or `tensor_field`, e.g. Figure 9.4. This contains further elements

that will allow the configuration of the stat file as outlined in table 9.4. Diagnostics that are more involved and require a longer description are listed in table 9.4 and documented in section 9.4.3.

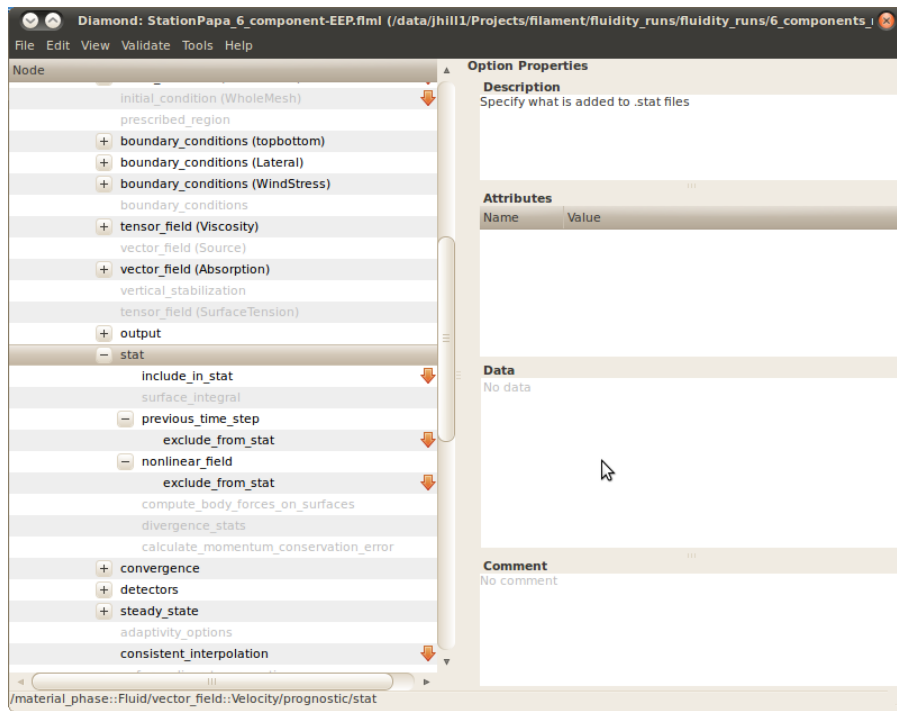


Figure 9.4: Example configuration of the stat file for `.../vector_field(Velocity)`.

The diagnostics will be output at every time step, from the end of the first time step onwards and, where relevant, are output before the mesh is adapted. The following options, regarding when the diagnostics are output, may also be chosen by activating the following elements in the Diamond file (the names are self-explanatory):

- `/io/stat/output_at_start`
- `/io/stat/output_before_adapts`
- `/io/stat/output_after_adapts`

It is also possible to include the values from the previous time step and nonlinear iterations for a vector field by picking the choice elements:

- `.../stat/previous_time_step/include_in_stat`
- `.../stat/nonlinear_field/include_in_stat`

and for a scalar field by activating the elements:

- `.../stat/include_nonlinear_field`
- `.../stat/include_previous_time_step`

9.4.1 File format

There are two file formats used for storing data in `.stat` files: a plain text format and a binary format.

9.4.1.1 Plain text .stat file format

A plain text .stat file consists of two sections: an XML header section and a data section. The header section appears at the start of the .stat file within `<header> ... </header>` tags, and defines all meta data concerning all statistics contained in the data section. The data section contains a number of lines, with each line containing a single data point for each of the statistics defined in the header section.

The header element contains `<constant>` and `<field>` child elements. The `<constant>` elements contain data relevant to the entire simulation, such as the Fluidity version and simulation start date:

```
<constant name = "field_name"
  type = "field_type" value = "field_value"/>
```

where the `type` attribute defines the data type (one of "string", "integer" or "float"). A plain text .stat file defines one additional constant element:

```
<constant name="format" type="string" value="plain_text"/>
```

The `<field>` elements contain meta-data for statistics contained in the data section:

```
<field name = "field_name"
  statistic = "statistic_name" column = "field_column"
  [material_phase = "material_phase_name"]
  [components = "field_components"]/>
```

For statistics of scalar, vector or tensor fields, `statistic` defines the statistic name for a field name in material / phase `material_phase`. For other objects (such as mesh statistics) the `material_phase` attribute may not be supplied. The `column` attribute is an integer defining the index of the first component of the field data in each line of the data section. The optional `component` attribute defines the number of field components (and defaults to one if not supplied).

```
<header>
<constant name="FluidityVersion" type="string" value="11780M" />
<constant name="CompileTime" type="string" value="Nov_25_2009_09:38:20" />
<constant name="StartTime" type="string" value="20091125_095448.326+0000" />
<constant name="HostName" type="string" value="Unknown" />
<constant name="format" type="string" value="plain_text" />
<field column="1" name="ElapsedTime" statistic="value"/>
<field column="2" name="dt" statistic="value"/>
<field column="3" name="ElapsedWallTime" statistic="value"/>
<field column="4" name="CoordinateMesh" statistic="nodes"/>
<field column="5" name="CoordinateMesh" statistic="elements"/>
<field column="6" name="CoordinateMesh" statistic="surface_elements"/>
<field column="7" name="Tracer" statistic="min" material_phase="Fluid"/>
<field column="8" name="Tracer" statistic="max" material_phase="Fluid"/>
<field column="9" name="Tracer" statistic="l2norm" material_phase="Fluid"/>
<field column="10" name="Tracer" statistic="integral" material_phase="Fluid"/>
</header>
1.0 1.0 0.108 41 40 2 0.0 0.0 0.0 0.0
2.0 1.0 0.146 41 40 2 0.0 0.0 0.0 0.0
```

Example 9.4: A simple plain text .stat file

9.4.1.2 Binary .stat file format

The binary .stat file format contains an XML header section stored in a plain text file, and a data section stored in binary in a separate file. The file name of the binary data file is equal to the header file name plus an additional *.dat* file extension. The XML header section is identical to the plain text XML header, except that the *format* constant element is replaced with:

```
<constant name="format" type="string" value="binary"/>
```

The binary .stat file format also defines two additional constant elements:

```
<constant name="real_size" type="integer" value="real_byte_size"/>
<constant name="integer_size" type="integer" value="integer_byte_size"/>
```

defining the size (in bytes) of a real and an integer in the data section.

At present the data section of binary .stat files contains only floating point data.

9.4.2 Reading .stat files in python

A .stat file can be read into a dictionary object using the *fluidity_tools* python module via:

```
import fluidity_tools
stat = fluidity_tools.stat_parser(filename)
```

fluidity_tools.stat_parser reads both plain text and binary .stat file formats. For .stat files using the binary format, *filename* should correspond to the XML header file.

You can find out which fields are contained in a state *Fluid* via:

```
stat["Fluid"].keys()
```

The “max” statistics of a field “Velocity%magnitude” in this state can be plotted via:

```
from matplotlib import pylab
time = stat["ElapsedTime"]["value"]
max_speed = stat["Fluid"]["Velocity%magnitude"]["max"]
pylab.plot(time, max_speed)
pylab.xlabel("Time")
pylab.ylabel("Max_Speed")
pylab.show()
```

9.4.3 Stat file diagnostics

Name	Statistic	Material phase name	Diamond information	Notes
File format information				
format	binary or plain_text		always included	the .stat file will be in plain text unless /io/detectors/binary_output is switched on
real_size	real_size		/io/detectors/binary_output	size of real
integer_size	integer_size		/io/detectors/binary_output	size of an integer
Mesh diagnostics e.g. /geometry/mesh::CoordinateMesh/from_file				
CoordinateMesh	nodes		.../stat/include_in_stat	number of nodes in the mesh
CoordinateMesh	elements		.../stat/include_in_stat	number of elements in the mesh
CoordinateMesh	surface elements		.../stat/include_in_stat	number of surface elements in the mesh
Machine statistics				
FluidityVersion	Fluidity version		always included	Fluidity version
CompileTime	date and time		always included	compile date and time
StartTime	date and time		always included	simulation start date and time
HostName	hostname		always included	name of host machine, default "Unknown"
Memory diagnostics - these are only included if Fluidity is configured with either -enable-debugging or -enable-memory-stats. For parallel runs they are over all processors/				
memory type	current	Memory	n/a	current memory usage
memory type	min	Memory	n/a	minimum memory usage during the last time step
memory type	max	Memory	n/a	maximum memory usage during the last time step
Time diagnostics				
Elapsed time	value		always included	current simulation time
dt	value		always included	time step used for the previous time step
Elapsed wall time	value		always included	how long, in real, wall clock time the simulation has been running
Scalar field diagnostics e.g. for /material_phase::fluid/scalar_field::Temperature/prognostic				

Temperature	min	fluid	.../stat/include_in_stat	minimum of the scalar field
Temperature	max	fluid	.../stat/include_in_stat	maximum of the scalar field
Temperature	l2norm	fluid	.../stat/include_in_stat	L_2 norm of the scalar field over the mesh the scalar field is on
Temperature	integral	fluid	.../stat/include_in_stat	integral of the field over the mesh over the mesh the scalar field is on
Temperature	cv_l2norm	fluid	.../stat/include_cv_stats	L_2 norm of the scalar field over the control volume dual mesh to the mesh the scalar field is on
Temperature	cv_integral	fluid	.../stat/include_cv_stats	integral of the field over the control volume dual mesh to the mesh the scalar field is on
Temperature	surface_integral% name	fluid	.../stat/surface_integral[0]	section 9.4.3.1
Temperature	mixing_bins	fluid	.../stat/ include_mixing_stats[0]	section 9.4.3.2
Vector field diagnostics e.g. for /material_phase::fluid/vector_field::Velocity/prognostic. The values of component will range from 1 to number of dimensions. The force, pressure force and viscous force statistics have not been rigorously tested in parallel				
Velocity%magnitude	min	fluid	.../stat/include_in_stat	minimum of the magnitude of the vector field
Velocity%magnitude	max	fluid	.../stat/include_in_stat	maximum of the magnitude of the vector field
Velocity%magnitude	l2norm	fluid	.../stat/include_in_stat	L_2 norm of the magnitude of the vector field
Velocity%component	min	fluid	.../stat/include_in_stat	minimum of component 1 of the vector field
Velocity%component	max	fluid	.../stat/include_in_stat	maximum of component 1 of the vector field
Velocity%component	l2norm	fluid	.../stat/include_in_stat	L_2 norm of component 1 of the vector field
Velocity%component	integral	fluid	.../stat/include_in_stat	integral of component 1 of the vector field over the mesh
Velocity	surface integral% name	fluid	.../stat/surface_integral[0]	section 9.4.3.1

Velocity	force	fluid	.../stat/ compute_body_forces_on_surfaces	this requires a list of surface ids over which the total force is calculated???
Velocity	pressure force	fluid	.../stat/ compute_body_forces_on_surfaces/ output_terms	pressure force over the surfaces with ids given in the attribute for .../stat/ compute_body_forces_on_surfaces ???
Velocity	viscous force	fluid	.../stat/ compute_body_forces_on_surfaces/ output_terms	outputs the viscous force over the surfaces with ids given in the attribute for .../stat/ compute_body_forces_on_surfaces ???

Table 9.4: Stat file diagnostics

This table contains the 'Name' of the diagnostic, 'Statistic' and 'Material phase name' as they will appear in the stat file, section 9.3.2. 'Diamond information' contains the spud path to locate where the option is in Diamond. Finally 'Notes' offers information about the diagnostic.

9.4.3.1 Surface integrals

The surface integral diagnostics allow the calculation of surface integrated quantities for arbitrary scalar or vector fields. The options can be found for any scalar or vector field at `.../stat/surface_integral`. All surface integral diagnostics are parallelised.

Scalar fields

There are two types of surface integrals for scalar fields, `value` and `gradient_normal`. These are selected in the `type` attribute of `.../stat/surface_integral`.

The surface integral type `value` calculates the surface integral of the scalar field c :

$$\int_{\partial\Omega} c, \quad (9.3)$$

and the surface integral type `gradient_normal` calculates surface integral of the dot product of the gradient of the field with the surface normal:

$$\int_{\partial\Omega} \nabla c \cdot \mathbf{n}. \quad (9.4)$$

The `gradient_normal` surface integral is calculated using the volume element shape function derivatives, via:

$$\sum_{i,j} \int_{\partial\Omega} c_i \frac{\partial \varphi_i}{\partial x_j} n_j. \quad (9.5)$$

Vector fields

There is one type of surface integral for vector fields, `value`. This calculates the surface integral of the dot product of the field with the surface normal:

$$\int_{\partial\Omega} \mathbf{v} \cdot \mathbf{n}. \quad (9.6)$$

Surface integral options

The `name` attribute must be set for all diagnostic surface integrals. In addition to this, surface IDs (see section E.1.4) may be specified at `.../stat/surface_integral/surface_ids`. If specified, the surface integral is computed over just these surfaces. If it is disabled the integral is computed over the whole surface of the mesh. If the element `.../stat/surface_integral/normalise` is activated the integral is normalised by dividing by the surface area.

9.4.3.2 Mixing stats

Mixing stats calculates the volume fraction of the scalar field in a set of ‘bins’ the bounds of which are specified by the user.

The mixing stats can be calculated using the control-volume mesh or for P_1P_1 using the mesh that the scalar field is on. This is specified by setting the choice element under `.../stat/include_mixing_stats[0]` to either `continuous_galerkin` or `control_volumes`. The bin bounds must also be specified. The element `.../stat/include_mixing_stats[0]/mixing_bin_bounds` requires a list of floats that are the values of the bin bounds e.g. if the list reads `0.0 1.0 2.0 3.0 4.0` then 5 bins will be returned with (c : the scalar field):

- size bin 1 = volume fraction of domain with $0.0 \leq c < 1.0$
- size bin 2 = volume fraction of domain with $1.0 \leq c < 2.0$
- ...
- size bin 5 = volume fraction of domain with $3.0 \leq c$

The volume fractions can be normalised by the total volume of the domain by activating the element `.../stat/include_mixing_stats[0]/control_volumes/normalise`. The 'tolerance' beneath the bin bounds for which the scalar field should be included can be specified by activating the element `.../stat/include_mixing_stats[0]/control_volumes/tolerance` (i.e. $1.0 - \text{tolerance} \leq c < 2.0 - \text{tolerance}$). If not selected it defaults to machine tolerance `epsilon(0.0)`. For an example of using the mixing stats see the test case in the Fluidity trunk `.../tests/lock_exchange_2d_cg` or `.../tests/cv_mixing_bin_test_serial/`.

9.4.4 Detectors

The detectors file contains information about the positions of the detectors which is the same for all time steps in the case of static detectors and change at each time step for the Lagrangian detectors as they are advected by the flow. It also contains information about the values of different flow variables at the positions of the detectors. Both, position and variables information are collected at run time.

The user selects which field variables should be included in the detectors file by setting the corresponding option in Diamond. For example, if interested in the value of Temperature at the different detectors positions and at each time step, the option `.../scalar_field/prognostic/detectors/include_in_detectors` should be set. Only the fields of interest should be included since extra information will make the detectors file very large and more difficult to handle. If many detectors are present and/or information from many flow variables is required, it is recommended to set the `/io/detectors/binary_output` in Diamond, since an ascii file will quickly become very large.

The information of the detectors can be extracted with `stat_parser` and it can be visualised with `stat_plot`.

The position of the detectors and value of the selected variables at those positions is saved into the output file at every time step, starting from the end of the first time step and it is done before the mesh is adapted.

9.4.4.1 File format

Similarly to the `.stat` file, the detectors file formats are ascii or plain text format and binary format.

In plain text format, the `.detectors` file contains first an XML header followed by a section with the data. The header is contained within `<header> ... </header>` tags and describes the content of each column in the data section. An example of the header is presented in 9.5.

when having the `/io/detectors/binary_output` option set in Diamond, the header is stored in a plain text file with the `.detectors` extension and the data is stored in binary in another file with `.detectors.dat` extension.

In order to read the `.detectors` files in python, the `fluidity_tools` python module described in 9.4.2 should be used.

```
import fluidity_tools
detectors = fluidity_tools.stat_parser(filename)
```

```

<header>
<field column="1" name="ElapsedTime" statistic="value"/>
<field column="2" name="dt" statistic="value"/>
<field column="3" name="LAGRANGIAN_DET_000001" statistic="position"
  components="3"/>
<field column="6" name="LAGRANGIAN_DET_000002" statistic="position"
  components="3"/>
<field column="9" name="LAGRANGIAN_DET_000003" statistic="position"
  components="3"/>
...
<field column="819348" name="Temperature"
  statistic="LAGRANGIAN_DET_000001" material_phase="BoussinesqFluid"/>
<field column="819349" name="Temperature"
  statistic="LAGRANGIAN_DET_000002" material_phase="BoussinesqFluid"/>
<field column="819350" name="Temperature"
  statistic="LAGRANGIAN_DET_000003" material_phase="BoussinesqFluid"/>
...
<field column="983217" name="Velocity" statistic="LAGRANGIAN_DET_000001"
  material_phase="BoussinesqFluid" components="3"/>
<field column="983220" name="Velocity" statistic="LAGRANGIAN_DET_000002"
  material_phase="BoussinesqFluid" components="3"/>
<field column="983223" name="Velocity" statistic="LAGRANGIAN_DET_000003"
  material_phase="BoussinesqFluid" components="3"/>

```

Example 9.5: An example of the header in .detectors file

As indicated earlier, this python module reads both plain text and binary file formats. For .detectors files using the binary format, `filename` should also correspond to the XML header file.

It is possible to find out which fields are contained in a state Water with:

```
stat["Water"].keys()
```

The Temperature versus time for a particular detector can be plotted with the following python lines:

```

from matplotlib import pylab
time = detectors["ElapsedTime"]["value"]
Temp = detectors["Water"]["Temperature"]["LAGRANGIAN_DET_000001"]
pylab.plot(time, Temp)
pylab.xlabel("Time")
pylab.ylabel("Temperature")
pylab.show()

```


Chapter 10

Examples

10.1 Introduction

This chapter describes several example problems that cover the various aspects of the functionality of Fluidity. The input files for these examples can be found in separate directories under `<<fluidity_source_path>>/examples/`. Each directory contains a Makefile, that implements the following commands. The examples are run by giving these 3 commands in order.

```
make preprocess
```

Runs any preprocessing steps that are needed for this example, e.g. mesh generation and parallel decomposition.

```
make run
```

Runs the actual example. Some examples run a series with different mesh resolution (`driven_cavity`, `rotating_channel`) or Reynold's number (`flow_past_sphere`).

```
make postprocess
```

Runs any postprocessing step that may need to be done after running fluidity. For example, in some examples plots are created and put in the directory.

For an overview of the estimated run time - using the recommended number of processors - for each example, see table 10.1 Some examples (`backward_facing_step_3d`, `restratification_after_oodc`, `flow_past_sphereRe100`, `flow_past_sphereRe1000`, and `tides_in_the_Mediterranean_Sea`) are set up to run in parallel. These can also be run with a different number of processes by changing `NPROCS`, e.g. to run with 8 processors:

```
make run NPROCS=8
```

or run in serial with:

```
make run NPROCS=1
```

If `NPROCS` is not supplied an error will occur.

10.2 One dimensional advection

10.2.1 Overview

In this test example a very simple one-dimensional problem is studied: the advection of an initial "top hat" distribution of a tracer (see figure 10.2.1). Since this test runs in a short time, it allows users

Example section	Example directory	n	Run time
10.2 One dimensional advection	top_hat/	1	2 min.
10.3 The lock-exchange	lock_exchange/	1	10 min.
10.4 Lid-driven cavity	driven_cavity/	1	7 hr.
10.5 2D Backward facing step (reference)	backward_facing_step_2d/	1	25 min.
10.5 2D Backward facing step ($k-\epsilon$)	backward_facing_step_2d/	1	6 hr.
10.6 3D Backward facing step	backward_facing_step_3d/	8	5hr.
10.7 Flow past a sphere	flow_past_a_sphereRe*/	8	9 hr.
10.8 Rotating periodic channel	rotating_channel/	1	10 min.
10.9 Water column collapse	water_collapse/	1	2 hr.
10.10 The restratification after OODC	restratification_after_oodc/	32	20 hr.
10.11 Tides in the Mediterranean Sea	tides_in.the.Mediterranean.Sea/	64	5 hr.
10.12 Hokkaido-Nansei-Okii tsunami	hokkaido-nansei-okii_tsunami/	4	1.5 hr.
10.13 Tephra settling	tephra_settling/	1	1 hr.

Table 10.1: Estimated run times (wall time) for the examples. For parallel examples this is using the indicated, default number of processes, n . The times are rough estimates and may vary between different systems.

to play around with various settings, e.g. spatial discretisation options, adaptivity and interpolation settings, etc., and quickly see the results of their changes. The challenge in this example is get an accurate answer while keeping the solution conservative and bounded.

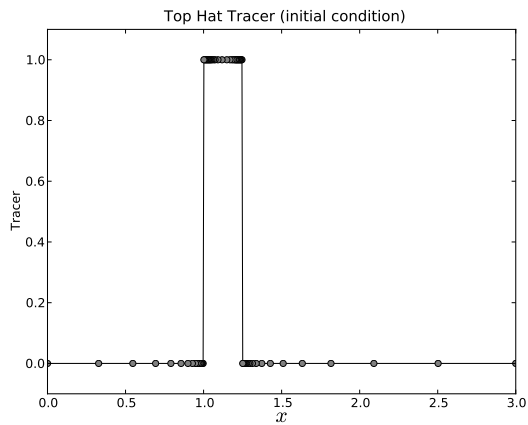
10.2.2 Configuration

The domain is the one-dimensional interval $[0, 3]$. The initial mesh with $\Delta x = 0.025 m$ is created using the “interval” script. The initial “top hat” distribution (figure 10.2.1) is prescribed by a python function. The advective velocity is prescribed as a constant of $u = 0.01 ms^{-1}$. Output is created in the form of a .vtu-file every second of simulation time and a .stat file. The total simulation time is 500 s. The “top-hat” does not reach the boundary in this time, so that boundary conditions play no role in this example.

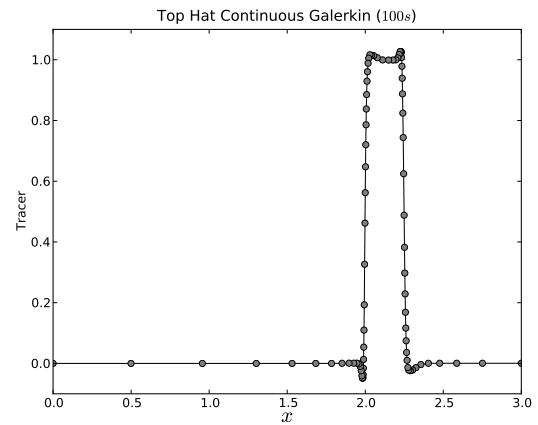
10.2.2.1 Spatial discretisation

This example provides three basic configurations, corresponding to the three main discretisation types of the advection-diffusion equation available in fluidity:

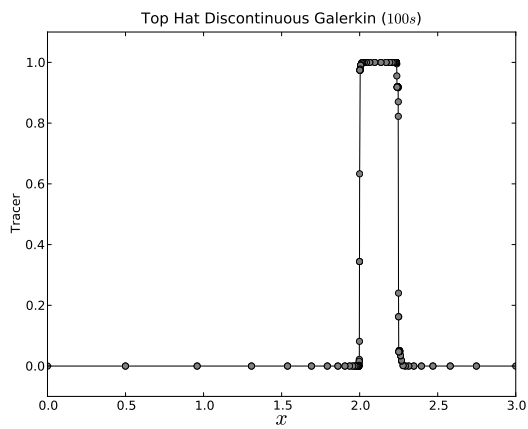
- **Continuous Galerkin (CG)**, section 3.2.1 This is the most basic finite-element discretisation that is simple and fairly efficient. It is however not very good for advection problems of tracers with sharp discontinuities. In the example configuration SUPG stabilisation is applied to improve the results.
- **Discontinuous Galerkin (DG)**, section 3.2.3 This is a popular method for advection problems, in particular for non-smooth tracer solutions. To prevent under and overshoots slope limiters may still be necessary near discontinuities, and are therefore used in this example.
- **Control Volume (CV)** This is a simple and efficient method, that however in some cases can be fairly diffusive. Its accuracy is determined by the choice of interpolation at the control volume faces - a “FiniteElement” interpolation is used here in combination with a “Sweby” slope limiter to keep the solution bounded.



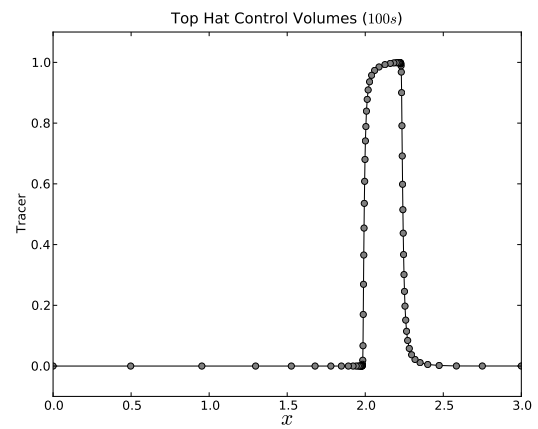
(a) Initial condition of the top hat tracer.



(b) Numerical solution using Continuous Galerkin.



(c) Numerical solution using Discontinuous Galerkin.



(d) Numerical solution using Control Volumes.

Figure 10.1: Initial condition and numerical solutions after 100 s, for the 1D top hat tracer advection problem.

10.2.2.2 Adaptivity and interpolation options

The mesh adaptivity procedure is guided by the metric that is based on the (interpolation) error bounds set for the tracer field. Since at the front and back of the top hat solution the derivatives become very large, this would lead to nearly infinite resolution at these discontinuities. A “MinimumEdgeLength” therefore needs to be set, to prevent such very small elements which would produce very high CFL numbers.

This example uses “`adapt_at_first_timestep`”, so that a suitably adapted mesh is used from the first time step. The initial condition will be directly prescribed on this first adapted mesh and not interpolated from the user provided initial mesh.

Since we want to exactly conserve the amount of tracer in this example a Galerkin projection based interpolation scheme is used to interpolate between subsequently adapted meshes. To keep the solution bounded the “minimally dissipative” bounding procedure is used (only available for CG and DG).

10.2.2.3 Time stepping

Although this example uses a stable implicit time integration scheme (Crank-Nicolson), it is generally desirable for advection problems to control the CFL number. This is done via the adaptive time stepping scheme which dynamically changes the time step based on a desired CFL number - set to 0.5 in the example configurations. Because the example uses a constant velocity, the time step is only determined by the smallest element size which may vary due to the adaptivity process, although it is here limited by the chosen “MinimumEdgeLength”. Since we use an adapted mesh from the first timestep we need the option `at_first_timestep` to use a corresponding adaptive initial timestep.

10.2.3 Results

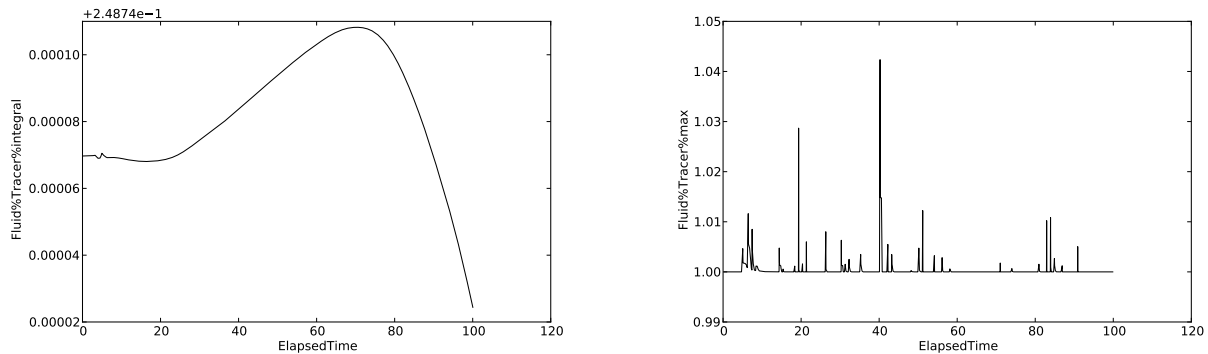
To view the results of this example in paraview, the following steps must be taken:

- Open up the series of vtus in the normal way.
- Add the `Warp (scalar)` filter. Choose a normal direction of 0, 1, 0 and click apply.
- To visualise the grid points: add the `Glyph` filter; Choose `Glyph Type Sphere`, select the `Edit` checkbox next to `Set Scale Factor` and choose a value of 0.02; Click apply.

Alternatively the results can be plotted using python. An example python script using `pylab` and the `vtktools` python module, is provided with this example. The `vtktools` module is located in `<<fluidity_source_path>>/python/`; this path needs to be added to the `PYTHONPATH` environment variable.

As can be seen in the numerical result using the CG method leads, to over and undershoots in the solution. The DG result seems to incur no noticeable bounds violation. The Control Volumes solution is perfectly bounded, but rather diffusive. Note that in each of these cases the initial mesh resolution has changed and is still focussed near the jumps in the solution.

A more precise analysis of the boundedness and conservation properties of the numerical solution, can be derived from the data in the `.stat` file, which can be visualised using the `statplot` program (section 9.3.3.31). As can be seen in figure 10.2.3 the tracer concentration in the CG case is not completely conserved. The CG and CV solutions show perfect conservation (to machine precision). The minimum and maximum (figure 10.2.3) tracer concentration of the DG solution show the occurrence of overshoots with the same frequency as the adapts. This is caused by the fact that the Galerkin projection used to interpolate during the mesh adaptation stage is not bounded. The slope limiting applied



(a) Tracer conservation in the Continuous Galerkin solution. (b) Maximum tracer bound in the Discontinuous Galerkin solution.

Figure 10.2: Conservation and bounds checking using statplot.

in the DG advection algorithm however filters these overshoots out again. The Galerkin projection used for CG and CV is combined with a bounding procedure, so that interpolation doesn't add additional bounds violations. In the case of CG however the numerical scheme itself is not bounded.

10.2.4 Exercises

- For the Continuous Galerkin example see what the effect is of the SUPG stabilisation by changing the option `.../spatial_discretisation/continuous_galerkin/stabilisation/streamline_upwind_petrov_galerkin` to `no_stabilisation`
- Change the resolution of the adapted meshes by changing `adaptivity_options/absolute_measure/scalar_field::InterpolationErrorBound` under `.../scalar_field::Tracer/prognostic/` and see what the effect of the option `/mesh_adaptivity/hr_adaptivity/enable_gradation` is.
- A conservative and bounded CV advection scheme that is less diffusive can be achieved by changing the following options:
 - Change `.../scalar_field::Tracer/prognostic/spatial_discretisation/control_volumes/face_value::FiniteElement` to `face_value::HyperC`.
 - Under `.../temporal_discretisation`, change `theta` to 0.0.
 - Under `.../temporal_discretisation/control_volumes` remove `number_advection_iterations` and `limit_theta`, and add `pivot_theta` with a value of 0.0.

10.3 The lock-exchange

10.3.1 Overview

The lock-exchange is a classic laboratory-scale fluid dynamics problem, [Fannelop, 1994, Huppert, 2006, Simpson, 1987]. A flat-bottomed tank is separated into two portions by a vertical barrier. One portion, the 'lock', is filled with the source fluid. This is of different density to the ambient fluid which fills the other portion. As the barrier is removed, the denser fluid collapses under the lighter. Two gravity currents form and propagate in opposite directions, one above the other, along the tank. After an initial acceleration, the gravity current fronts travel at a constant speed until the end walls

exert an influence or viscous forces begin to dominate, [Cantero et al., 2007, Härtel et al., 1999, Huppert and Simpson, 1980]. At the current front a bulbous head may develop and become taller than the trailing fluid. A shear instability can manifest at the density–interface (hereafter interface) between the two fluids, [Turner, 1973], and this leads to the formation of Kelvin–Helmholtz billows that enhance mixing.

The lock-exchange has been the subject of many theoretical, experimental and numerical studies, and the front speed (or Froude number) is commonly calculated, making it an excellent diagnostic for verification of Fluidity [Benjamin, 1968, Kelmp et al., 1994, Härtel et al., 2000, e.g.]. Furthermore the same physical processes that are encountered in gravity currents over a range of scales are incorporated. The lock–exchange, therefore, presents a tractable means of studying the processes involved and contributes to our understanding of real–world flows, such as sediment–laden density currents and oceanic overflows, and their impact.

In this example, Fluidity is used to simulate a lock–exchange and the following functionality is demonstrated:

- 2D flow
- Non–hydrostatic flow
- Incompressible flow
- Boussinesq flow
- Mesh adaptivity

10.3.2 Configuration

The domain and physical parameters are set up after Fringer et al. [2006] and Härtel et al. [2000], table 10.2. The domain is a 2D rectangular box, $0 \leq x \leq 0.8$ m, $0 \leq z \leq 0.1$ m. Initially, dense, cold water of $T = -0.5$ °C fills one half of the domain, $x < 0.4$ m, and light, warm water of $T = 0.5$ °C fills the other half, $x \geq 0.4$ m, figure 10.3. At $t = 0$ s, $\mathbf{u} = \mathbf{0}$ everywhere. A no-slip boundary condition is applied along the bottom of the domain, $\mathbf{u} = \mathbf{0}$ at $z = 0$, and a free-slip boundary condition is applied to the top of the domain and the side walls, $\mathbf{u} \cdot \mathbf{n} = 0$ at $z = 0.1$ m and $x = 0.0, 0.8$ m.

The basic choices for the numerical set–up are outlined in table 10.3. They comprise a set of standard options for a buoyancy–driven flow such as the lock-exchange. The mesh is adapted to both the velocity and the temperature fields.

gravitational acceleration (ms^{-2})	g	10
kinematic viscosity (m^2s^{-1})	ν	10^{-6}
thermal diffusivity (m^2s^{-1})	κ	0
thermal expansion coefficient ($^{\circ}\text{C}^{-1}$)	α	10^{-3}
domain height (m)	$H = 2h$	0.1
reduced gravity (ms^{-2})	$g' = g \frac{\rho_1 - \rho_2}{\rho_0} = -g\alpha(T_1 - T_2)$	10^{-2}
buoyancy velocity	$u_b = \sqrt{g'H}$	$\sqrt{10^{-3}}$
Grashof number	$Gr = \left(\frac{h\sqrt{g'h}}{\nu} \right)^2$	1.25×10^6

Table 10.2: Physical parameters for the lock-exchange set-up.

10.3.3 Results

Note, the example is set up for a quick start. Many of these results require the simulation to be run for a longer time.

Numerical component	Configuration	Section
Geometry	from triangle file, via gmsh	E.2, 6.7.2
Time step	0.025 s	
Time loop	2 non-linear Picard iterations	3.3
Equation of state	linear	2.3.3
Momentum and Pressure spatial discretisation	P_1P_1	3.5, 3.6, 8.7.1.1, 8.8
Momentum temporal discretisation	$\theta = 0.5$ (Crank-Nicolson)	3.3.3, 8.7.2
	non-linear relaxation term = 0.5	3.3.2
Temperature advection	advection-diffusion equation	2.2
	control volumes	8.7.1.2
Temperature temporal discretisation	$\theta = 0.5$ (Crank-Nicolson)	3.3.3, 8.7.2
	3 advection iterations, limit theta	3.4.2

Table 10.3: Numerical configuration for the lock-exchange

The expected dynamics of a lock-exchange flow are observed, figure 10.3: two gravity currents propagate in opposite directions with the foremost point of the no-slip front raised above the lower boundary. Kelvin–Helmholtz billows form at the interface enhancing the mixing of the two fluids which would not be observed with a hydrostatic formulation. The mesh adapts well, increasing the resolution around the interface and Kelvin–Helmholtz billows with anisotropic elements. Similar images can be generated by visualising the vtu files using Paraview, see the [Cook Book](#) for more information.

Both the gravity current front speeds and the mixing are considered. First the Froude number, $Fr = U/u_b$, which is the ratio of front speed, U , to the buoyancy velocity, u_b , table 10.2 is calculated. The speed with which the no-slip and free-slip fronts propagate along the domain, U_{ns} and U_{fs} , are calculated from the model output and are used to give the corresponding no-slip and free-slip Froude numbers, Fr_{ns} and Fr_{fs} , figure 10.4. For the basic set up given in the example the values are comparable to but generally smaller than previously published values. Spatially varying the horizontal velocity adaptivity settings, as suggested in the exercises, section 10.3.4, can increase the Froude number, showing good agreement between the Fluidity values and previously published values [Hiester et al., 2011]. Changing the metric, as suggested in the exercises, can also increase the Froude number.

The second diagnostic is the background potential energy, which can be used to assess the mixing. The background potential energy is the potential energy of the system reference state [Winters et al., 1995, Winters and D’Asaro, 1996]. The reference state is obtained by adiabatically redistributing the fluid elements into the minimum energy state and is described spatially by an isopycnal coordinate, z_* . The background potential energy, E_b , is then calculated from

$$E_b = \int \rho(z_*)gz_* dz_*, \quad (10.1)$$

where g is gravity and ρ the density. Most crucially, for a closed system, the background potential energy can only be altered by diapycnal mixing and increases in the background potential energy correspond to mixing in the system. The reference state is calculated using the method of Tseng and Ferziger [2001] which uses a probability density function to calculate the value of z_* associated with a given ρ (or here temperature). Here the probability density function is obtained from a set of ‘mixing bins’. Each mixing bin contains the fraction of fluid in the domain that has temperature within a given range or ‘class’. A set of mixing bins, the reference state and background potential energy are presented in figure 10.5. As the simulation progresses, the mixing and the amount of mixed fluid increases, more rapidly at first as Kelvin–Helmholtz billows form and just after the fronts hit the end walls. Later, as the dynamics become less turbulent and the fluid oscillates back and forth in the tank the mixing is less vigorous. The increase in mixed fluid is compensated for by a decrease in non-mixed fluid.

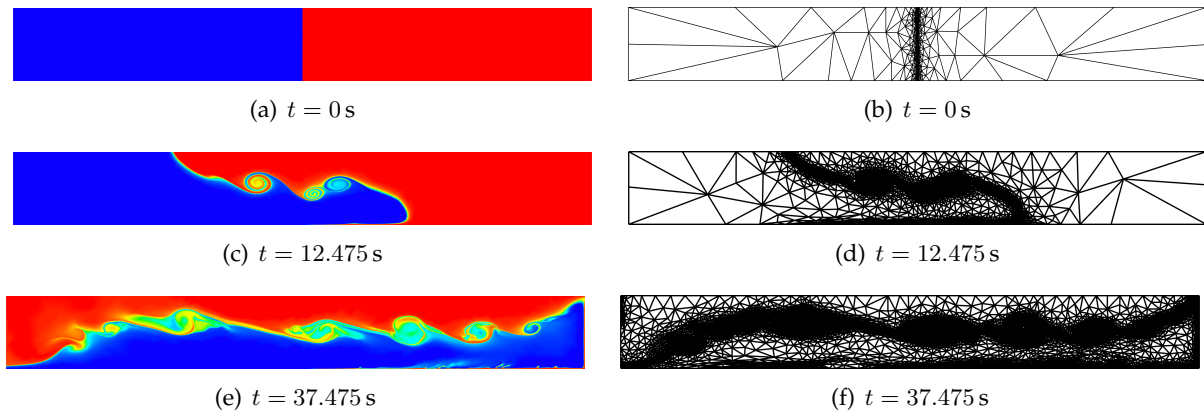


Figure 10.3: Lock-exchange temperature distribution (colour) with meshes, over time (t)

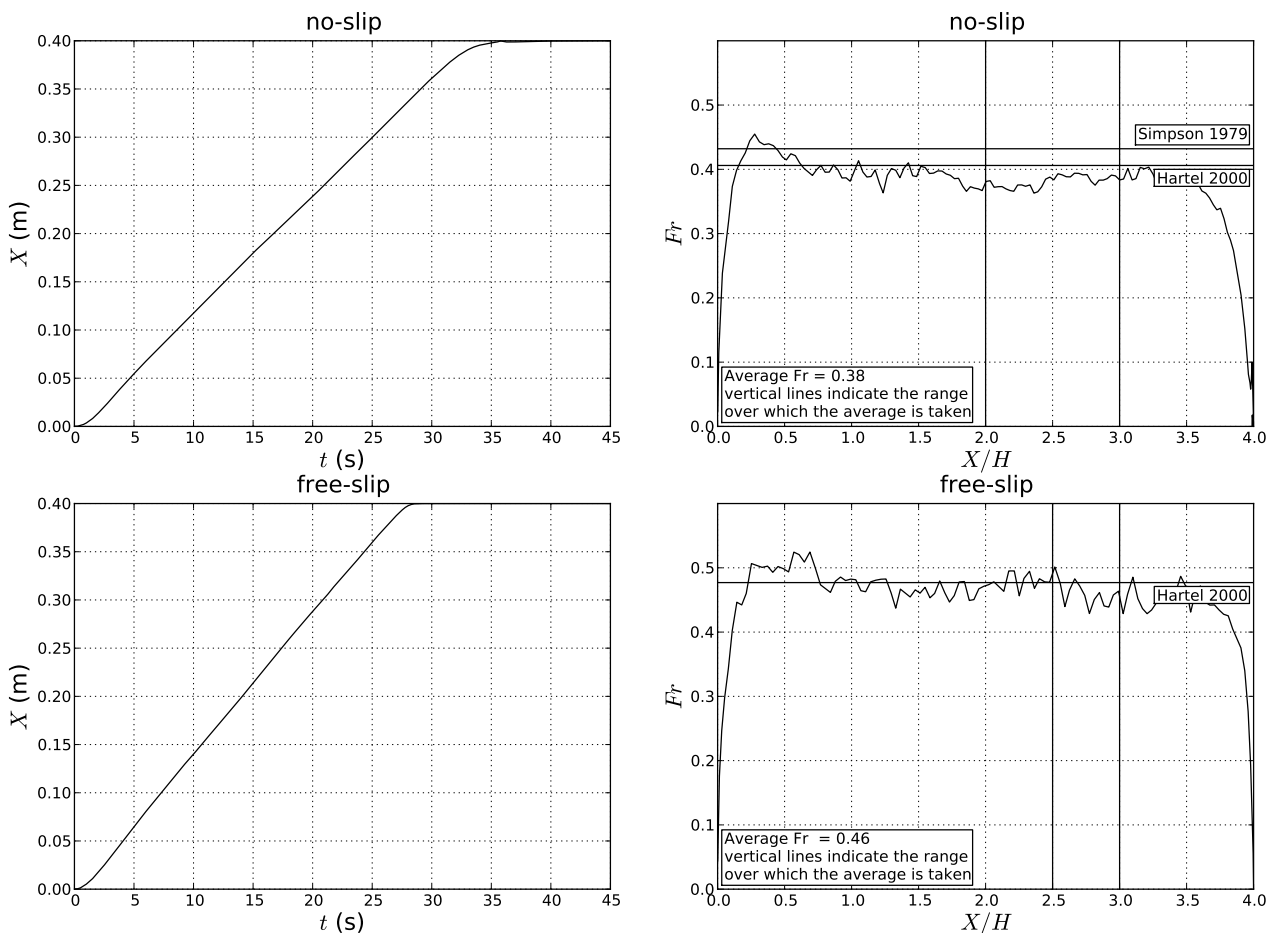


Figure 10.4: Distance along the domain (X) and Froude number (Fr) for the no-slip and free-slip fronts in the lock-exchange. The values of Härtel et al. [2000] and Simpson and Britter [1979] are included for reference.

10.3.4 Exercises

The example settings are designed to provide a quick start for running the simulation. To explore the diagnostics and functionality of Fluidity, the following variations on this example would be constructive exercises:

- Run the simulation for a longer time period by increasing the finish time to 30.0 s to look at the

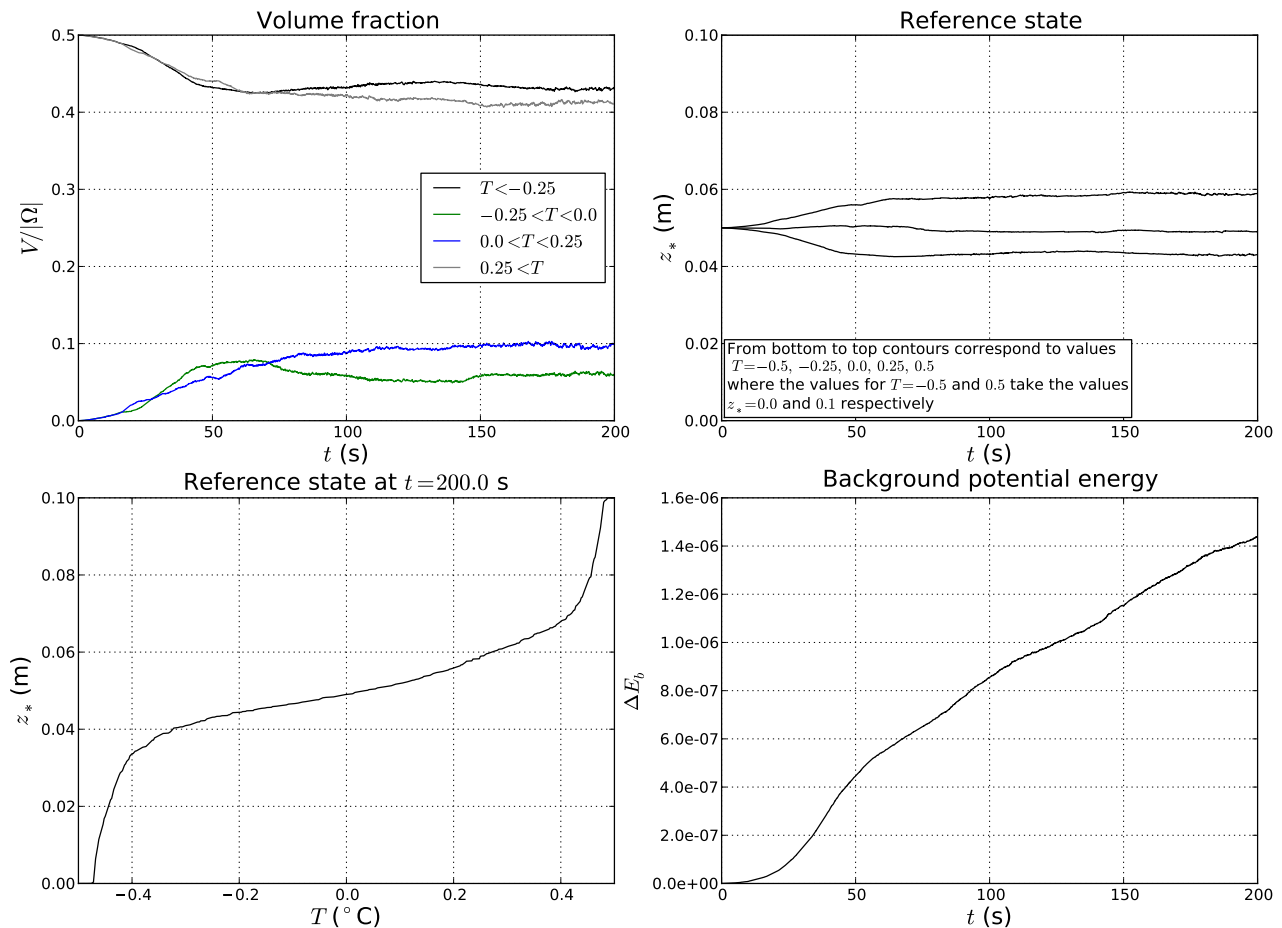


Figure 10.5: Mixing diagnostics for the lock-exchange. As time progresses mixing takes place leading to an increase in the volume fraction of mixed fluid, a spreading of the z_* contours and an increase in the background potential energy. From top left to bottom right: volume fraction of the domain with fluid in the given temperature classes. The classes with ranges $-0.25 < T < 0.0$ and $0.0 < T < 0.25$ are considered representative of the mixed fluid; evolution of contours of z_* for different values of temperature; snapshot of the reference state at $t = 200$ s; the change in the background potential energy, ΔE_b over time.

Froude number and calculate an average value;

- Run the simulation from the checkpoint for a longer time period (until $t = 200.0$ s) to look at the mixing later in the simulation, section 8.3.4.10. When running from the checkpoint don't forget to rename the vtus with `rename_checkpoint.py`, table 9.2;
- In the adaptivity settings for the horizontal velocity field try using a spatially varying interpolation error and compare Froude numbers (python can be found in the comment section for the velocity adaptivity settings), section 8.18.1.2;
- In the adaptivity settings for both the velocity field and temperature field use a p -norm with $p = 2$, section 8.18.1.2. Suggested values for the interpolation error are 0.00005, 0.00005 and 0.0005 for the horizontal velocity, vertical velocity and temperature fields respectively.
- Change the frequency of the adapt, section 8.18.2;
- Turn on metric advection, sections 7.5.7 and 8.18.2.3;

- Change the diffusivity and viscosity values;
- Run with a fixed mesh (note this will require making a new input mesh), sections E.2, 6.7.2;
- Try adding some detectors to visualise the particle trajectories, section 9.4.4

Note to compare Froude numbers and mixing between different runs the remember to copy the images and stat files otherwise they will be overwritten.

10.4 Lid-driven cavity

10.4.1 Overview

The lid-driven cavity is a problem that is often used as part of the verification procedure for CFD codes. The geometry and boundary conditions are simple to prescribe and in two dimensions there are a number of highly accurate numerical benchmark solutions available for a wide range of Reynolds numbers [Botella and Peyret, 1998, Erturk et al., 2005, Bruneau and Saad, 2006]. Here the two-dimensional problem at a Reynolds number of 1000 is given as an example.

10.4.2 Configuration

The problem domain is $(x, y) \in [0, 1]^2$ and this is represented by an unstructured triangular mesh of approximately uniform resolution. To enable convergence analysis a sequence of meshes are considered with spacing of $h = 1/2^n$, $n = 4, 5, 6, 7$. The two-dimensional incompressible Navier-Stokes equations are considered with unit density.

No-slip velocity boundary conditions are imposed on the boundaries $x = 0, 1$ and $y = 0$, and the prescribed velocity $u = 1, v = 0$ are set on the boundary $y = 1$ (the ‘lid’).

Here the problem is initialised with a zero velocity field and the solution allowed to converge to steady state via time-stepping. The steady state convergence criterion is that in the infinity norm the velocity varies by less than 10^{-6} between time levels.

The time step is constant between all runs, $\Delta t = 0.05$, and data is dumped to disk every 1.0 time units (20 time steps). The Crank-Nicolson ($\theta = 0.5$) discretisation is used in time. A P_1P_1 Galerkin method is used for the discretisation of velocity and pressure, with no stabilisation of velocity. Based on a domain size of $L = 1.0$ and a velocity scale taken from the lid ($U = 1.0$), to achieve a Reynolds number of 1000 viscosity is taken to be isotropic with a value of $\nu = 0.001$.

10.4.3 Results

Plots of the streamfunction and vorticity from the $h = 1/128$ simulation are shown in figure 10.6. Observe the good qualitative agreement with [Botella and Peyret, 1998, Erturk et al., 2005, Bruneau and Saad, 2006]. For a quantitative comparison we take advantage of the tabulated benchmark data available in these papers. Only a subset of these are computed: the u velocity at a series of points along the line $x = 0.5$ and the v velocity at a series of points along the line $y = 0.5$ from [Erturk et al., 2005], the same quantities at different points along the same lines as well as the pressure along both the $x = 0.5$ and $y = 0.5$ lines from [Botella and Peyret, 1998], and the kinetic energy and the minimum streamfunction value from [Bruneau and Saad, 2006]. The RMS difference in the case of the first six sets of benchmark data are taken with values extracted from the numerical solution, and the absolute difference taken in the final two. These eight error values are defined as error1, ...error8 respectively and plotted for the four mesh resolutions in figure 10.7. Second order

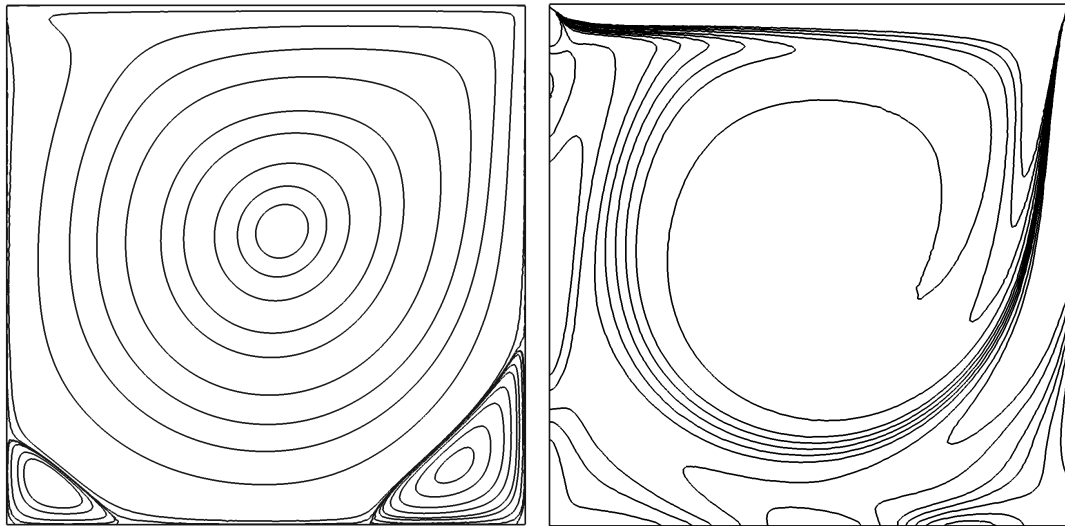


Figure 10.6: Diagnostic fields from the lid-driven cavity problem at steady state at 1/128 resolution. Left: the streamfunction. Right: the vorticity. The contour levels are taken from those given by Botella and Peyret [1998] in their tables 7 and 8.

spatial convergence can clearly be seen. Note that when reproducing these results, subtle changes to the mesh usually result in slight variations to the calculated errors and hence a different plot to that of figure 10.7 will result. The order of convergence, however, should be the same. Adaptive refinement is not particularly advantageous for the problem at this reasonably low Reynolds number, but yields significant improvements in efficiency at higher Reynolds number where boundary layers and recirculating eddies are more dynamic, anisotropic and smaller in size compared to the entire domain.

10.4.4 Exercises

1. Examine the way that the $u = 1$ lid condition is applied in the flml file. Why has it not been set as simply a constant? Try changing it to a constant and see what happens to the errors that you achieve. [Hint: some people consider the “regularised” lid-driven cavity problem. Try finding some papers that discuss this, and update the boundary condition so it matches the regularised problem. Compare to benchmark data if you can find it].
2. The references given above include data from other Reynolds numbers. Try updating the problem set-up and the post-processing script which computes the errors for a higher Reynolds number.
3. Try switching on mesh adaptivity, see section 7.5 (you will need to ensure that you have configured your Fluidity executable with `--enable-2d-adaptivity`). Test adapting based on different metrics, e.g. try weighting u , v and p differently and see what meshes you get. Try varying these weights as well as the maximum and minimum allowed element sizes to see how they affect each other and the mesh that results. Can you get a metric that results in a lower error for the same number of nodes compared to the fixed mesh (hint: it may be easier to achieve this at higher Reynolds numbers)?.
4. This example runs the lid-driven cavity problem with four different resolutions. On a 2.4GHz Intel machine, the runtime for the coarse resolution setup is below 15min, the next higher resolution takes about 30min, then 90min and finally about 350min for the highest resolution. To decrease the runtime for the high resolution case, try to run the it on 4 processors and check how much the runtime decreases.

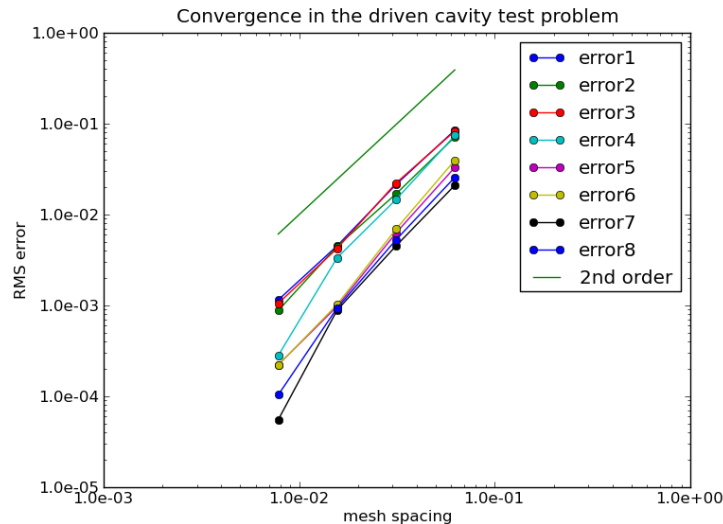


Figure 10.7: Convergence of the eight error metrics computed for the lid-driven cavity problem with mesh spacing. The eight metrics are described in the text.

10.5 2D Backward facing step

10.5.1 Overview

The backward-facing step is a classical CFD example and one of the most frequently selected problems for simulating the separation and reattachment of turbulent flows. It is also often used as a test problem for validating and benchmarking numerical codes. At high Reynolds numbers and in three spatial dimensions the problem has substantial computing requirements, making it an ideal HPC benchmark problem for use here. In the context of ocean modelling flow separation is important in large scale western boundary currents such as the Gulf Stream.

The problem has several important flow characteristics, in particular the downstream length at which the flow reattaches with the bottom of the domain. The reattachment length is considered a sensitive measure of the quality of the numerical method. It is used here to examine the impact of the k-epsilon turbulence model.

Results from Fluidity simulations at Reynolds number 132,000 are presented here. Numerical results using RANS from [Ilinca and Pelletier \[1997\]](#) and experimental results from Kim [Ilinca and Pelletier \[1997\]](#) are used for comparison.

To run the example, use the commands `make preprocess TYPE=type`, `make run TYPE=type` and `make postprocess TYPE=type`, where `type` is one of `reference` or `kepsilon`. `reference` is on a fixed mesh with no turbulence model or stabilisation. `kepsilon` uses the high-Re k-epsilon turbulence model (see [4.1.1.2](#)) on the same fixed mesh.

10.5.2 Geometry

A schematic of the domain is shown in figure [10.8](#). The expansion ratio is 3:2, consistent with [Ilinca and Pelletier \[1997\]](#).

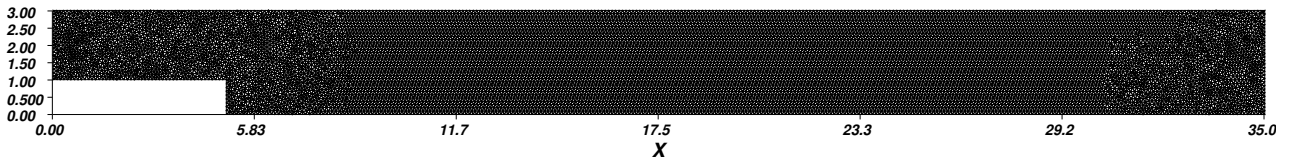


Figure 10.8: Schematic of the domain for the two-dimensional flow past a backward facing step.

10.5.3 Initial and boundary conditions

The inlet velocity profile is a log profile extending reflected in the midpoint of the inlet. Below a small distance $z_0 = 0.1$ from the top and bottom of the inlet, the velocity is 0.

$$u(z) = \begin{cases} 0.0 & \text{if } z \leq 1 + z_0 \\ \log\left(\frac{z-1}{z_0}\right) & \text{if } 1 + z_0 < z \leq 2.0 \\ \log\left(\frac{3.0-z}{z_0}\right) & \text{if } 2.0 < z \leq (3.0 - z_0) \\ 0.0 & \text{if } 3.0 - z_0 < z \end{cases}$$

The log law of the wall is applied weakly on the upper and lower boundaries. This approach has been found useful in high-Reynolds number flow, allowing coarser near-wall mesh resolution.

10.5.4 Results

Three snapshots of the velocity magnitude from the k-epsilon run at $Re = 132000$ are shown in figure 10.9 at 3 times during the flow's evolution to steady-state.

Vertical profiles of u at several points downstream of the step are shown in figure 10.10. The evolution of the flow to the converged solution can be seen. The evolution of the reattachment length is shown in figure 10.11.

The reattachment length was defined here to be the length (normalised by step height $h = 1$) from the step at which the zero-contour of the x -component of u intersects with the bottom boundary. This quantity was computed from u using the VTK library. For the simulation described, the reattachment length converged to approximately 9.1 times the step height, whereas the RANS simulation of Ilinca and Pelletier [1997] reattaches at 6.2 and Kim's experiment at 7.0. The discrepancy may be due to the difference in discretisation or solution procedure from Ilinca.

The solution for the k-epsilon run also contains information on the turbulent kinetic energy, turbulent dissipation and eddy viscosity, which can be plotted in Mayavi or Paraview.

10.6 3D Backward facing step

10.6.1 Configuration

Please note, this 3D example is intended to be run in parallel (see 6.6.4), because it requires relatively fine mesh to resolve the eddies behind the step. More information on running Fluidity in parallel is found in 1.4.2.

The example is run in a very similar way to the other examples. To run in serial, the example is run in exactly the same way to the other examples, using the commands `make preprocess`, `make run` and `make postprocess`. To run in parallel with your chosen number of processors (`np`), using

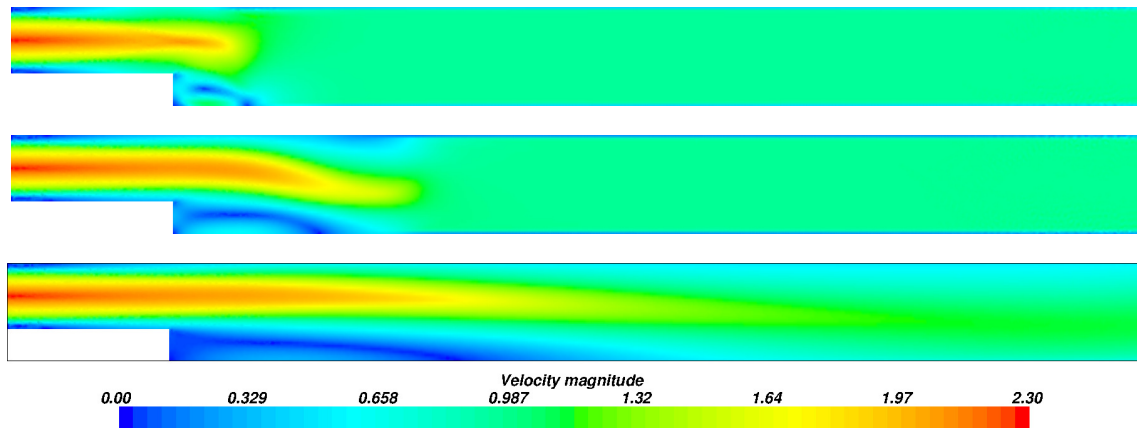


Figure 10.9: Snapshots of the velocity magnitude from the 2D run at times 5, 10 and 50 time units (top to bottom) from the k-epsilon run. The evolution of the dynamics to steady state can be seen, in particular the downstream movement of the streamline reattachment point (where zero-magnitude contour touches bottom).

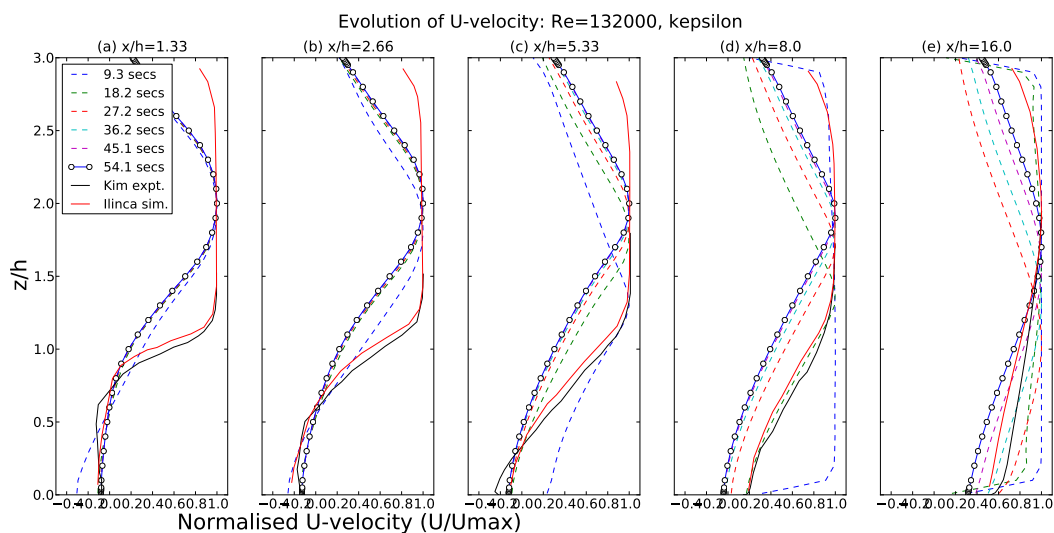


Figure 10.10: Streamwise velocity profiles from the 2D run at $x/h = 1.33, 2.66, 5.33, 8.0$ and 16.0 downstream of the step, where $h = 1$ is the step height. The converged solution is in blue. Ilinca's numerical and Kim's experimental data [Ilinca and Pelletier, 1997] are in red and black respectively. The recirculation region is indicated by negative velocities.

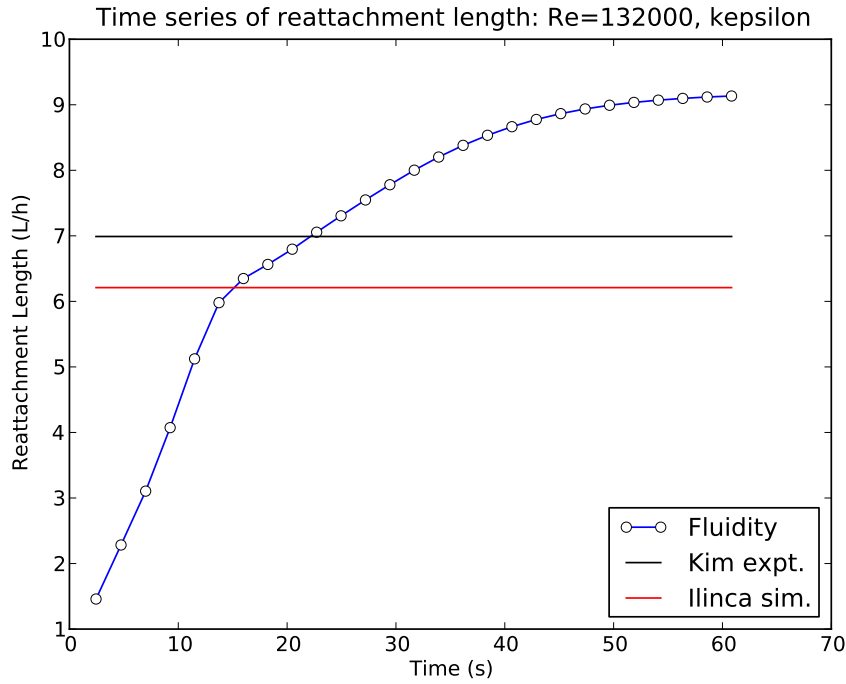


Figure 10.11: Evolution of reattachment length in k-epsilon simulation.

the command `make preprocess NPROCS=np` creates a mesh and decomposes it into (np) parts, and the command `make run NPROCS=np` runs Fluidity as (np) processes on (np) processors. The command `make postprocess NPROCS=np` runs the parallel-specific data processing script.

The Reynolds number of this example is set deliberately low ($Re=155$) in order to allow convergence to a steady state, and demonstrate the reduction in runtime obtained by running in parallel.

10.6.2 Geometry

A schematic of the domain is shown in figure 10.12. A logarithmic velocity profile is imposed at the left hand boundary (inflow). The region directly downstream of the step is of interest in this problem.

Following Le et al. [1997] the dimensions are: $L_x = 30$, $L_i = 10$, $L_y = 4$, $h = 1$, $L_z = 6$, so that $L_z - h = 5$ and the expansion ratio is $L_z/(L_z - h) = 1.2$. The base of the domain is located at $z = 0$, the inflow plane is given by $x = -10$ with the step at $x = 0$, and the back of the domain in the spanwise direction is given by $y = 0$.

10.6.3 Initial and boundary conditions

The inflow boundary condition at $x = -10$ is a log profile given by

$$u(z) = \begin{cases} 0.0 & \text{if } z - h \leq z_0 \\ \frac{u_\tau}{\kappa} \log\left(\frac{z-h}{z_0}\right) & \text{if } z_0 < z - h \end{cases}$$

with parameters $u_\tau = 0.1$, $z_0 = 0.01$ and $\kappa = 0.41$.

No-normal flow, free-stress boundary conditions are applied at the upper and lateral (spanwise)

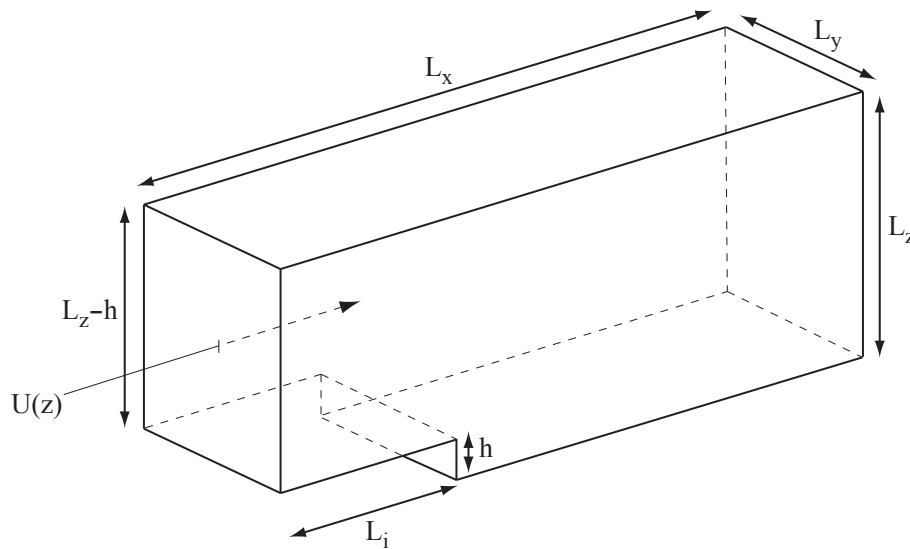


Figure 10.12: Schematic of the domain for the three-dimensional flow past a backward facing step problem.

boundaries:

$$\begin{aligned}
 w = 0, \quad \frac{\partial u}{\partial z} = \frac{\partial v}{\partial z} = 0 & \quad \text{— upper boundary,} \\
 v = 0, \quad \frac{\partial u}{\partial z} = \frac{\partial w}{\partial z} = 0 & \quad \text{— lateral boundaries.}
 \end{aligned}$$

Free-stress boundary conditions are applied at the outflow boundary boundary:

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial x} = \frac{\partial w}{\partial x} = 0.$$

No-slip boundary conditions are applied at the bottom of the domain and at the step down:

$$u = v = w = 0.$$

10.6.4 Results

Three snapshots of the velocity vectors are shown in figure 10.13 at times 5, 10 and 50 seconds. Vertical profiles of u at several points downstream of the step are shown in figure 10.14, in which the velocity data has been averaged across the span of the domain to obtain quasi-2D data. The reattachment point is clearly between $x/h = 6$ and $x/h = 10$. In fact the flow reattaches at $x/h \approx$.

The postprocessing script plots graphs of the reattachment length and velocity profiles, which are compared against the DNS data from Le et al. [1997].

A more rigorous analysis would involve repeating the experiment for a range of Reynolds numbers and comparing the reattachment length of multiple model runs. The Reynolds number in this example is considerably lower than the DNS, in order to reduce simulation run time and suppress turbulent eddies, which results in the differences seen. To get closer to the DNS, try reducing the viscosity from $1.0e - 2$ (Re=155) to $3.04e - 4$ (Re=5100).

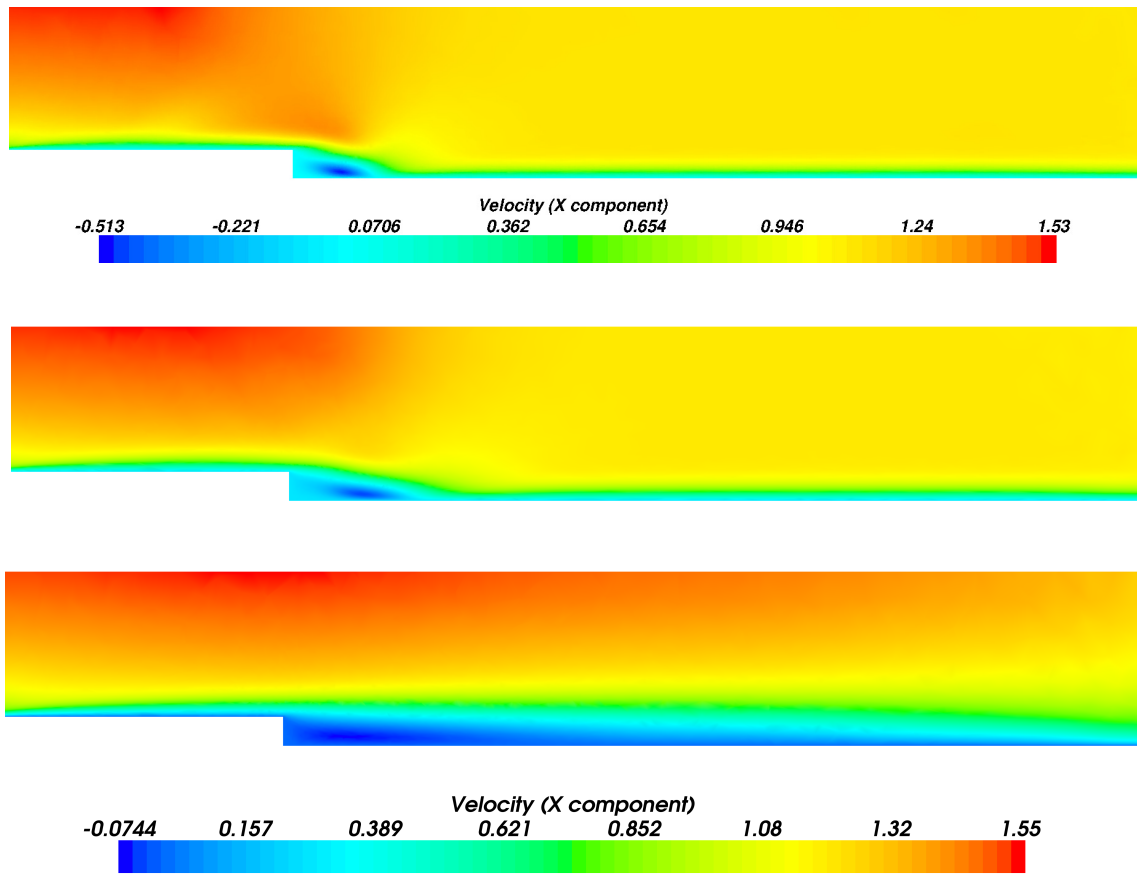


Figure 10.13: From top to bottom: vertical plane cuts through the 3D domain showing the velocity magnitude at times 5, 25 and 50 time units. The evolution of the dynamics to steady state can be seen, in particular the downstream movement of the streamline reattachment point (indicated by contours of $U = 0$).

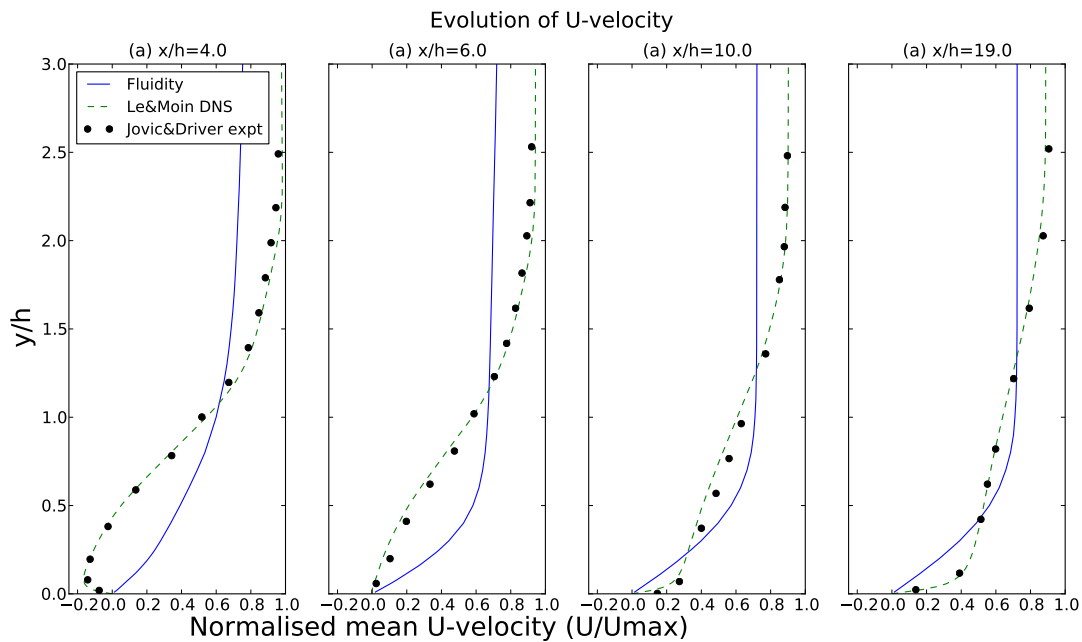


Figure 10.14: Streamwise velocity profiles from the 3d run at $x/h = 4, 6, 10$ and 19 downstream of the step, where $h = 1$ is the step height, at $t = 5$ seconds.

10.7 Flow past a sphere: drag calculation

10.7.1 Overview

In this validation test uniform flow past an isolated sphere is simulated and the drag on the sphere is calculated and compared to a curve optimised to fit a large amount of experimental data.

10.7.2 Configuration

The sphere is of unit diameter centred at the origin. The entire domain is the cuboid defined by $-10 \leq x \leq 20$, $-10 \leq y \leq 10$, $-10 \leq z \leq 10$. GiD is used to mesh the initial geometry.

The unsteady momentum equations with nonlinear advection and viscous terms along with the incompressibility constraint are solved. Free slip velocity boundary conditions are applied at the four lateral boundaries, $u = 1$, $v = w = 0$ is applied at the inflow boundary $x = -10$, and a free stress boundary condition applied to the outflow at $x = 20$.

A series of Reynolds numbers in the range $Re \in [1, 1000]$ are considered. The problem is run for a long enough period that the low Reynolds number simulations reach steady state, and the higher Reynolds number runs long enough that a wake develops behind the sphere and boundary layers on the sphere are formed. This is deemed sufficient for the purposes of this test; the example is to demonstrate how such a problem would be set up, not conduct an in-depth investigation of the physics of this problem.

Here an unstructured tetrahedral mesh is used along with mesh adaptivity. Figure 10.16 shows a snapshot of the mesh and velocity vectors taken from a Reynolds number 1000 simulation. The mesh can be seen to be resolving the wake and the boundary layers on the sphere with enhanced anisotropic resolution. At higher Reynolds numbers the dynamics become more complex and if a full numerical study was being conducted here more care would be taken in the choice of mesh optimisation parameters and the use of averaged values from simulations allowed to run for longer periods. The drag coefficient is calculated from

$$C_D = \frac{F_x}{\frac{1}{2}\rho u_0^2 A}, \quad F_x = \int_S (n_x p - n_i \tau_{ix}) dS, \quad (10.2)$$

where ρ is the density, taken here to be unity; u_0 is the inflow velocity, here unity; and A is the cross-sectional area of the sphere, here $\pi^2/4$. F_x is the force exerted on the sphere in the free stream direction; S signifies the surface of the sphere; n is the unit outward pointing normal to the sphere (n_x is the x -component and n_i the i^{th} component, here summation over repeated indices is assumed); p is the pressure and τ is the stress tensor; see [Panton \[2006\]](#).

10.7.3 Results

Figure 10.15 shows streamlines close to the sphere and the surface mesh. The mesh can be seen to be much finer on the sphere compared to the far wall seen in the background. This is emphasised in figure 10.16 where a more detailed plot of the mesh (with half of the domain removed) over the whole domain and close to the sphere, and the velocity vectors close to the sphere are shown.

Figure 10.17 shows a comparison between the computed drag coefficient with a correlation (to a large amount of laboratory data) taken from [Brown and Lawler \[2003\]](#):

$$C_D = \frac{24}{Re} (1 + 0.15Re^{0.681}) + \frac{0.407}{1 + \frac{8710}{Re}}. \quad (10.3)$$

Excellent agreement can be seen at the range of Reynolds numbers tested in this exercise.

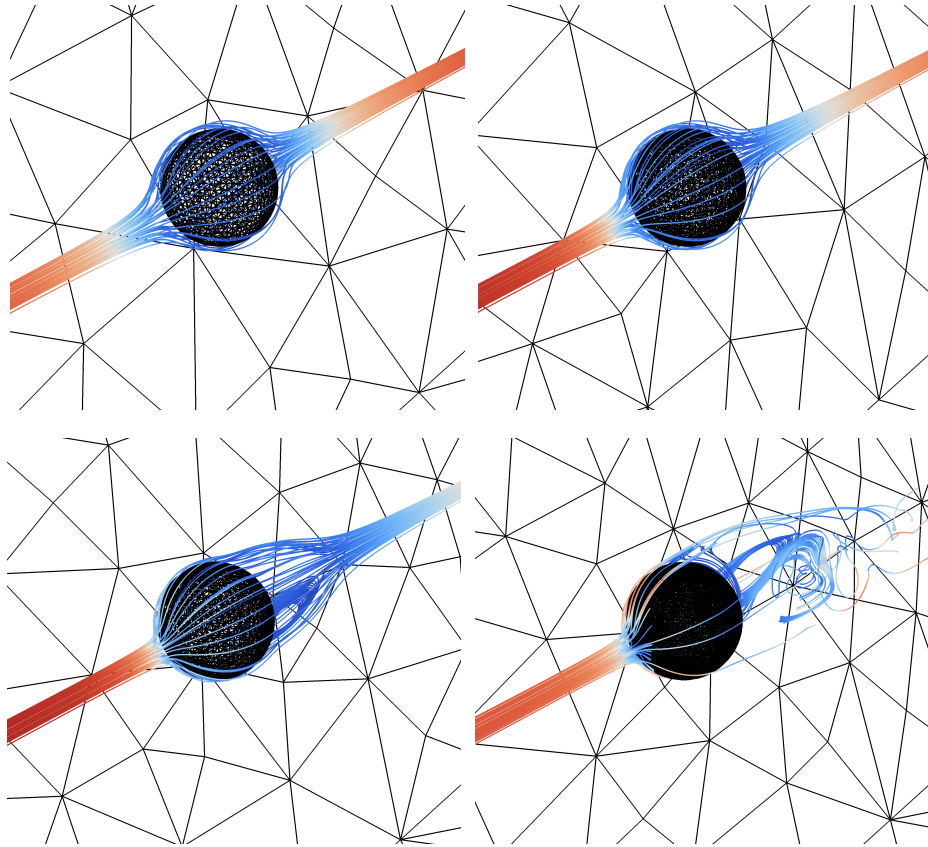


Figure 10.15: Streamlines and surface mesh in the flow past the sphere example. Top-left to bottom-right show results from Reynolds numbers $Re = 1, 10, 100, 1000$.

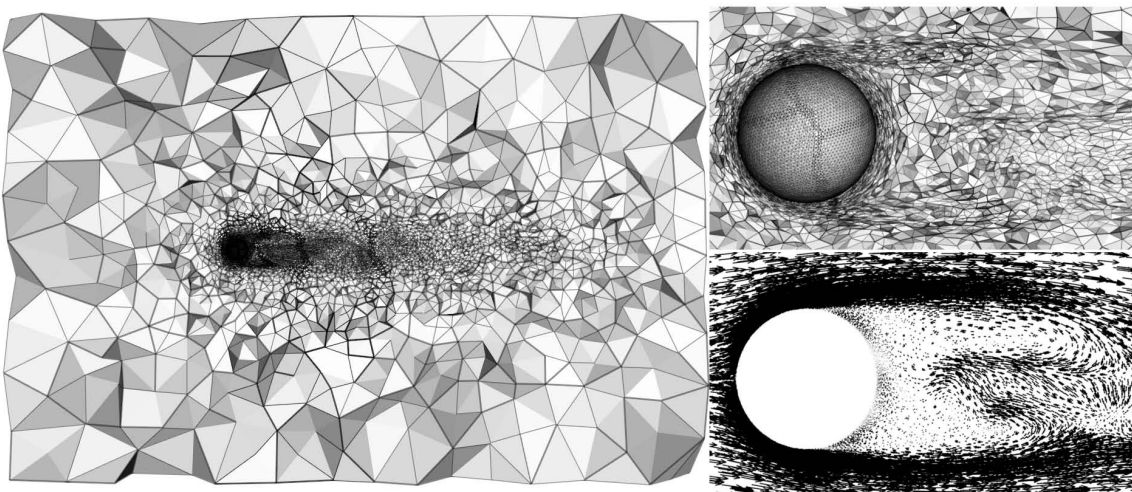


Figure 10.16: Details of the mesh and flow at $Re = 1000$.

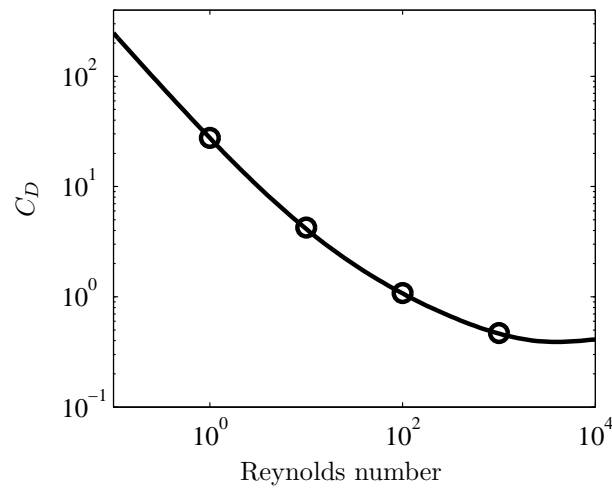


Figure 10.17: Comparison between the numerically calculated drag coefficients (C_D , circles) and the correlation (10.3) (solid line) for $Re = 1, 10, 100, 1000$.

10.7.4 Exercises

1. We actually compute the (vector) force on the sphere and output this to the stat file. This can then be converted to the drag coefficient via (10.2). Write a Python function to do this conversion and the error from the correlation (10.3).
2. Try varying some of the discretisation and adaptivity parameters to see what the impact on the accuracy of the calculated drag is.
3. Try changing the shape of the object, e.g. benchmark data is also available for flow past a cylinder [Schäfer et al., 1996].

10.8 Rotating periodic channel

10.8.1 Overview

This problem provides a convergence test for the $P_{1DG}P_2$ element pair. Utilising almost all the terms of the incompressible Navier-Stokes equations in two dimensions.

The domain is a unit square which is periodic in the zonal direction. The North and South boundaries are zero slip (i.e. $\mathbf{u} = 0$). The remaining parameters are as follows:

Coriolis parameter	f	1
Viscosity	ν	1

The flow is driven by a velocity source term:

$$\mathbf{F} = \begin{bmatrix} y^3 \\ 0 \end{bmatrix} \quad (10.4)$$

So the whole system of equations becomes:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - v = -\nabla p + \nabla^2 u + y^3 \quad (10.5)$$

$$\frac{\partial v}{\partial t} + v \frac{\partial v}{\partial y} + u = -\nabla p + \nabla^2 v \quad (10.6)$$

$$\nabla \cdot \mathbf{u} = 0 \quad (10.7)$$

This system has a steady solution at:

$$\mathbf{u} = \begin{bmatrix} \frac{1}{20}(y - y^5) \\ 0 \end{bmatrix} \quad (10.8)$$

$$p = \frac{1}{120}y^6 - \frac{1}{40}y^2 + C \quad (10.9)$$

Where C is an arbitrary constant resulting from the use of Dirichlet boundary conditions on both the domain boundaries.

10.8.2 Results

Since the maximum velocity in the domain is approximately 0.025, the Reynolds number for this solution is much smaller than 1 so the flow is safely within the laminar regime and will remain steady. Figure 10.18 shows the forcing term for velocity and the analytic solutions for velocity and pressure.

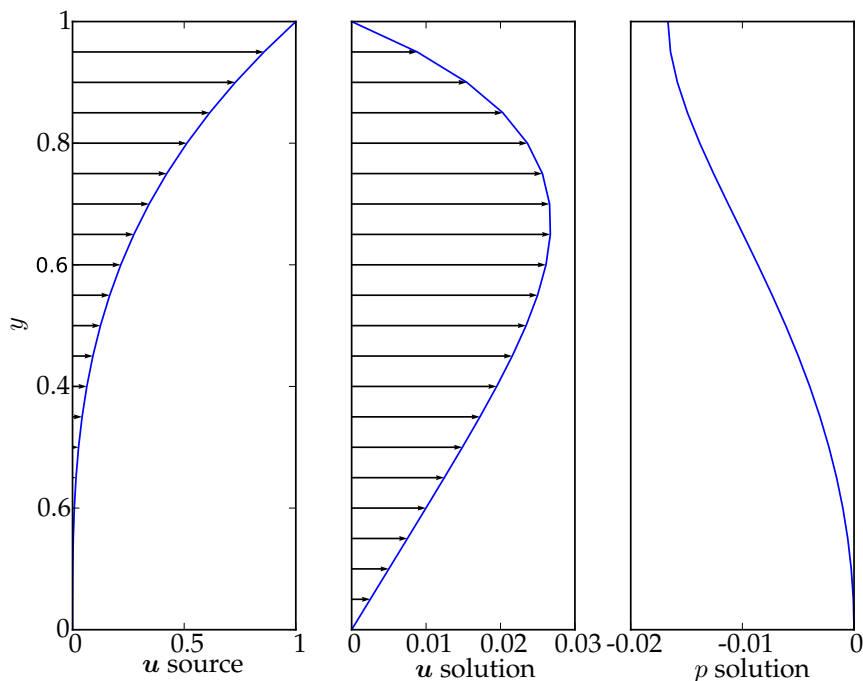


Figure 10.18: Velocity forcing term and analytic solutions for velocity and pressure for the rotating periodic channel test case. Note that each of these quantities is constant in the x direction.

The convergence test is conducted by repeating this simulation on unstructured meshes with typical resolution $1/4$, $1/8$, $1/16$, $1/32$, $1/64$. The results are then compared to the analytic solution. In the case of pressure, the answer is translated to account for the arbitrary constant. Figure 10.19 shows the L^2 error for velocity and pressure. It is apparent that both quantities converge at second order.

10.8.2.1 Exercises

This example can be used to understand the use of analytic forcing functions in Fluidity. Try modifying the function `channel_tools.forcing`. The forcing function and the analytic velocity and pressure results can be visualised by running the `plot_theory` script.

Examine the rest of the `channel_tools` Python module to see how the analytic solution is automatically calculated from the forcing function. Next, examine the

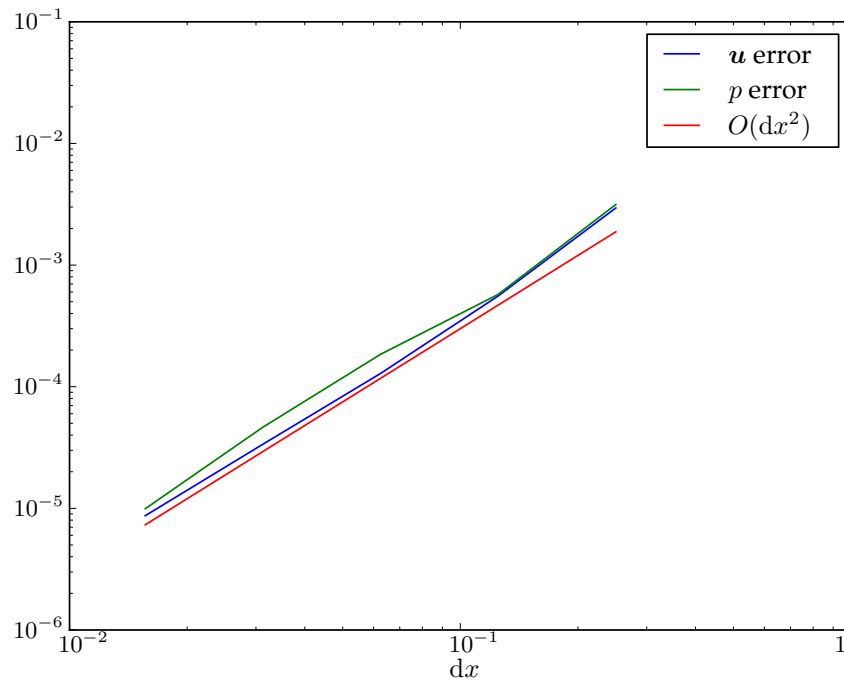


Figure 10.19: Error in the pressure and velocity solutions for the rotating channel as a function of resolution.

`AnalyticUVelocitySolutionError` and `AnalyticPressureSolutionError` to see how this is used to calculate the error in the model solution. The documentation of the Python state interface in appendix B may be useful.

10.9 Water column collapse

10.9.1 Overview

A commonly used validation experiment for multi-material models is that of a collapsing column of liquid, normally water, within an atmosphere or vacuum [Lakehal et al., 2002], also known as the dam break problem. In the experimental set-up a reservoir of water is held behind an impermeable barrier separating it from the rest of the tank. The barrier is then quickly removed, allowing the water column to collapse and flood the remaining sections of the tank. In the numerical analogue the initial condition is generally taken as the trapped water column, still behind the dam. At the start of the simulation the barrier is imagined to have been removed instantaneously and switching on gravity, $|g| = 9.81$, causes the column to collapse. Several experimental set-ups have been published and used as comparison and validation tools for numerical models [Martin and Moyce, 1952, Greaves, 2006]. Those with water depth gauges distributed throughout the tank are particularly useful, allowing the direct comparison of data. Furthermore, pressure gauges located on the tank walls or on any obstacles within the tank provide another useful validation tool.

In this example, Fluidity is used to simulate a simple dam break experiment [Zhou et al., 1999]. The example demonstrates the following functionality:

- 2D flow
- Multi-material flow

- Incompressible flow
- Inviscid flow
- Mesh adaptivity
- Static detectors

The simulation illustrated here took ~ 2 hours to run in serial on a Intel Xeon X5355 2.66 GHz processor.

10.9.2 Problem specification

The experiment on which this example is based was a simple dam break problem in a $3.22 \times 2 \times 1m$ (length \times height \times depth) tank [Zhou et al., 1999]. A reservoir of water $1.2 \times 0.6 \times 1m$ (length \times height \times depth) was held behind a barrier at one end of the tank. Water depth gauges were placed at two points, marked H1 and H2 in Figure 10.20(a), at $x_1 = 2.725m$ and $2.228m$ respectively. Additionally, a pressure gauge was located at the point marked P2 in Figure 10.20(a), at $x_2 = 0.16m$ on the wall facing the initial water column.

As no variations were introduced in the third dimension, the experiment is reproduced here numerically in two dimensions within the domain Ω : $x_1 \in [0, 3.22]$, $x_2 \in [0, 2]$ [Lee et al., 2002, Colagrossi and Landrini, 2003, Park et al., 2009]. The two materials (water and air) are distinguished by scalar fields α_k representing their volume fraction, where the volume fraction of air $\alpha_2 = 1 - \alpha_1$ and α_1 is the volume fraction of water. The initial condition of the water volume fraction is shown in Figure 10.20(a). The presence of water is indicated as a blue region and the interface to air is delineated by contours at volume fraction values of 0.025, 0.5 and 0.975. The densities of the water and air are taken as $1,000kg\ m^{-2}$ and $1kg\ m^{-2}$ respectively. Both fluids are treated inviscidly. As the simulation is inviscid, free slip boundary conditions are imposed on the tank bottom, $x_2 = 0$, and sides, $x_1 = 0, 3.22$. The top of the tank, $x_2 = 2$, is left open.

The water volume fraction, α_1 , is solved for using an advection equation (section 2.2). It is advected using HyperC on a control volume mesh (section 3.2.4.1), while the velocity and pressure are discretised using the P_0P_{1CV} element pair (section 3.7) with $\theta = 1/2$ and $\theta_i = 1/2$ (section 3.3.3).

This example employs an adaptive mesh (chapter 7). The minimum edge length in the mesh is constrained to $3.33mm$. The upper bound on the edge lengths was specified as half the domain length and height in each dimension. The water volume fraction was directly adapted to using an interpolation error bound, $\hat{\varepsilon}$, of 0.075. Given the range of this field seen in the fixed mesh runs this corresponds to a desired error of less than 5%. The volume fraction is transferred between successive meshes using a minimally diffusive bounded projection algorithm. The velocity is transferred using a straightforward projection while the pressure is consistently interpolated using the linear basis functions from its parent mesh. For details of the remaining adaptivity settings we refer to the documented flml file. The initial mesh using these settings is shown in Figure 10.20(b).

The timestep is selected to achieve a Courant number of 2.5 while the advection equation uses approximately 10 subcycles (section 3.4.2) so the volume fraction is advected at a Courant number of 0.25.

10.9.3 Results

Several timesteps of the example simulation can be seen in Figure 10.21 where the interface is represented by contours of the water volume fraction, α_1 , at 0.025, 0.5 and 0.975. Similar images can be generated by visualising the vtu files using Paraview or Mayavi2, see the [AMCG website](#) for more information.

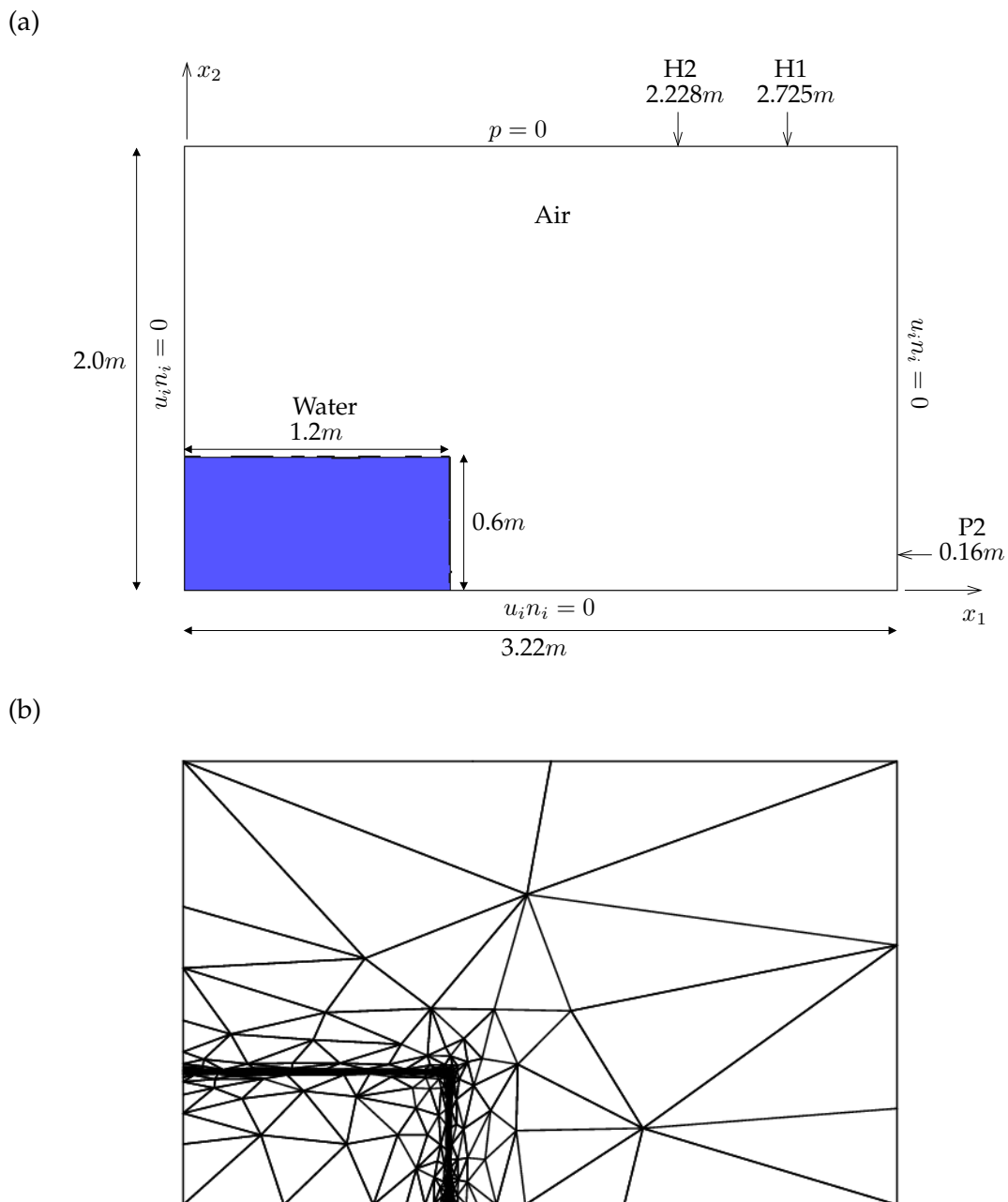


Figure 10.20: (a) Initial set-up of the water volume fraction, α_1 , and the velocity and pressure boundary conditions for the two-dimensional water column collapse validation problem [Zhou et al., 1999]. The presence of water is indicated as a blue region and the interface to air is delineated by contours of the volume fraction at 0.025, 0.5 and 0.975. The locations of the pressure (P2) and water depth gauges (H1, H2) are also indicated. (b) The adapted mesh used to represent the initial conditions.

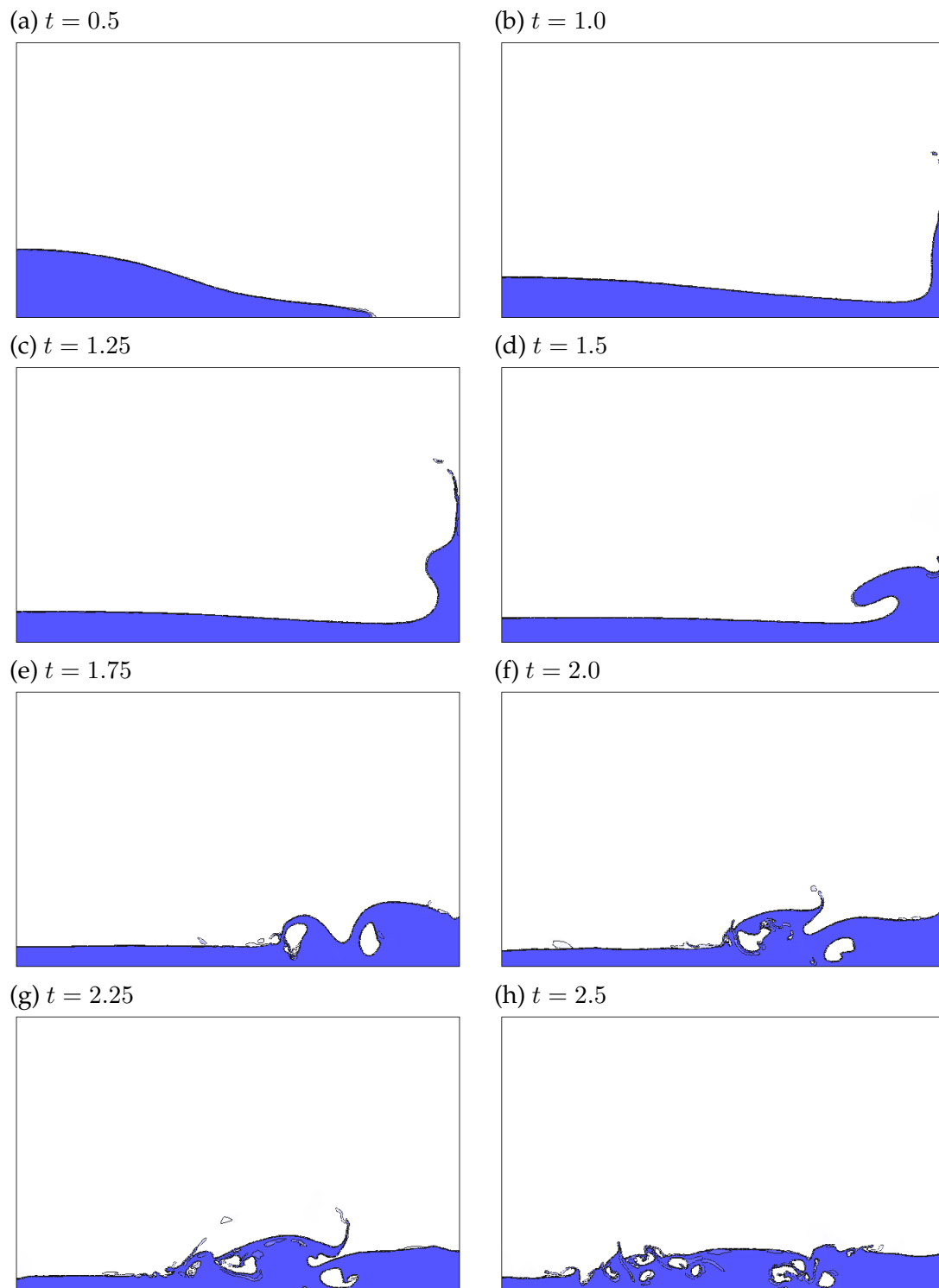


Figure 10.21: The evolution of the water volume fraction, α_1 , over several timesteps. The presence of water, $\alpha_1 = 1$, is indicated as a blue region and the interface to air, $\alpha_1 = 0$, is delineated by contours at $\alpha_1 = 0.025, 0.5$ and 0.975 .

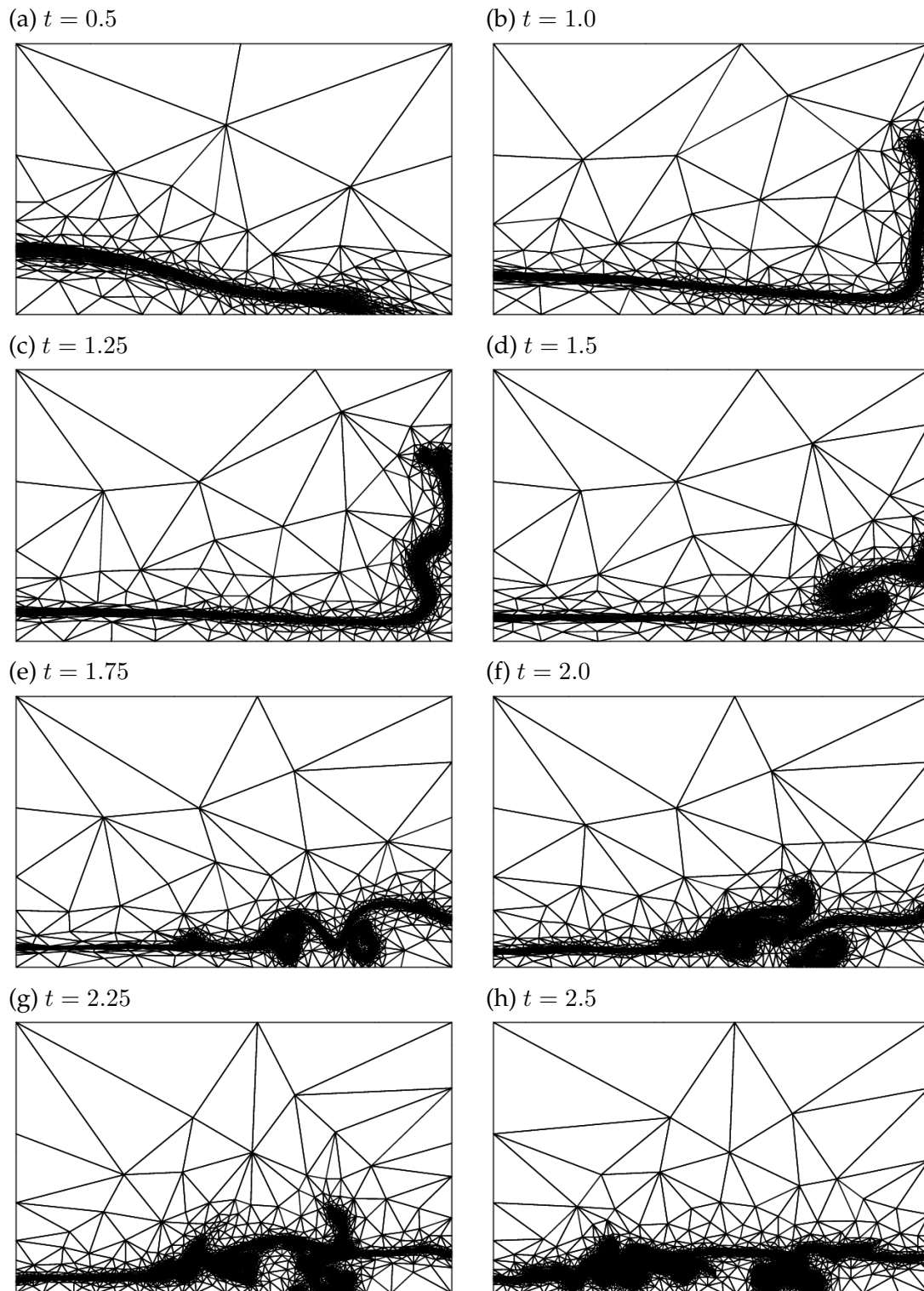


Figure 10.22: The evolution of the adaptive mesh over the same timesteps displayed in Figure 10.21. The mesh can be seen to closely follow the interface between the water and air.

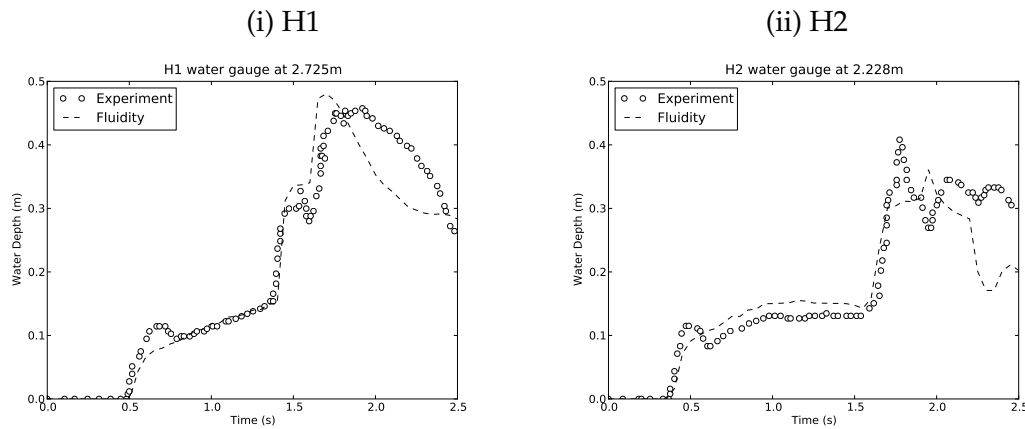


Figure 10.23: Comparison between the experimental (circles) and numerical water gauge data at H1 (i) and H2 (ii), $x_1 = 2.725m$ and $2.228m$ respectively. Experimental data taken from Zhou et al. [1999] through Park et al. [2009].

The images show the column collapse (Figure 10.21(a)), run-up against the opposing wall (Figure 10.21(b, c)) and the subsequent overturning wave (Figure 10.21(d)) and entrainment of air bubbles (Figure 10.21(e-h)). The evolution of the adaptive mesh over the same timesteps is shown in Figure 10.22.

For validation purposes, the post-processing script provided for this example extracts information from the detector file (pressure) and from the water volume fraction field stored in the vtu files (water depth) and plots these data in comparison with the experimental results. The water depth gauge data is displayed in Figure 10.23 alongside the experimental data. The numerical results show the total thickness of water at the points H1 and H2, discounting any air bubbles that cross the gauges. The simulation results show a close similarity to the experimental results with the exception of a small lip of water when the initial water head passes the gauge. It is unclear what causes this structure, though it may be related to the initial withdrawal of the barrier in the experiment or drag effects from the bottom of the tank. All previous published attempts to model the experiment also fail to reproduce this initial lip [Zhou et al., 1999, Lee et al., 2002, Colagrossi and Landrini, 2003, Park et al., 2009].

After $t = 1.5s$ the overturning wave starts to pass the water gauges and the match between the experimental results and the numerical simulation deteriorates. As would be expected from such complex behaviour, all previous published attempts have also failed to reproduce the experimental depth gauge data after this point. However, the broad pattern and average depth observed in the simulation after $t = 1.5s$ can be seen in Figure 10.23 to match the experiment reasonably well.

Experimental pressure gauge data are also available at the point P2, $(3.22, 0.16)m$, on the right wall of the tank. This is compared to the numerical pressure results in Figure 10.24. After the initial noise in the experimental data, a sudden step in pressure is seen as the water run-up reaches the pressure gauge at about $t = 0.6s$. This is also seen in the numerical simulations however it is slightly delayed, occurring at $t = 0.7s$. As upwinding is being used in the discretisation of the velocity field, the delay may be due to numerical viscosity slowing the advancing water front. However, as the delay was not as extreme at the depth gauges H1 and H2 other factors may also play a role. For instance, if the lip seen in the experimental water gauge data is a head on the water front, that has not been reproduced numerically, it may reach the height of the pressure gauge faster than a front with no head.

Once the pressure jump occurs the experimental and numerical data are in broad agreement until the overturning wave impacts with the water layer at approximately $t = 1.5s$ (Figure 10.21(d)). At the point of contact a pressure pulse is transmitted to the pressure gauge resulting in a modest pressure spike in the experimental data. This is matched by slightly delayed pressure pulses in all the numer-

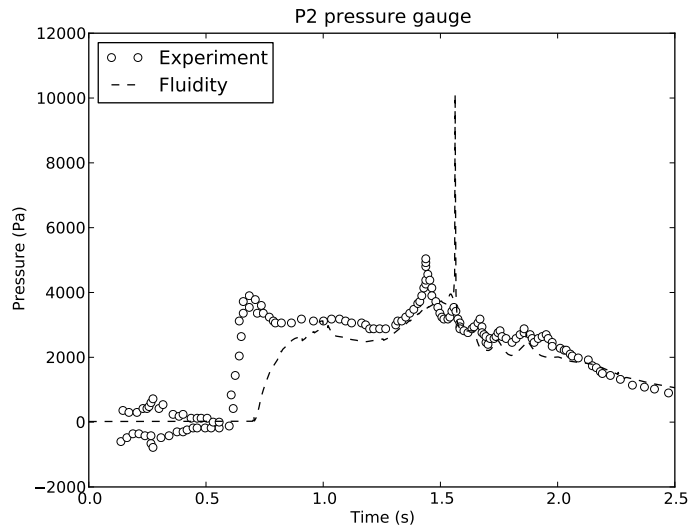


Figure 10.24: Comparison between the experimental (circles) and numerical pressure gauge data at P2, $x_1 = 3.22m$, $x_2 = 0.16m$. Experimental data taken from Zhou et al. [1999] through Park et al. [2009].

ical simulation. However, the pulses seen in the numerical data are of a much larger magnitude than the experimental data. This discrepancy may be due to the fact that in the experiment the pressure gauge measures the pressure over a broader area than in the simulation.

10.9.4 Exercises

To explore the functionality of Fluidity, the following variations on this example would be constructive learning exercises:

- Disable the adaptivity option to run on a fixed mesh
- Alter the water/air viscosity/density
- Modify the tank geometry

10.10 The restratification following open ocean deep convection

10.10.1 Overview

During Open Ocean Deep Convection (OODC), cold water is mixed up to the surface via vigorous convection, forming a column of dense water tens of kilometres across known as a convection chimney. This happens in particular sites of the ocean, including the Labrador Sea, the Mediterranean Sea and the Weddell Sea. In the North Atlantic, the convection typically happens in winter and is triggered by intense cooling at the surface. The convection is important to the formation of North Atlantic Deep Water, which joins the southward part of the Atlantic Meridional Overturning Circulation (AMOC).

It is thought that OODC could be affected by future climate change. If ice caps melt or the hydrological cycle intensifies due to climate change then there would be an influx of fresh water at the surface

of the ocean in the North Atlantic convection sites. This could lead to less NADW forming and a slowing of the AMOC.

This example is an idealised model of the restratification phase of OODC. During restratification the water column formed during OODC mixes back with the surroundings, forming baroclinic eddies due to the coriolis force. The setup is taken from Rousset et al. [2009].

10.10.2 Configuration

The domain is a cylinder of diameter $L = 500\text{km}$ and height $H = 1\text{km}$. The aspect ratio is given by $R = L/H = 500$, which is relatively high. The initial temperature field is shown in figure 10.25. The temperature is linearly stratified except inside the cylinder in the middle with radius 70km , which is cooler than the surroundings.

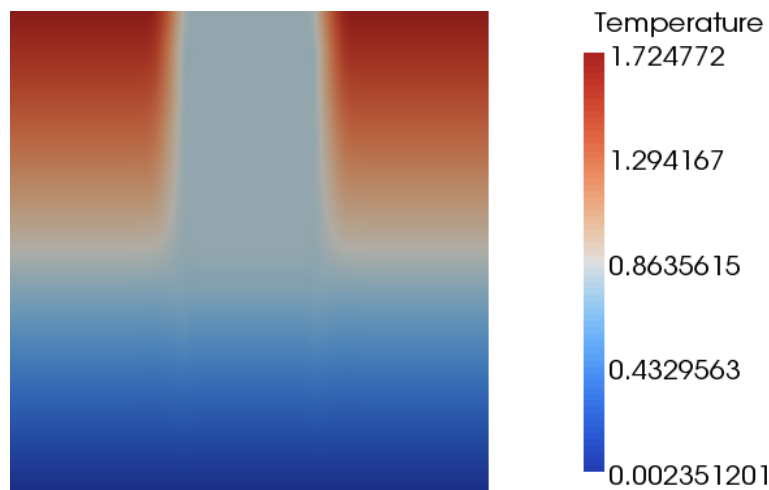


Figure 10.25: A vertical slice through the domain showing the initial temperature stratification. The domain is a cylinder of radius 250 km and height 1 km.

The mesh used is a layered mesh, which is created within Fluidity from the two dimensional input mesh, `circle.geo`. The unstructured triangular mesh is extruded in the vertical, forming triangular prisms which are then divided into unstructured tetrahedra. The columnar arrangement of the nodes is important because the problem has a large aspect ratio, and the elements themselves are wider than they are tall. A fully unstructured mesh would create errors in the pressure, which would then cause errors in the velocity. The layered mesh may be considered to be a special case of the two plus one mesh, in which the nodes are still vertically aligned, but there are different mesh resolutions in different areas of the domain. The resolution is 5 km in the horizontal and 83 m in the vertical.

A $P_{1DG}P_2$ element pair is used. This means that the discretisation is P_{1DG} for velocity and P_2 continuous for pressure. In this case, the temperature is P_1 continuous. The benefits of $P_{1DG}P_2$ are discussed in 3.7. To make it more stable, subcycling is switched on under velocity with a maximum Courant number per subcycle of 0.1. There is a diagnostic free surface field, which requires a free surface boundary condition under the `Velocity` options.

The timestep is 7200 s and there are two non-linear iterations. The timestep is constrained by the Courant condition, but is allowed to be bigger than would otherwise be expected because an absorption field is added under the `Velocity` options. If this field were not added, the time steps would be limited by the scale of the baroclinic waves. This term has a vertical component equal to $1/\rho_0 \theta \Delta t g \frac{\partial \rho}{\partial z}$ and the other components are zero. ρ_0 is the reference density, θ is the value set under `.../Velocity/temporal_discretisation/theta`, Δt is the timestep, g is the acceleration due to gravity and $\frac{\partial \rho}{\partial z}$ is the background density stratification. In this case the absorption term is 0.025

in the vertical and 0.0 in the horizontal. The `.../Absorption/include_pressure_correction` option is turned on.

10.10.3 Results

Figure 10.26 shows the temperature field for a cross section at 40 m depth at days 10, 20, 30 and 40. As the column mixes with the surroundings, eddies form at the perimeter of the cylinder. These eddies play a role in the mixing. If this is run with a different resolution, then a different number of eddies may form. The mixing stats diagnostic allows us to quantify the amount of mixing that takes place. It calculates the volume of fluid that has a value of temperature (or other tracer) within certain user-specified bounds as a function of time, and saves this information in the stat file. Figure 10.27 shows the output from this diagnostic, plotted using the python file provided in the examples directory. The temperature is in units from 0 to 0.72. The 0.7–0.8 bin decreases in volume during the course of the simulation because the cold cylinder is mixing with the surrounding stratified fluid.

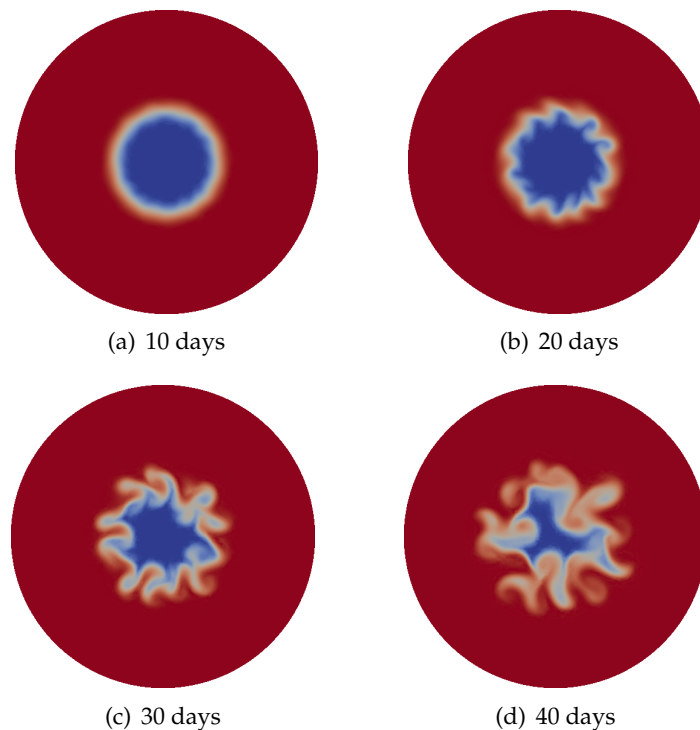


Figure 10.26: The temperature cross-section at a depth of 40m.

10.11 Tides in the Mediterranean Sea

10.11.1 Overview

Tidal modelling is a widely used method for validating free surface implementations [Shum et al., 1997]. The Mediterranean Sea is a good example as it requires both astronomical and co-oscillating boundary tide forcing to obtain an accurate solution [Tsimplis et al., 1995, Wells, 2008]. An abundance of available tide gauge data recording the harmonic constants for both the amplitude and phase of a wide variety of different tidal constituents facilitates comparisons between ICOM and real-world data.

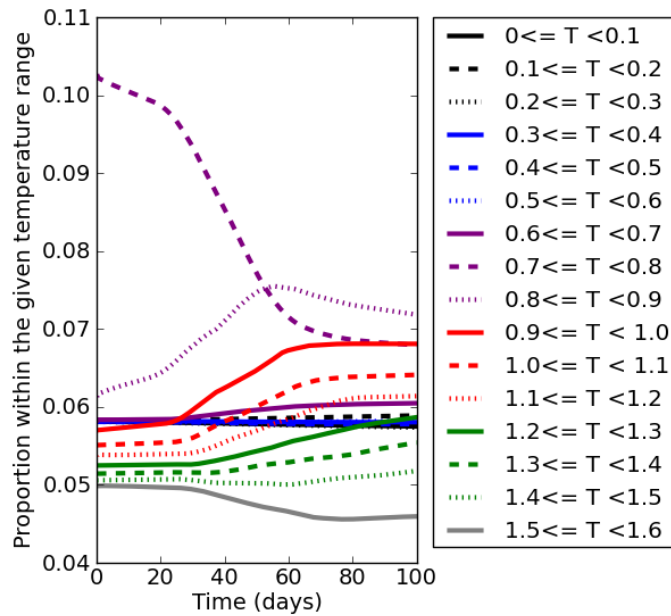


Figure 10.27: Results of the mixing stats diagnostic, showing how the temperature is mixed during the simulation. There is initially most water with a temperature of 0.7-0.8. This mixes during the course of the simulation.

10.11.2 Configuration

The domain extends from 8°W to 40°E and from 28°N to 48°N with an open boundary adjacent to the Atlantic Ocean in the west. The fixed mesh was generated using *gms* with shoreline data taken from the intermediate resolution *gshhs* dataset (see <http://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html>). The single-element deep tetrahedral elements were then extruded in the vertical to fit a 3 arc-minute resolution bathymetric profile subsampled from the 1 arc-minute GEBCO dataset (see <http://www.gebco.net/>). A minimum depth of 3 m is set to prevent wetting-and-drying related numerical instabilities.

The model is driven by both astronomical and co-oscillating boundary tide forcing (see sections 2.4.5.3 and 2.4.5.2) for the four main tidal constituents; M_2 , S_2 , K_1 and O_1 (see Schwiderski, 1980, Wells, 2008). Boundary tide data is sourced from the highly accurate FES2004 model [Lyard et al., 2006] which is read in from a NetCDF file (see section 8.12.4). Frictional drag is applied as a surface-integral boundary condition to the bottom and sides and is based on a quadratic friction law of the form $-C_D|u|u$, where C_D is the drag coefficient, u is the velocity vector (ms^{-1}) and $|u|$ is the magnitude of the velocity vector ($|u| = \sqrt{u_c^2 + v_c^2 + w_c^2}$ where u_c is the x component of velocity (ms^{-1}), v_c is the y component of velocity (ms^{-1}) and w_c is the z component of velocity (ms^{-1}) respectively). The C_D is set 0.0025, a value considered suitable for the majority of numerical ocean tidal models [Wells, 2008].

The model outputs the harmonic constants for the amplitude and phase of each constituent as calculated from a time series using the least-squares method. The timestep is 200 entries in length with data recorded every 5 timesteps after an initial spin up period of 20 hours (simulated time). The timestep itself is 5 minutes. The total runtime is 200 hours (simulated time).

10.11.3 Results

The harmonic amplitudes are presented as plotted scalar fields and compared with a high-resolution 2D model of Tsimplis et al. [1995] in Figure 10.28. The model results are similar to those of Tsimplis

et al. [1995] and generally predict the correct patterns. The semidiurnal constituents give very similar results due to their similar frequencies (Figure 10.28A - D). The amphidromic systems are correctly located in the Sicilian Channel between Sicily and Libya and in the northern Adriatic. The degenerate amphidromes are also accurately positioned near the Balearic Islands and in between Crete and Libya. The model also captures the amplification of the tidal amplitudes in both the Gulf of gables and the northern Adriatic, a phenomena primarily due to resonance of the wave in these regions.

The amplitudes for the diurnal constituents show a similarly good match with the results of Tsimplis et al. [1995] (Figure 10.28E - H). The lowest amplitudes occur in the eastern part of the basin and there is pronounced amplification in the Adriatic Sea caused by this region acting as a quarter-wave oscillator with the diurnal frequency [Wells, 2008]. The degenerate amphidrome along the Libyan coast (around the Gulf of Sirte) is correctly predicted.

The phases as predicted by ICOM and by Tsimplis et al. [1995] are presented in Figure 10.29. These show a general agreement with the amphidromic systems shown to be rotating in an anti-clockwise direction; a feature brought about by Coriolis force deflecting the tidal wave to the right in the Northern Hemisphere. ICOM appears to slightly overpredict the wave speed in all cases; a result that could be attributed to any number of eatures including the bathymetric/mesh resolution and/or an insufficiently low drag coefficient. Another possible source of error is that the phase of the boundary tide and the natural mode of oscillation of the basin might not be synchronised; something that Tsimplis et al. [1995] adjusted repeatedly until they achieved their best results.

The harmonic amplitudes are compared with data from 62 tide gauges (Figure 10.30; Tsimplis et al., 1995, Wells, 2008). The RMS differences for ICOM are typically 2-3 times those of Tsimplis et al. [1995] with the largest errors being for the diurnal constituents (Table 10.4). Despite these discrepancies, the magnitudes of the RMS differences indicate a good match with the tide gauges, even in a strongly microtidal environment such as the Mediterranean Sea.

Tidal Constituent	RMS Differences	
	Tsimplis et al. [1995]	ICOM
M ₂	1.37	3.2
S ₂	0.68	1.9
K ₁	0.82	1.7
O ₁	0.34	1.2

Table 10.4: RMS differences between modelled harmonic amplitudes and real-world data from 62 tide gauges. Data is presented from Tsimplis et al. [1995] and ICOM.

The quality of the match is further highlighted in scatter diagrams plotting the harmonic amplitudes from ICOM at each gauge location against the tide gauge data (Figure 10.31). These reveal how although ICOM tends to marginally overpredict the amplitude there is a strong positive correlation closely delineating $y = x$.

10.12 Hokkaido-Nansei-Okai tsunami

10.12.1 Overview

This example demonstrates the capabilities of simulating wetting and drying processes in Fluidity. The event of interest is the Okushiri tsunami in 1993 caused by the Hokkaido Nansei-Okai earthquake offshore of southwestern Hokkaido Island, Japan. This earthquake reached a magnitude of 7.8 (Mw) and the resulting tsunami hit a sparsely populated part of the Okushiri island, Japan with a runup height of up to 30m.

To investigate the danger of such events, the Research Institute for Electric Power Industry (CRIEPI) in Abiko, Japan constructed a 1/400 laboratory model of the area around the island Liu [2008]. The

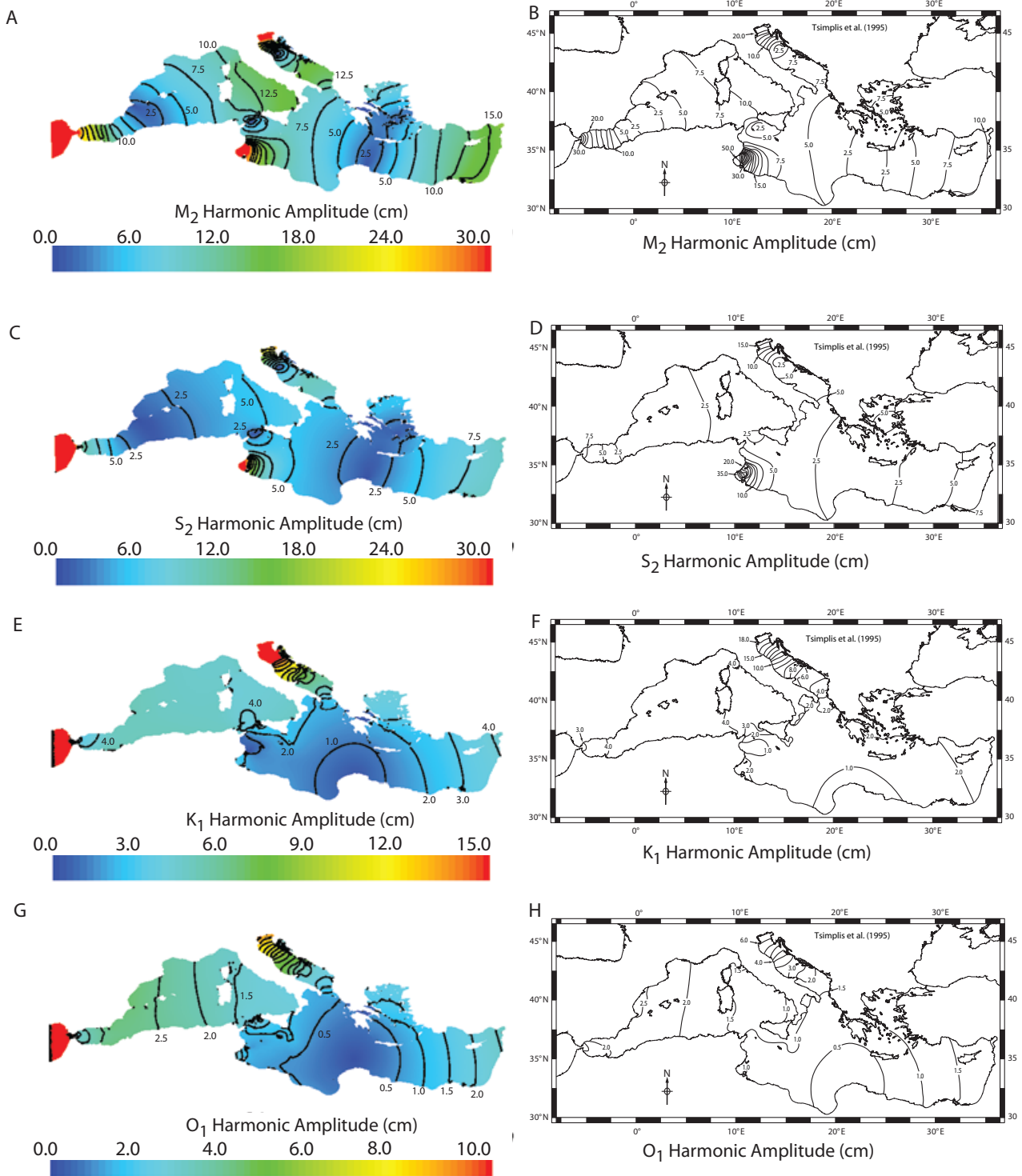


Figure 10.28: Plots of the tidal harmonic amplitudes in the Mediterranean Sea from ICOM and the high resolution model of Tsimplis et al. [1995].

following configuration simulates this laboratory setup and uses the experimental measurements to benchmark Fluidity.

Fluidity features used in this example:

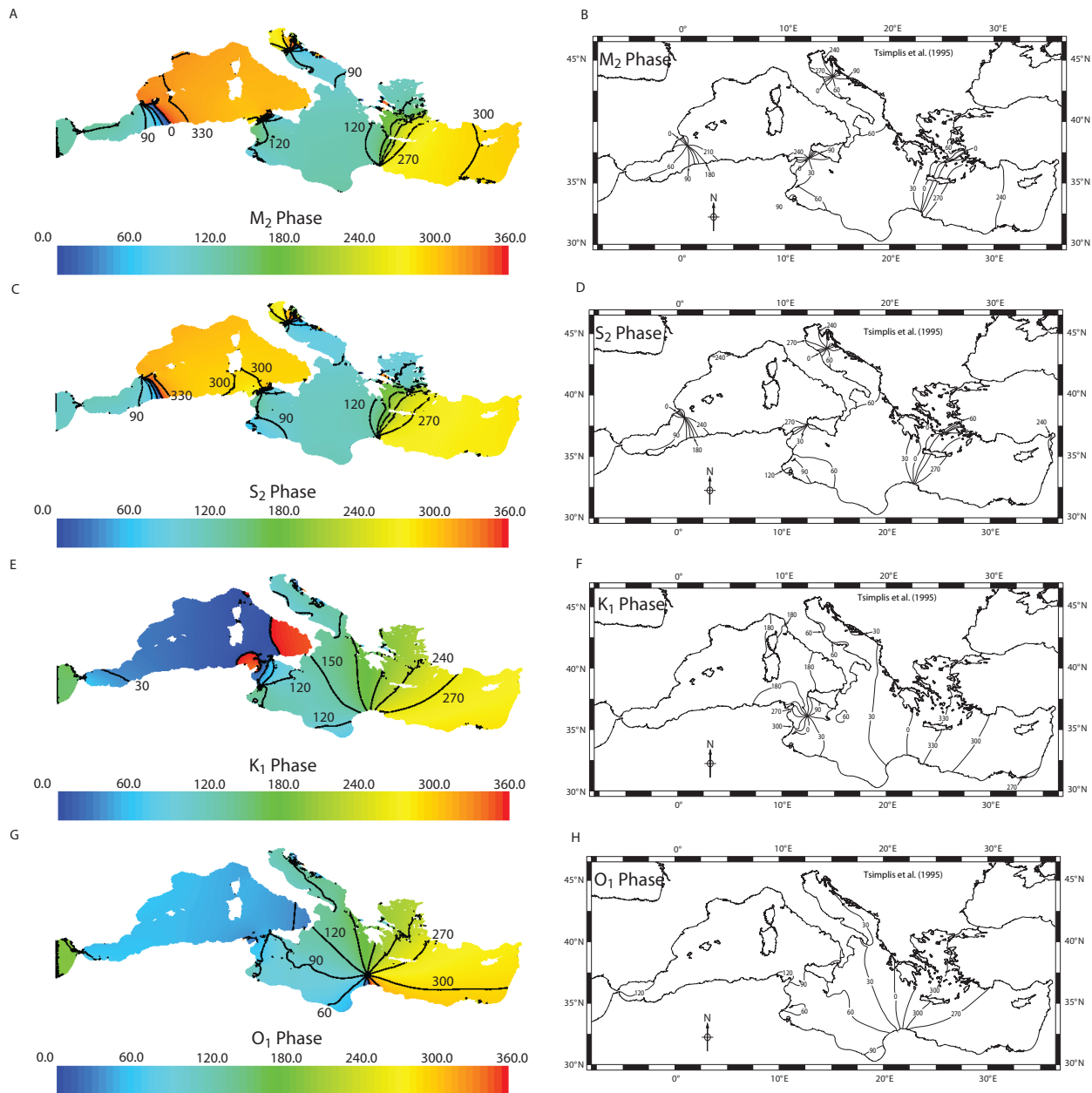


Figure 10.29: Plots of the tidal harmonic phases in the Mediterranean Sea from ICOM and the high resolution model of Tsimplis et al. [1995].

- Free surface with wetting and drying, see 8.12.3.6.
- Detectors, see 8.3.4.12.

10.12.2 Configuration

The simulation configuration resembles the experimental setup as closely as possible. The considered domain is a basin with walls on each side except the left where the water level is enforced. The basin measures $5.448\text{m} \times 3.402\text{m}$ and the bathymetry and coastal topography correspond to measurement data, see Figure 10.32. Three surface elevation gauge stations were deployed in the experiment. Detectors extract the surface elevation information at every timestep.

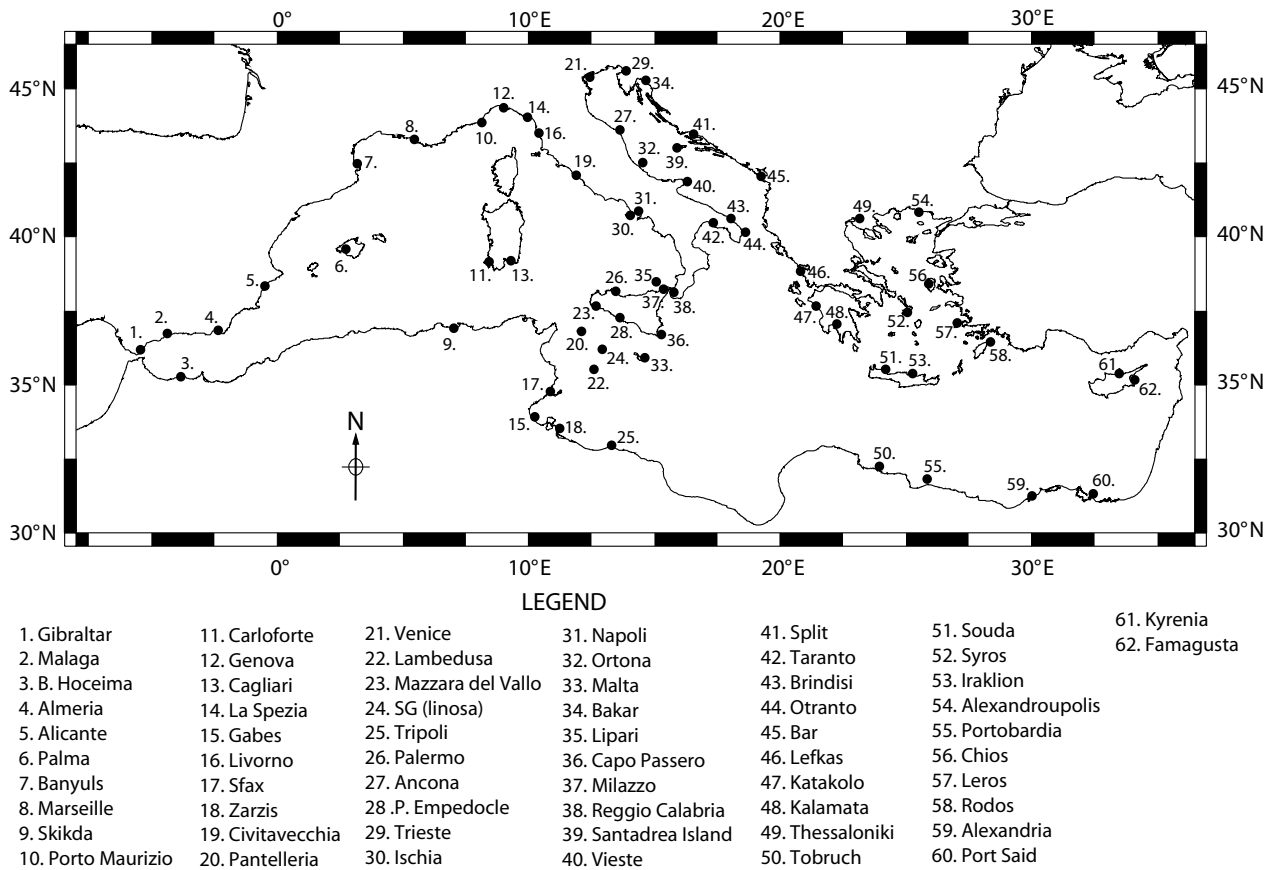


Figure 10.30: Locations of 62 tide gauges in the Mediterranean Sea. Modified from Wells [2008] with data originally taken from Tsimplis et al. [1995].

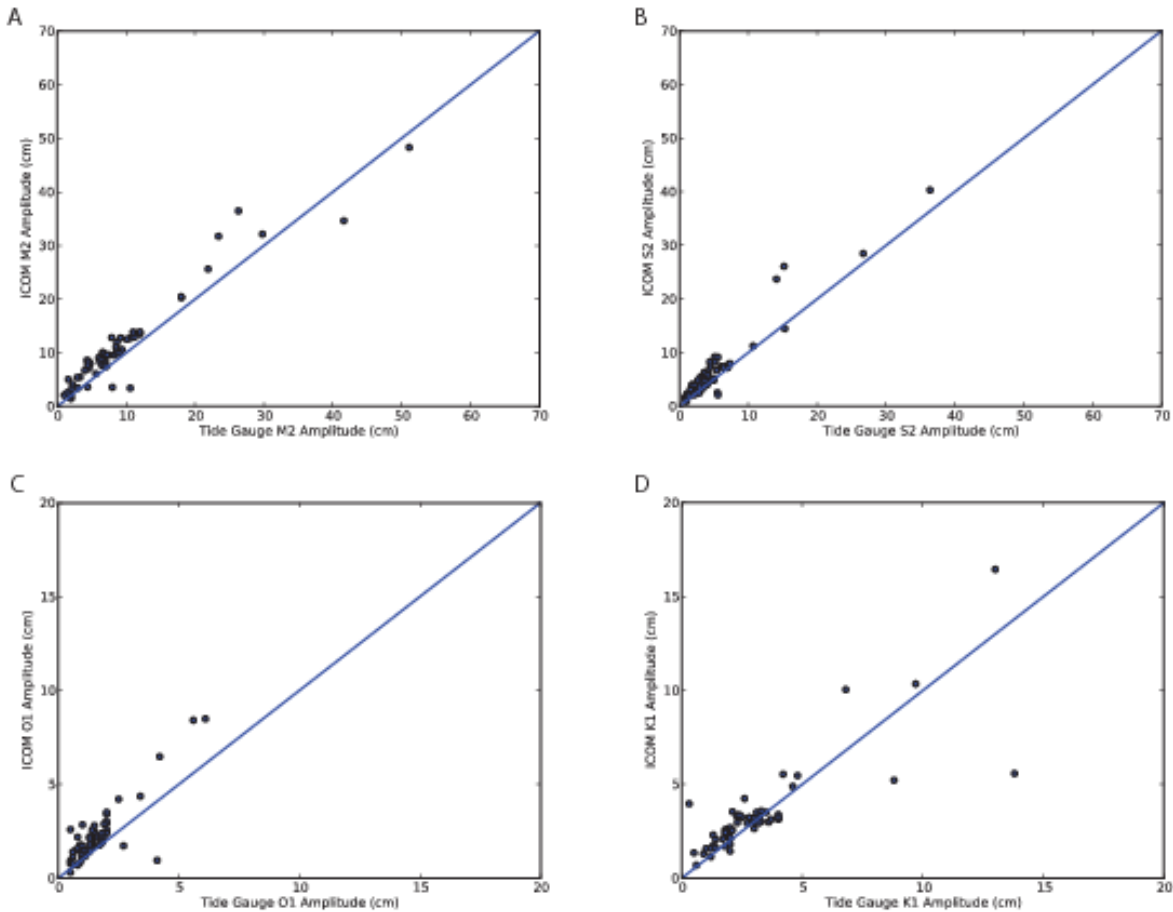


Figure 10.31: Scatter diagrams plotting harmonic amplitudes from ICOM at each gauge location against tide gauge data.

The mesh used for the simulation is a single layer horizontally unstructured mesh consisting of 19,506 tetrahedral elements with increased resolution near the inundation areas. The equations are solved with the $P_1 - P_1$ finite element pair, a backward Euler time discretisation with a time-step of 0.1s. The wetting and drying algorithm in Fluidity requires the user to set a minimal water level thickness, which is here set to $d_0 = 0.5\text{mm}$. If the water surface reaches that level at a point, this point is defined to be dry. The isotropic kinematic viscosity and gravity magnitude are set to $0.01\text{m}^2\text{s}^{-1}$ and 9.81ms^{-2} , respectively. On the left boundary the tsunami wave shown in 10.32 is prescribed and no-normal flow boundary conditions are applied at the other sides of the domain and the bottom to resemble the solid boundaries in the experiment. In addition, a Manning-Strickler drag is used at the bottom with $n = 0.002\text{sm}^{-\frac{1}{3}}$. To prevent wave breaking in the simulation, this coefficient is increased to $0.2\text{sm}^{-\frac{1}{3}}$ in a rectangular area with a side length of 0.5m centred at (3.4m, 1.7m) (which is the centre of the island in the domain) and a fourth order stabilization is applied to prevent wave breaking.

10.12.3 Results

The result of this example is shown in Figure 10.33. The plot shows the surface elevation measurements at the three gauge stations of the laboratory experiment compared to the values from the simulation.

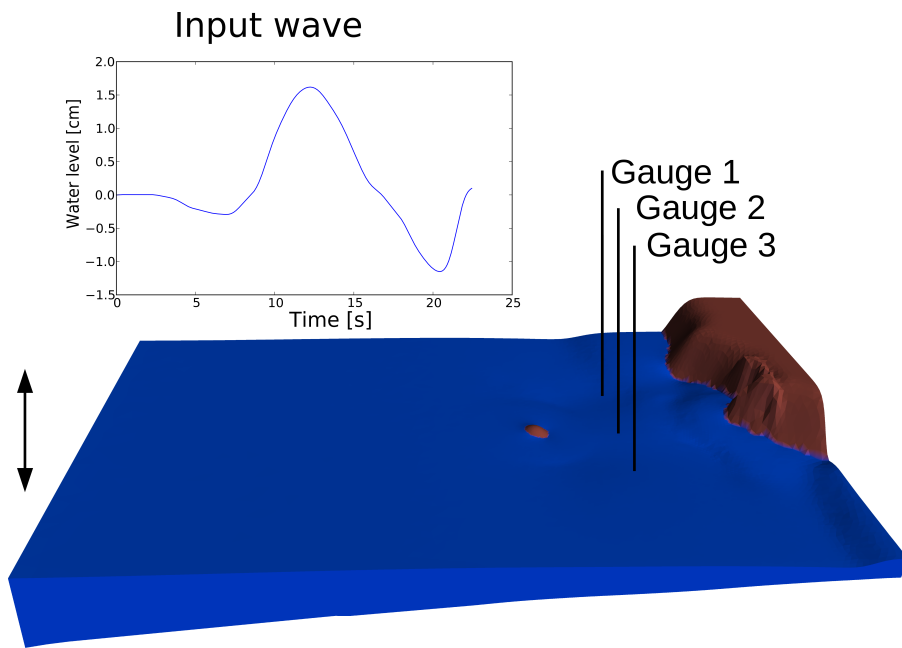


Figure 10.32: The bathymetry and the three gauge stations used for the Hokkaido-Nansei-Oki tsunami example.

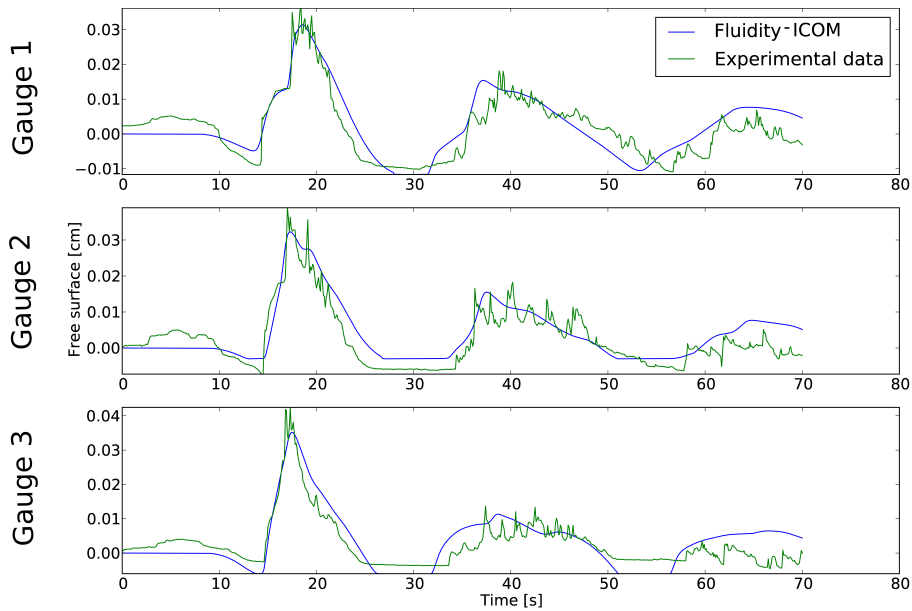


Figure 10.33: The input wave elevation of the Okushiri tsunami test case (a) and the numerical and experimental results at “Gauge 1” (top), “Gauge 2” (middle) and “Gauge 3” (bottom) (b).

10.12.4 Exercises

1. Add more detectors to the simulation.
2. Increase the wetting and drying threshold parameter. Which effects does it have to the result?
3. Change the viscosity parameter. Does it make a difference to the inundation of the tsunami event?

10.13 Tephra settling

10.13.1 Overview

In this example, Fluidity is used to replicate a laboratory experiment of tephra (fine volcanic ash particles) settling through a tank of water [Carey, 1997]. The Carey [1997] experiments introduced tephra particles into a $0.3 \times 0.3 \times 0.7$ m tank, filled with water, from above using a delivery system and a particle disperser. The particles settled through the air in the tank at an approximately constant rate until they landed in the water and began to settle through the water at a much reduced velocity. While the particles in the water were sufficiently dispersed their settling velocity was that predicted by Stokes' flow (a few mm/s). However, the build-up of particles caused by the air-water interface eventually created a layer of particles and water with a bulk density so great, relative to the density of the particle-poor water beneath, as to be gravitationally unstable and promote the formation of a vertical gravity current (plumes). The settling velocity of these plumes of particles and water was observed to be an order of magnitude greater than the Stokes settling velocity of the individual particles.

In the Fluidity simulations, rather than represent each tephra particle individually using a multi-material approach (which would be prohibitively expensive), a "dispersed multiphase" approach is adopted (see 2.4.7), in which one phase represents the water in the tank and the other phase represents all the particles dispersed within the water. Similar to the water column collapse example (10.9), a volume fraction field is used to distinguish between the continuous phase (water) and the dispersed phase (particles). In an element the volume fraction of particles represents the fraction of the element volume that is occupied by particles, which is typically very small. In contrast to the multimaterial approach, however, the multiphase approach assigns each phase a separate velocity, allowing the dispersed phase (particles) to move through the continuous phase (water). In this example, the dispersed phase is more dense than the continuous phase and so sinks under the influence of gravity.

The simulation replicates one of the Carey [1997] experiments (with a characteristic particle size of $48 \mu\text{m}$) by considering a constant influx of the particle phase into a 2D tank of water. The simulation results can be compared to the experiment in terms of the conditions under which plumes of tephra particles form as well as the settling velocities of the individual particles and plumes. The example demonstrates the following functionality:

- 2D flow
- Multi-phase flow
- Incompressible flow
- Adaptive timestepping

The simulation illustrated here took ~ 1 hour to run in serial on an Intel Xeon E5430 2.66 GHz processor.

10.13.2 Problem specification

The simulation uses a 0.3×0.7 metre domain, replicating the cross-section of the water tank used in the experiments, with an initial characteristic element size of 0.01 metres. No normal flow boundary conditions are weakly imposed along with a zero velocity initial condition. The parameters for density, viscosity, particle diameter and gravity are $\rho_p = 2340 \text{ kgm}^{-3}$, $\rho_l = 1000 \text{ kgm}^{-3}$, $\mu_l = 0.001 \text{ Pa} \cdot \text{s}$, $d = 48 \times 10^{-6} \text{ m}$, and $\mathbf{g} = 9.8 \text{ ms}^{-2}$ respectively (with the subscripts p and l denoting the properties of the particle/tephra and liquid phase).

The influx of particles from the air above is simulated using a `flux` boundary condition at the top of the domain. This allows tephra to flux in at a rate of $0.472 \text{ gm}^{-2}\text{s}^{-1}$. The results of the experiments showed that tephra particles initially settled individually at a Stokes velocity of 0.17 cms^{-1} . When the concentration was high enough, plumes formed and descended to the bottom of the tank with velocities more than ten times greater than those of individual particles.

10.13.3 Results

Several timesteps of the example simulation can be seen in Figure 10.34. These images can be generated by visualising the `vtu` files using Paraview. Better resolved plumes can be produced by using a smaller characteristic element size (e.g. $\Delta x = 0.0025 \text{ m}$, shown in Figure 10.35).

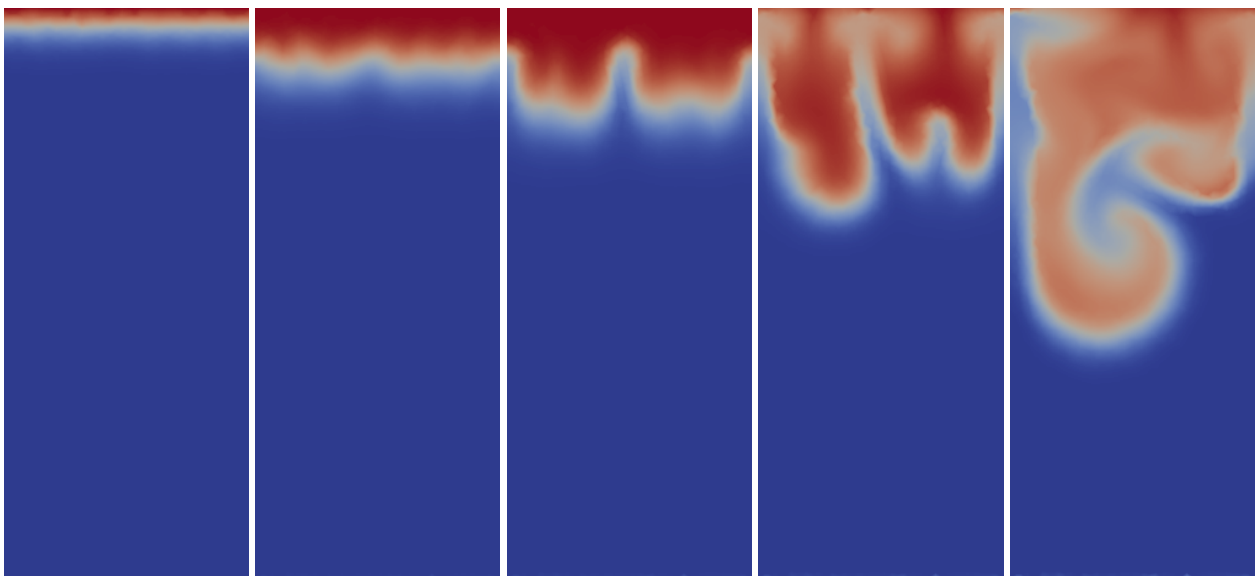


Figure 10.34: Tephra (particle) phase volume fraction at time $t = 10, 30, 50, 80, 110$ seconds. These images are from a lower-resolution version of the tephra settling example problem, with a characteristic element size of $\Delta x = 0.01 \text{ m}$.

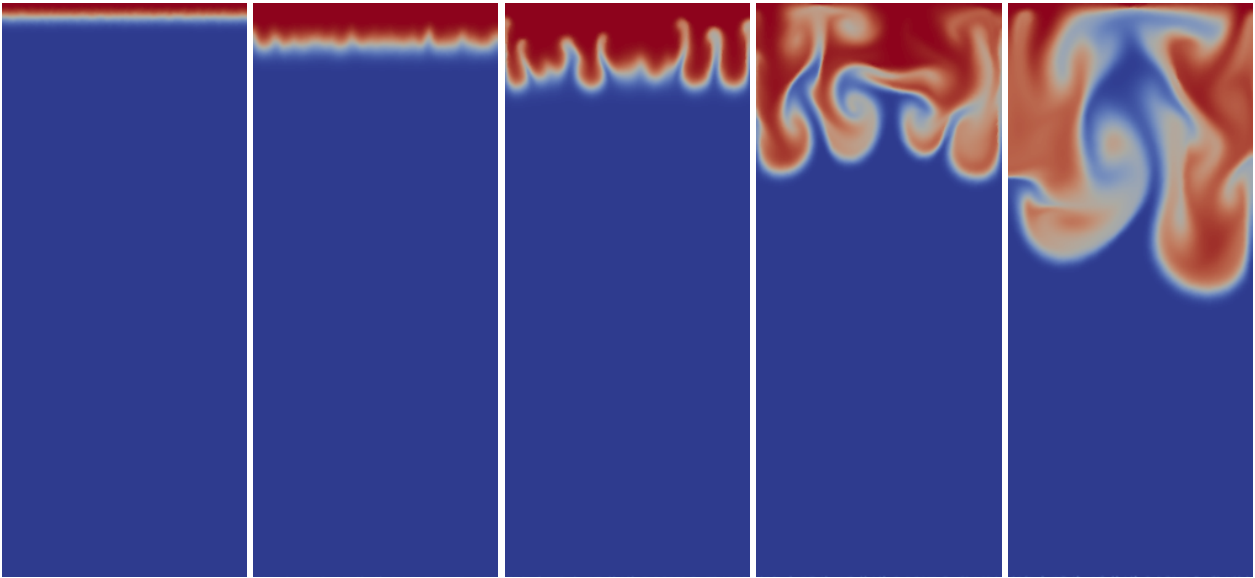


Figure 10.35: Tephra (particle) phase volume fraction at time $t = 10, 30, 50, 80, 110$ seconds. The onset of plumes is between 10 and 30 seconds. Note that these images are from a high-resolution version of the tephra settling example problem, with a characteristic element size of $\Delta x = 0.0025$ m.

Figure 10.36 (reproducible by typing *make postprocess* at the command line) illustrates how the tephra particles initially settle at approximately 0.17 cm s^{-1} , as predicted by Stokes' law. As more tephra fluxes in, the layer becomes unstable and plumes begin to form, resulting in settling velocities over 10 times greater than that of an individual particle.

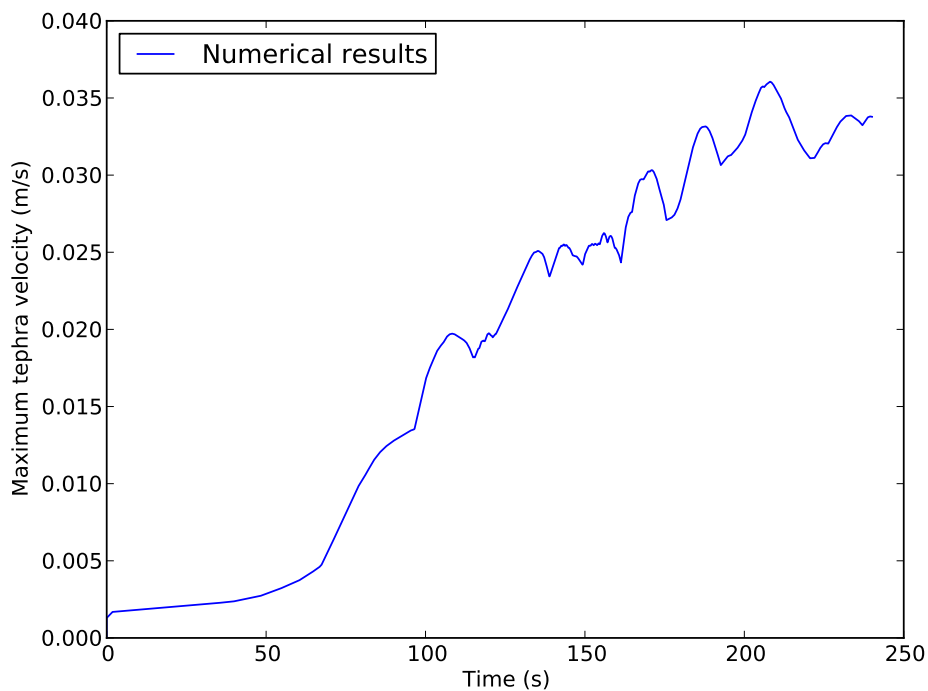


Figure 10.36: Maximum velocity of the tephra phase against time.

10.13.4 Exercises

To explore the multiphase functionality of Fluidity, the following variations on this example would be constructive learning exercises:

- Decrease the characteristic element size to better resolve the plume behaviour
- Alter the particle size to observe its affect on plume formation
- Add a second dispersed phase (with a different particle size).

Bibliography

- M. T. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Wiley, New York, 2000. ISBN 978-0-471-29411-5. URL <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047129411X.html>. 95
- F. Alauzet, P. L. George, B. Mohammadi, P. J. Frey, and H. Borouchaki. Transient fixed point-based unstructured mesh adaptation. *International Journal for Numerical Methods in Fluids*, 43(6-7):729–745, 2003. doi: [10.1002/flid.548](https://doi.org/10.1002/flid.548). 102
- F. Alauzet, A. Loseille, A. Dervieux, and P. Frey. Multi-dimensional continuous metric for mesh adaptation. In P. P. Pébay, editor, *Proceedings of the 15th International Meshing Roundtable*, pages 191–214, Birmingham, Alabama, 2006. Springer. doi: [10.1007/978-3-540-34958-7_12](https://doi.org/10.1007/978-3-540-34958-7_12). 98
- F. Alauzet, S. Borel-Sandou, L. Dumas, A. Dervieux, Q. Dinh, S. Kleinveld, A. Loseille, Y. Mesri, and G. Rogé. Multi-model and multi-scale optimization strategies. Application to sonic boom reduction. *European Journal of Computational Mechanics*, 17(1-2):191–214, 2008. doi: [10.3166/remn.17.245-269](https://doi.org/10.3166/remn.17.245-269). 98, 101
- D.N. Arnold, F. Brezzi, B. Cockburn, and L.D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5):1749–1779, 2002. 41, 42, 43
- I. Babuška and W. C. Rheinboldt. Error estimates for adaptive finite element computations. *SIAM Journal on Numerical Analysis*, 15(4):736–754, 1978a. doi: [10.1137/0715049](https://doi.org/10.1137/0715049). 95
- I. Babuška and W. C. Rheinboldt. A-posteriori error estimates for the finite element method. *International Journal for Numerical Methods in Engineering*, 12(10):1597–1615, 1978b. doi: [10.1002/nme.1620121010](https://doi.org/10.1002/nme.1620121010). 95
- I. Babuška and M. Suri. The p and h - p versions of the Finite Element method, basic principles and properties. *SIAM Review*, 36(4):578, 1994. doi: [10.1137/1036141](https://doi.org/10.1137/1036141). 95
- W. Bangerth and R. Rannacher. Adaptive finite element techniques for the acoustic wave equation. *Journal of Computational Acoustics*, 9(2):575–591, 2001. doi: [10.1142/S0218396X01000668](https://doi.org/10.1142/S0218396X01000668). 102
- W. Bangerth and R. Rannacher. *Adaptive finite element methods for differential equations*. ETH Zürich Lectures in Mathematics. Birkhäuser, 2003. ISBN 3764370092. 95
- J Bardina, J Ferziger, and W Reynolds. Improved subgrid scale models for Large Eddy Simulations. *AIAA paper 80-1357*, 1980. 76
- F. Bassi and S. Rebay. A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier-stokes equations. *J. Comput. Phys.*, 131:267–279, 1997. 43, 53
- G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967. 13, 16, 17, 18, 20, 22
- Z. P. Bažant. Spurious reflection of elastic waves in nonuniform finite element grids. *Computer Methods in Applied Mechanics and Engineering*, 16(1):91–100, 1978. doi: [10.1016/0045-7825\(78\)90035-X](https://doi.org/10.1016/0045-7825(78)90035-X). 102

- Y. Bazilevs, C. Michler, V.M. Calo, and T.J.R. Hughes. Weak Dirichlet boundary conditions for wall-bounded turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 196(49-52):4853 – 4862, 2007. ISSN 0045-7825. doi: [10.1016/j.cma.2007.06.026](https://doi.org/10.1016/j.cma.2007.06.026). URL <http://www.sciencedirect.com/science/article/B6V29-4PCPFD2-1/2/aeb31fb110ba0801dae355107802d59a>. 137
- R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001. doi: [10.1017/S0962492901000010](https://doi.org/10.1017/S0962492901000010). 95
- T. Brooke Benjamin. Gravity currents and related phenomena. *Journal of Fluid Mechanics*, 31(2):209–248, 1968. 204
- J. H. T. Bentham. *Microscale Modelling of Air Flow and Pollutant Dispersion in the Urban Environment*. PhD thesis, Imperial College London, 2003. 75, 77, 133
- M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989. doi: [10.1016/0021-9991\(89\)90035-1](https://doi.org/10.1016/0021-9991(89)90035-1). 95
- Alan F. Blumberg, Boris Galperin, and Donald J. O'Connor. Modeling vertical structure of open-channel flows. *Journal of Hydraulic Engineering*, 118(8):1119–1134, 1992. doi: [10.1061/\(ASCE\)0733-9429\(1992\)118:8\(1119\)](https://doi.org/10.1061/(ASCE)0733-9429(1992)118:8(1119)). 71
- H. Borouchaki, F. Hecht, and P. J. Frey. Mesh gradation control. *International Journal for Numerical Methods in Engineering*, 43(6):1143–1165, 1998. doi: [10.1002/\(SICI\)1097-0207\(19981130\)43:6;1143::AID-NME470;3.0.CO;2-I](https://doi.org/10.1002/(SICI)1097-0207(19981130)43:6<1143::AID-NME470>3.0.CO;2-I). 102
- O. Botella and R. Peyret. Benchmark spectral results on the lid-driven cavity flow. *Computers & Fluids*, 27(4):421–433, 1998. xvii, 208, 209
- T. Boyer, S. Levitus, H. Garcia, R.A. Locarnini, C. Stephens, and J. Antonov. Objective analyses of annual, seasonal, and monthly temperature and salinity for the world ocean on a 0.25° grid. *International Journal of Climatology*, 25(7):931–945, 2005. doi: [10.1002/joc.1173](https://doi.org/10.1002/joc.1173). 173
- A. N. Brooks and T. J. R. Hughes. Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations. *Comput. Methods Appl. Mech. Eng.*, 32:199–259, 1982. 32, 33, 34
- P. P. Brown and D. F. Lawler. Sphere drag and settling velocity revisited. *Journal of Environmental Engineering*, 129(3):222–231, 2003. 216
- C. H. Bruneau and M. Saad. The 2D lid-driven cavity problem revisited. *Computers and Fluids*, 35(3):326–348, 2006. 208
- C. J. Budd, W. Huang, and R. D. Russell. Adaptivity with moving grids. *Acta Numerica*, 18:111–241, 2009. doi: [10.1017/S0962492906400015](https://doi.org/10.1017/S0962492906400015). 95
- H. Burchard, K. Bolding, and M. R. Villarreal. GOTM – a general ocean turbulence model. theory, applications and test cases. Technical Report EUR 18745, European Commission Rep, 1999. 71, 72
- Hans Burchard. On the q^2l equation by mellor and yamada (1982). *Journal of Physical Oceanography*, 31(5):1377–1387, 2001. 70
- Hans Burchard, Ole Petersen, and Tom P. Rippeth. Comparing the performance of the mellor-yamada and the k- ϵ two-equation turbulence models. *J. Geophys. Res.*, 103(C5):10543–10554, 1998. ISSN 0148-0227. doi: [10.1029/98JC00261](https://doi.org/10.1029/98JC00261). 70
- A.S. Candy. *Subgrid Scale Modelling of Transport Processes*. PhD thesis, Imperial College London, 2008. 76

- M. I. Cantero, J. R. Lee, S. Balachandar, and M. H. Garcia. On the front velocity of gravity currents. *jfm*, 586:1–39, 2007. doi: [10.1017/S0022112007005769](https://doi.org/10.1017/S0022112007005769). 204
- V Canuto and Y Cheng. Determination of the smagorinski-lilly constant cs . *Physics of Fluids*, 9(5): 1368–1378, 1997. 75
- V. M. Canuto, A. Howard, Y. Cheng, and M. S. Dubovikov. Ocean turbulence. part i: One-point closure model – momentum and heat vertical diffusivities. *Journal of Physical Oceanography*, 31(6): 1413–1426, 2001. 71
- S. Carey. Influence of convective sedimentation on the formation of widespread tephra fall layers in the deep sea. *Geology*, 25(9):839–842, 1997. 236
- M. J. Castro-Díaz, F. Hecht, B. Mohammadi, and O. Pironneau. Anisotropic unstructured mesh adaptation for flow simulations. *International Journal for Numerical Methods in Fluids*, 25(4):475–491, 1997. doi: [10.1002/\(SICI\)1097-0363\(19970830\)25:4<475::AID-FLD575>3.0.CO;2-6](https://doi.org/10.1002/(SICI)1097-0363(19970830)25:4<475::AID-FLD575>3.0.CO;2-6). 101
- Long Chen, Pengtao Sun, and Jinchao Xu. Optimal anisotropic meshes for minimizing interpolation errors in L^p -norm. *Mathematics of Computation*, 76:179–204, 2007. doi: [10.1090/S0025-5718-06-01896-5](https://doi.org/10.1090/S0025-5718-06-01896-5). 101
- P. G. Ciarlet and J. L. Lions. *Handbook of Numerical Analysis Volume 7*. Elsevier, 2000. 53
- B. Cockburn and C.-W. Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM J. Numer. Anal.*, 35:2440–2463, 1998. 42
- B. Cockburn and C-W. Shu. Runge-Kutta Discontinuous Galerkin Methods for Convection-Dominated Problems. *Journal of Scientific Computing*, 16(3):173–261, 2001. 38, 39
- A. Colagrossi and M. Landrini. Numerical simulation of interfacial flows by smoothed particle hydrodynamics. *Journal of Computational Physics*, 191(2):448–475, 2003. ISSN 0021-9991. doi: [10.1016/S0021-9991\(03\)00324-3](https://doi.org/10.1016/S0021-9991(03)00324-3). URL <http://www.sciencedirect.com/science/article/B6WHY-49CMJ22-1/2/9d25ed6668fde488a2f4429e97411815>. 221, 225
- G. Compère, J. F. Remacle, J. Jansson, and J. Hoffman. A mesh adaptation framework for dealing with large deforming meshes. *International Journal for Numerical Methods in Engineering*, 82(7):843–867, 2010. doi: [10.1002/nme.2788](https://doi.org/10.1002/nme.2788). 99
- G. Coppola, S. J. Sherwin, and J. Peiro. Nonlinear particle tracking for high-order elements. *Journal of Computational Physics*, 172:356–386, 2001. doi: [doi:10.1006/jcph.2001.6829](https://doi.org/10.1006/jcph.2001.6829). 67
- C. J. Cotter, D. A. Ham, and C. C. Pain. A mixed discontinuous/continuous finite element pair for shallow-water ocean modelling. *Ocean Modelling*, 26:86–90, 2009. 60, 61
- F. Courty, D. Leservoisier, P.-L. George, and A. Dervieux. Continuous metrics and mesh adaptation. *Applied Numerical Mathematics*, 56(2):117–145, 2006. doi: [10.1016/j.apnum.2005.03.001](https://doi.org/10.1016/j.apnum.2005.03.001). 98
- C.T. Crowe, M. Sommerfeld, and Y. Tsuji. *Multiphase Flows with Droplets and Particles*. CRC Press, 1998. 26
- B. Cushman-Roisin. *Introduction to Geophysical Fluid Dynamics*. Prentice Hall, New Jersey, USA, 1994. 13, 20, 21
- James Deardorff. A numerical study of three-dimensional turbulent channel flow at large Reynolds numbers. *J. Fluid Mech.*, 41:453–480, 1970. 75
- James Deardorff. On the magnitude of the subgrid scale eddy coefficient. *J. Comp. Phys.*, 7:120–133, 1971. 75

- B. Després and F. Lagoutière. Contact discontinuity capturing schemes for linear advection and compressible gas dynamics. *Journal of Scientific Computing*, 16(4):479–524, December 2001. doi: [10.1023/A:1013298408777](https://doi.org/10.1023/A:1013298408777). URL <http://dx.doi.org/10.1023/A:1013298408777>. 49
- Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002. 105
- J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. John Wiley & Sons, 2003. 32, 33, 34
- H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press, USA, 2005. 29, 30
- E. Erturk, T. C. Corke, and C. Gokcol. Numerical solutions of 2-d steady incompressible driven cavity flow at high Reynolds numbers. *International Journal for Numerical Methods in Fluids*, 48(7):747–774, 2005. 208
- C. W. Fairall, E. F. Bradley, J. E. Hare, A. A. Grachev, and J. B. Edson. Bulk parameterization of air-sea fluxes: Updates and verification for the coare algorithm. *Journal of Climate*, 16(4):571–591, 2003. doi: [10.1175/1520-0442\(2003\)016;0571:BPOASF;2.0.CO;2](https://doi.org/10.1175/1520-0442(2003)016;0571:BPOASF;2.0.CO;2). 24, 138
- T. K. Fannelop. *Fluid Mechanics for Industrial Safety and Environmental Protection*. Elsevier Science Ltd, 1994. 203
- P. E. Farrell. *Galerkin projection of discrete fields via supermesh construction*. PhD thesis, Imperial College London, 2009. 98
- P. E. Farrell and J. R. Maddison. Conservative interpolation between volume meshes by local Galerkin projection. *Computer Methods in Applied Mechanics and Engineering*, 2010. doi: [10.1016/j.cma.2010.07.015](https://doi.org/10.1016/j.cma.2010.07.015). 147, 184
- P. E. Farrell and J. R. Maddison. Conservative interpolation between volume meshes by local Galerkin projection. *Computer Methods in Applied Mechanics and Engineering*, 200(1-4):89–100, 2011. doi: [10.1016/j.cma.2010.07.015](https://doi.org/10.1016/j.cma.2010.07.015). 104
- P. E. Farrell, M. D. Piggott, C. C. Pain, G. J. Gorman, and C. R. Wilson. Conservative interpolation between unstructured meshes via supermesh construction. *Computer Methods in Applied Mechanics and Engineering*, 198(33-36):2632–2642, 2009. doi: [10.1016/j.cma.2009.03.004](https://doi.org/10.1016/j.cma.2009.03.004). 104, 105, 147, 184
- P. J. Frey. Generation and adaptation of computational surface meshes from discrete anatomical data. *International Journal for Numerical Methods in Engineering*, 60(6):1049–1074, 2004. 102
- O. B. Fringer, M. Gerritsen, and R. L. Street. An unstructured-grid, finite-volume, nonhydrostatic, parallel coastal ocean simulator. *Ocean Modelling*, 14(3-4):139–173, 2006. doi: [10.1016/j.ocemod.2006.03.006](https://doi.org/10.1016/j.ocemod.2006.03.006). 204
- S.W. Funke, C.C. Pain, S.C. Kramer, and M.D. Piggott. A wetting and drying algorithm with a combined pressure/free-surface formulation for non-hydrostatic models. *Advances in Water Resources*, 2011. ISSN 0309-1708. doi: [10.1016/j.advwatres.2011.08.007](https://doi.org/10.1016/j.advwatres.2011.08.007). 20
- P. L. George and H. Borouchaki. *Delaunay Triangulation and Meshing: Application to Finite Elements*. Hermes, 1998. 98, 104
- Massimo Germano. Turbulence: the filtering approach. *J. Fluid Mech.*, 238:325–336, 1992. 75, 77
- Massimo Germano, Ugo Piomelli, Parviz Moin, and William H. Cabot. A dynamic subgrid-scale eddy viscosity model. *Physics of Fluids*, 3(7):1760–1765, 1991. ISSN 08998213. doi: [10.1063/1.857955](https://doi.org/10.1063/1.857955). 75, 76, 77

- C. Geuzaine, B. Meys, F. Henrotte, P. Dular, and W. Legros. A Galerkin projection method for mixed finite elements. *IEEE Transactions on Magnetics*, 35(3):1438–1441, 1999. doi: [10.1109/20.767236](https://doi.org/10.1109/20.767236). 104
- M. M. Gibson and B. E. Launder. Ground effects on pressure fluctuations in the atmospheric boundary layer. *Journal of Fluid Mechanics Digital Archive*, 86(03):491–511, 1978. doi: [10.1017/S0022112078001251](https://doi.org/10.1017/S0022112078001251). 71
- A.E. Gill. *Atmosphere-ocean dynamics*. Academic press, 1982. 20
- Jeffrey Grandy. Conservative remapping and region overlays by intersecting arbitrary polyhedra. *Journal of Computational Physics*, 148(2):433 – 466, 1999. ISSN 0021-9991. doi: DOI: [10.1006/jcph.1998.6125](https://doi.org/10.1006/jcph.1998.6125). 147
- Donald D. Gray. The validity of the boussinesq approximation for liquids and gases. *International Journal of Heat and Mass Transfer*, 19(5):545 – 551, 1976. doi: [doi:10.1016/0017-9310\(76\)90168-X](https://doi.org/10.1016/0017-9310(76)90168-X). 24
- D. M. Greaves. Simulation of viscous water column collapse using adapting hierarchical grids. *International Journal for Numerical Methods in Fluids*, 50(6):693–711, 2006. doi: [10.1002/fld.1073](https://doi.org/10.1002/fld.1073). URL <http://dx.doi.org/10.1002/fld.1073>. 220
- P. M. Gresho and S. T. Chan. Solving the incompressible navier-stokes equations using consistent mass and a pressure poisson equation. *ASME Symposium on recent development in CFD, Chicago 95*, pages 51 – 73, 1988. 29, 58, 60
- P. M. Gresho and R. Sani. On pressure boundary conditions for the incompressible Navier-Stokes equations. *Int. J. Numer. Methods Fluids*, 7:1111–1145, 1987. 127
- D. A. Ham, P. E. Farrell, G. J. Gorman, J. R. Maddison, C. R. Wilson, S. C. Kramer, J. Shipton, G. S. Collins, C. J. Cotter, and M. D. Piggott. Spud 1.0: generalising and automating the user interfaces of scientific computer models. *Geoscientific Model Development*, 2(1):33–42, 2009. doi: [10.5194/gmd-2-33-2009](https://doi.org/10.5194/gmd-2-33-2009). 107
- C. Härtel, E. Meiburg, and F. Necker. Vorticity dynamics during the start-up phase of gravity currents. *Società Italiana di Fisica*, 22(6):823–833, 1999. 204
- C. Härtel, E. Meiburg, and Fr Necker. Analysis and direct numerical simulation of the flow at a gravity-current head. part 1. flow topology and front speed for slip and no-slip boundaries. *Journal of Fluid Mechanics*, 418:189–212, 2000. xvii, 204, 206
- O. Hassan, E. J. Probert, and K. Morgan. Unstructured mesh procedures for the simulation of three-dimensional transient compressible inviscid flows with moving boundary components. *International Journal for Numerical Methods in Fluids*, 27(1-4):41–55, 1998. doi: [10.1002/\(SICI\)1097-0363\(199801\)27:1/4;1::AID-FLD649;3.0.CO;2-5](https://doi.org/10.1002/(SICI)1097-0363(199801)27:1/4;1::AID-FLD649;3.0.CO;2-5). 98
- H. R. Hiester, M. D. Piggott, and P. A. Allison. The impact of mesh adaptivity on the gravity current front speed in a two-dimensional lock-exchange. *Ocean Modelling*, 38(1–2):1–21, 2011. doi: [10.1016/j.ocemod.2011.01.003](https://doi.org/10.1016/j.ocemod.2011.01.003). 106, 205
- B. P. Hignett, A. A. White, R. D. Carter, W. D. N. Jackson, and R. M. Small. A comparison of laboratory measurements and numerical simulations of baroclinic wave flows in a rotating cylindrical annulus. *Quarterly Journal of the Royal Meteorological Society*, 111(463), 1985. 165
- P. Houston and E. Süli. *hp*-Adaptive discontinuous Galerkin finite element methods for first-order hyperbolic problems. *SIAM Journal on Scientific Computing*, 23(4):1226–1252, 2001. doi: [10.1137/S1064827500378799](https://doi.org/10.1137/S1064827500378799). 95
- T. J. R. Hughes. Recent progress in the development and understanding of SUPG methods with special reference to the compressible Euler and Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 7(11):1261–1275, 1987. 32, 33

- H. E. Huppert. Gravity currents: a personal perspective. *Journal of Fluid Mechanics*, 554:299–322, 2006. 203
- H. E. Huppert and J. E. Simpson. The slumping of gravity currents. *Journal of Fluid Mechanics*, 99(4): 785–799, 1980. 204
- F. Ilinca and D. Pelletier. An adaptive finite element method for a two- equation turbulence model in wall-bounded flows. *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS*, 24: 101–120, 1997. xvii, 210, 211, 212
- The femtools manual*. Imperial College London, 2009. 97
- M. Ishii. *Thermo-Fluid Dynamic Theory of Two-Phase Flow*. Eyrolles, 1975. 26
- N. Jarrin, S. Benhamadouche, D. Laurence, and R. Prosser. A synthetic-eddy-method for generating inflow conditions for large-eddy simulations. *International Journal of Heat and Fluid Flow*, 27:585–593, 2006. 138
- A. Jenkins and A. Bombosch. Modeling the effects of frazil ice crystals on the dynamics and thermodynamics of ice shelf water plumes. *J. Geophys. Res.*, 100:6967–6981, 1995. 78
- L. H. Kantha and C. A. Clayson. *Numerical Models of Oceans and Oceanic Processes*. Academic Press, International Geophysics Series, San Diego, 2000. 25
- Lakshmi H. Kantha and Carol Anne Clayson. An improved mixed layer model for geophysical applications. *Journal of Geophysical Research*, 99(C12):25235–25266, 1994. doi: 10.1029/94JC02257. 71
- A. Birol Kara, Harley E. Hurlburt, and Alan J. Wallcraft. Stability-dependent exchange coefficients for airsea fluxes*. *Journal of Atmospheric and Oceanic Technology*, 22(7):1080–1094, 2005. doi: 10.1175/JTECH1747.1. 24, 138
- J. B. Kelmp, R. Rotunno, and W. C. Skamarock. On the dynamics of gravity currents in a channel. *Journal of Fluid Mechanics Digital Archive*, 269:169–198, 1994. doi: 10.1017/S0022112094001527. URL <http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=352578&fulltextType=RA&fileId=S0022112094001527>. 204
- P. M. Knupp. Algebraic mesh quality metrics for unstructured initial meshes. *Finite Elements in Analysis and Design*, 39(3):207–216, 2003. 98
- Stephan C. Kramer, Colin J. Cotter, and Christopher C. Pain. Solving the poisson equation on small aspect ratio domains using unstructured meshes. *Ocean Modelling*, 2010. Submitted, preprint at <http://arxiv.org/abs/0912.1976>. 66
- D. Kuzmin. A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods. *J. Comp. and App. Math.*, 233(12):3077–3085, 2010. 39
- D. Lakehal, M. Meier, and M. Fulgosi. *Interface tracking towards the direct simulation of heat and mass transfer in multiphase flows*, volume 23. Elsevier Science Inc, 2002. 220
- LD Landau and EM Lifshitz. *Fluid mechanics. Volume 6 of Course of Theoretical Physics. Transl. from the Russian by JB Sykes and WH Reid*. Oxford, 1987. 13
- W. Large and S. Yeager. Diurnal to decadal global forcing for ocean and seaice models: the data sets and climatologies. Technical report, NCAR Technical Report TN-460+ST, 2004. 24, 138
- P. Laug and H. Borouchaki. Molecular surface modeling and meshing. *Engineering with Computers*, 18(3):199–210, 2002. 102

- H. Le, P. Moin, and J. Kim. Direct numerical simulation of turbulent flow over a backward-facing step. *J. Fluid Mech.*, 330:349–374, 1997. [213](#), [214](#)
- P. D. Ledger, K. Morgan, J. Peraire, O. Hassan, and N. P. Weatherill. The development of an *hp*-adaptive finite element procedure for electromagnetic scattering problems. *Finite Elements in Analysis and Design*, 39(8):751–764, 2003. doi: [10.1016/S0168-874X\(03\)00057-X](#). 14th Robert J. Melosh Competition. [95](#)
- C. K. Lee. Automatic metric 3D surface mesh generation using subdivision surface geometrical model. Part II: Mesh generation algorithm and examples. *International Journal for Numerical Methods in Engineering*, 56:1615–1646, 2003. [102](#)
- T.-H. Lee, Z. Zhou, and Y. Cao. Numerical simulations of hydraulic jumps in water sloshing and water impacting. *Journal of Fluids Engineering*, 124(1):215–226, 2002. doi: [10.1115/1.1436097](#). URL <http://link.aip.org/link/?JFEG/124/215/1>. [221](#), [225](#)
- B. P. Leonard. The ULTIMATE conservative difference scheme applied to unsteady one-dimensional advection. *Computer Methods in Applied Mechanics and Engineering*, 88(1):17–74, June 1991. ISSN 0045-7825. doi: [10.1016/0045-7825\(91\)90232-U](#). URL <http://www.sciencedirect.com/science/article/B6V29-48050F8-X/2/7bd4f5a59bfe567b5bdb9b36bd1898a1>. [47](#), [49](#)
- R. J. LeVeque. *Finite-volume methods for hyperbolic problems*. Cambridge University Press, Cambridge, 2002. [47](#), [56](#)
- X. Li, M. S. Shephard, and M. W. Beall. 3D anisotropic mesh adaptation by mesh modification. *Computer Methods in Applied Mechanics and Engineering*, 194:4915–4950, 2005. doi: [10.1016/j.cma.2004.11.019](#). [99](#)
- D. K. Lilly. A proposed modification of the germano subgrid-scale closure method. *Physics of Fluids A*, 4(3):633–635, 1992. ISSN 08998213. doi: [10.1063/1.858280](#). [77](#)
- P.L.F. Liu. *Advanced numerical models for simulating tsunami waves and runup*, volume 10. World Scientific Pub Co Inc, 2008. [230](#)
- R. Löhner. Robust, vectorized search algorithms for interpolation on unstructured grids. *Journal of Computational Physics*, 118(2):380–387, 1995. doi: [10.1006/jcph.1995.1107](#). [104](#)
- R. Löhner. Extensions and improvements of the advancing front grid generation technique. *Communications in Numerical Methods in Engineering*, 12(10):683–702, 1996. doi: [10.1002/\(SICI\)1099-0887\(199610\)12:10<683::AID-CNM983>3.0.CO;2-1](#). [102](#)
- Adrien Loseille and Frédéric Alauzet. Continuous Mesh Framework Part I: Well-Posed Continuous Interpolation Error. *SIAM Journal on Numerical Analysis*, 49(1):38–60, 2011a. doi: [10.1137/090754078](#). [98](#)
- Adrien Loseille and Frédéric Alauzet. Continuous Mesh Framework Part II: Validations and Applications. *SIAM Journal on Numerical Analysis*, 49(1):61–86, 2011b. doi: [10.1137/10078654X](#). [98](#), [101](#)
- A. E. H. Love. The yielding of the Earth to disturbing forces. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 82(551):73–88, 1909. [25](#)
- F. Lyard, F. Lefevre, T. Letellier, and O. Francis. Modelling the global ocean tides: Modern insights from FES2004. *Ocean Dynamics*, 56:394–415, 2006. [229](#)
- J. C. Martin and W. J. Moyce. Part IV. an experimental study of the collapse of liquid columns on a rigid horizontal plane. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 244(882):312–324, 1952. ISSN 00804614. URL <http://www.jstor.org/stable/91519>. [220](#)

- J. Mathieu and J. Scott. *An Introduction to Turbulent Flow*. Cambridge University Press, 2000. 72
- Trevor J. McDougall, David R. Jackett, Daniel G. Wright, and Rainer Feistel. Accurate and computationally efficient algorithms for potential temperature and density of seawater. *Journal of Atmospheric and Oceanic Technology*, 20(5):730–741, 2003. 18, 130
- M.G. McPhee. *Air-Ice-Ocean Interaction*. Springer, 2008. 78
- G. L. Mellor. *Introduction to Physical Oceanography*. Springer-Verlag, New York, 1996. 25
- George L. Mellor and Tetsuji Yamada. Development of a turbulence closure model for geophysical fluid problems. *Reviews of Geophysics*, 20(4):851–875, 1982. ISSN 8755-1209. doi: [10.1029/RG020i004p00851](https://doi.org/10.1029/RG020i004p00851). 70
- S. Micheletti and S. Perotto. Reliability and efficiency of an anisotropic Zienkiewicz–Zhu error estimator. *Computer Methods in Applied Mechanics and Engineering*, 195(9-12):799–835, 2006. 98
- F Nicoud and F. Durcos. Subgrid-scale stress modelling based on the square of the velocity gradient tensor. *Flow, Turbulence and Combustion*, 3:183–200, 1999. 75, 76
- S. J. Owen and S. Saigal. Surface mesh sizing control. *International Journal for Numerical Methods in Engineering*, 47(497):511, 2000. doi: [10.1002/\(SICI\)1097-0207\(20000110/30\)47:1/3;497::AID-NME781;3.0.CO;2-H](https://doi.org/10.1002/(SICI)1097-0207(20000110/30)47:1/3;497::AID-NME781;3.0.CO;2-H). 102
- C. C. Pain, A. P. Umpheby, C. R. E. de Oliveira, and A. J. H. Goddard. Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. *Computer Methods in Applied Mechanics and Engineering*, 190(29-30):3771–3796, 2001. doi: [10.1016/S0045-7825\(00\)00294-2](https://doi.org/10.1016/S0045-7825(00)00294-2). 75, 98, 99
- R.L. Panton. *Incompressible flow*. Wiley India Pvt. Ltd., 2006. 18, 216
- G. Parent, P. Dular, J. P. Ducreux, and F. Piriou. Using a Galerkin projection method for coupled problems. *IEEE Transactions on Magnetics*, 44(6):830–833, 2008. doi: [10.1109/TMAG.2008.915798](https://doi.org/10.1109/TMAG.2008.915798). 104
- I. R. Park, K. S. Kim, J. Kim, and S. H. Van. A volume-of-fluid method for incompressible free surface flows. *International Journal for Numerical Methods in Fluids*, 2009. doi: [10.1002/flid.2000](https://doi.org/10.1002/flid.2000). URL <http://dx.doi.org/10.1002/flid.2000>. xviii, 221, 225, 226
- J. Peraire and P. O. Persson. The Compact Discontinuous Galerkin (CDG) Method For Elliptic Problems. *Siam J. Sci. Comput.*, 30(4):1806–1824, 2008. doi: [10.1137/070685518](https://doi.org/10.1137/070685518). 43, 125
- J. Peraire, M. Vahdati, K. Morgan, and O.C Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, 72(2):449–466, 1987. doi: [10.1016/0021-9991\(87\)90093-3](https://doi.org/10.1016/0021-9991(87)90093-3). 98
- J. Peraire, J. Peiró, and K. Morgan. Finite element multigrid solution of Euler flows past installed aero-engines. *Computational Mechanics*, 11(5):433–451, 1993. doi: [10.1007/BF00350098](https://doi.org/10.1007/BF00350098). 104
- P.-O. Persson. Mesh size functions for implicit geometries and PDE-based gradient limiting. *Engineering with Computers*, 22(2):95–109, 2006. doi: [10.1007/s00366-006-0014-1](https://doi.org/10.1007/s00366-006-0014-1). 102
- M. D. Piggott, P. E. Farrell, C. R. Wilson, G. J. Gorman, and C. C. Pain. Anisotropic mesh adaptivity for multi-scale ocean modelling. *Philosophical Transactions of the Royal Society A*, 367(1907):4591–4611, 2009. doi: [10.1098/rsta.2009.0155](https://doi.org/10.1098/rsta.2009.0155). 99
- Ugo Piomelli, Thomas Zang, Speziale Charles, and Yousuff Hussaini. On the large-eddy simulation of transitiona wall-bounded flows. *Physics of Fluids A*, 2:257–265, 1990. 76
- Stephen B. Pope. *Turbulent Flows*. Cambridge University Press, 2000. 74, 76, 77

- E. E. Popova, A. C. Coward, G. A. Nurser, B. de Cuevas, M. J. R. Fasham, and T. R. Anderson. Mechanisms controlling primary and new production in a global ecosystem model – part i: Validation of the biological simulation. *Ocean Science*, 2(2):249–266, 2006. doi: [10.5194/os-2-249-2006](https://doi.org/10.5194/os-2-249-2006). 83
- A. Prosperetti and G. Tryggvason. *Computational Methods for Multiphase Flow*. Cambridge University Press, 2007. 26
- D. T. Pugh. *Tides, Surges and Mean Sea-Level: A Handbook for Engineers and Scientists*. Wiley, Chichester, 1987. 25
- W. H. Raymond and A. Garder. Selective damping in a Galerkin method for solving wave problems with variable grids. *Monthly weather review*, 104(12):1583–1590, 1976. 33
- C. Rousset, M.-N. Houssais, and E. P. Chassignet. A multi-model study of the restratification phase in an idealized convection basin. *Ocean Modelling*, 26:115–133, 2009. doi: [10.1016/j.ocemod.2008.08.005](https://doi.org/10.1016/j.ocemod.2008.08.005). 227
- Yousef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993. ISSN 1064-8275. doi: <http://dx.doi.org/10.1137/0914028>. 65
- Pierre Sagaut. *Large Eddy Simulation for Incompressible Flows*. Springer, 1998. ISBN 3-540-43753-3. 74
- M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher. Benchmark computations of laminar flow around a cylinder. In E. H. Hirschel et al., editor, *Flow Simulation with High-Performance Computers, II: DFG Priority Research Program Results 1993–1995*, volume 52 of *Notes on Numerical Fluid Mechanics*, pages 547–566. Friedrich Vieweg & Sohn Verlagsgesellschaft mbH, 1996. 218
- E. W. Schwiderski. On charting global ocean tides. *Rev. Geophys. Space Phys.*, 18:243–268, 1980. 229
- S.J. Sherwin, R.M. Kirby, J. Peirò, R.L. Taylor, and O.C. Zienkiewicz. On 2D elliptic discontinuous Galerkin methods. *Int. J. Num. Meth. Eng.*, 65(5):752–784, 2006. 42
- J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. <http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf>, 1994. 65
- C. K. Shum, P. L. Woodworth, O. B. Andersen, G. D. Egbert, O. Francis, C. King, S. M. Klosko, C. Le Provost, X. Li, J.-M. Molines, M. E. Parke, R. D. Ray, M. G. Schalx, D. Stammer, C. C. Tierney, P. Vincent, and C. I. Wunsch. Accuracy assessment of recent ocean tide models. *J. Geophys. Res.*, 102(C11):25,173–25,19, 1997. 228
- J. E. Simpson. *Gravity Currents in the Environment and the Laboratory*. Ellis Horwood Ltd, 1987. 203
- J.E. Simpson and R. E. Britter. The dynamics of the head of a gravity current advancing over a horizontal surface. *jfm*, 94(3):477–495, 1979. xvii, 206
- R. B. Simpson. Anisotropic mesh transformations and optimal error control. *Applied Numerical Mathematics*, 14(1-3):183–198, 1994. doi: [10.1016/0168-9274\(94\)90025-6](https://doi.org/10.1016/0168-9274(94)90025-6). 97
- J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly weather review*, 91(3):99–164, 1963. 75
- Klaus Stüben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128:281, 2001. 66
- P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM Journal on Numerical Analysis*, 21(5):995–1011, October 1984. ISSN 00361429. doi: [10.2307/2156939](https://doi.org/10.2307/2156939). URL <http://www.jstor.org/stable/2156939>. 47
- Ulrich Trottenberg, Cornelis Oosterlee, and Anton Schüller. *Multigrid*. Elsevier Academic Press, 2001. 66

- Yu-Heng Tseng and Joel H. Ferziger. Mixing and available potential energy in stratified flows. *Physics of Fluids*, 13(5):1281–1293, 2001. doi: [10.1063/1.1358307](https://doi.org/10.1063/1.1358307). 205
- M. N. Tsimplis, R. Proctor, and R. A. Flather. A two-dimensional tidal model for the Mediterranean Sea. *J. Geophys. Res.*, 100(C8):16,223–16,239, 1995. xviii, 228, 229, 230, 231, 232, 233
- J. S. Turner. *Buoyancy effects in fluids*. Cambridge University Press, 1973. 204
- L. Umlauf and H. Burchard. A generic length-scale equation for geophysical turbulence models. *Journal of Marine Research*, 61:235–265(31), 2003. doi: [10.1357/002224003322005087](https://doi.org/10.1357/002224003322005087). 69, 70
- L. Umlauf and H. Burchard. Second-order turbulence closure models for geophysical boundary layers. A review of recent work. *Continental Shelf Research*, 25(7-8):795–827, 2005. doi: [10.1016/j.csr.2004.08.004](https://doi.org/10.1016/j.csr.2004.08.004). 71
- S. M. Uppala, P. W. Kllberg, A. J. Simmons, U. Andrae, V. Da Costa Bechtold, M. Fiorino, J. K. Gibson, J. Haseler, A. Hernandez, G. A. Kelly, X. Li, K. Onogi, S. Saarinen, N. Sokka, R. P. Allan, E. Andersson, K. Arpe, M. A. Balmaseda, A. C. M. Beljaars, L. Van De Berg, J. Bidlot, N. Bormann, S. Caires, F. Chevallier, A. Dethof, M. Dragosavac, M. Fisher, M. Fuentes, S. Hagemann, E. Hlm, B. J. Hoskins, L. Isaksen, P. A. E. M. Janssen, R. Jenne, A. P. Mcnally, J.-F. Mahfouf, J.-J. Morcrette, N. A. Rayner, R. W. Saunders, P. Simon, A. Sterl, K. E. Trenberth, A. Untch, D. Vasiljevic, P. Viterbo, and J. Woollen. The era-40 re-analysis. *Quarterly Journal of the Royal Meteorological Society*, 131(612): 2961–3012, 2005. doi: [10.1256/qj.04.176](https://doi.org/10.1256/qj.04.176). 24
- M.-G. Vallet. Génération de maillages anisotropes adaptés – application à la capture de couches limites. Technical Report RR-1360, INRIA Rocquencourt, Rocquencourt, Le Chesnay, France, 1990. URL <http://www.inria.fr/RRRT/RR-1360.html>. 97
- P. Vanek, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56:179–196, 1996. 66
- Y. Vasilevskii and K. Lipnikov. An adaptive algorithm for quasioptimal mesh generation. *Computational Mathematics and Mathematical Physics*, 39(9):1468–1486, 1999. 98, 99
- R. Verfürth. *A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*. John Wiley and Sons Ltd., 1996. ISBN 0-471-96795-5. 95
- N.P. Waterson and H. Deconinck. Design principles for bounded higher-order convection schemes - a unified approach. *Journal of Computational Physics*, 224(1):182–207, May 2007. ISSN 0021-9991. doi: [10.1016/j.jcp.2007.01.021](https://doi.org/10.1016/j.jcp.2007.01.021). URL <http://www.sciencedirect.com/science/article/B6WHY-4MYMG2M-5/2/fc1a18cfa833cd1434f4dc8bf2bdced4>. 47
- M. R. Wells. *Tidal modelling of modern and ancient seas and oceans*. PhD thesis, Imperial College London, 2008. xviii, 24, 25, 139, 228, 229, 230, 233
- M. R. Wells, P. A. Allison, M. D. Piggott, G. J. Gorman, G. J. Hampson, C. C. Pain, and F. Fang. Numerical modelling of tides in the Pennsylvanian Midcontinent Seaway of North America with implications for hydrography and sedimentation. *Journal of Sedimentary Research*, 77:843–865, 2007. 25
- D.C. Wilcox. *Turbulence modeling for CFD*. DCW industries La Canada, CA, 1998. 73
- C. R. Wilson. *Modelling multiple-material flows on adaptive, unstructured meshes*. PhD thesis, Imperial College London, 2009. 47, 49, 51, 53, 103
- Kraig B. Winters and Eric A. D’Asaro. Diascalar flux and the rate of fluid mixing. *Journal of Fluid Mechanics*, 317(1):179–193, 1996. doi: [10.1017/S0022112096000717](https://doi.org/10.1017/S0022112096000717). 205

-
- Kraig B. Winters, Peter N. Lombard, James J. Riley, and Eric A. D'Asaro. Available potential energy and mixing in density-stratified fluids. *Journal of Fluid Mechanics*, 289(-1):115–128, 1995. doi: [10.1017/S002211209500125X](https://doi.org/10.1017/S002211209500125X). URL <http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=354020&fulltextType=RA&fileId=S002211209500125X>. 158, 161, 205
- Z. Q. Zhou, J. O. De Kat, and B. Buchner. A nonlinear 3-D approach to simulate green water dynamics on deck. In *Report No. 82000-NSH*, volume 7, Nantes, France, 1999. xvii, xviii, 220, 221, 222, 225, 226

Appendix A

About this manual

A.1 Introduction

This document attempts to give an introduction to the use of the Fluidity/ICOM code for CFD and ocean modelling applications. The layout of this manual is briefly covered in the overview at the beginning of this document.

Although this document may of course be printed, viewing it on screen may be wise as it allows colour images to be viewed, and links between sections and parameters should work. Also, where possible figures are 'vectorised' so that if viewed electronically it is possible to zoom right in to see the structure of meshes for example.

This manual is very much a work in progress. Therefore spelling, grammar, and accuracy can not be guaranteed. Users with commit access to the Fluidity source tree are encouraged to make contributions directly. Other users are invited to email comments, corrections and contributions to m.d.piggott@imperial.ac.uk

A further source of material may be found at <http://amcg.ese.ic.ac.uk/> This points to a collection of wiki web pages that all users are able to update and add to.

A.2 Audience and Scope

The manual is primarily designed to enable Fluidity users to run Fluidity effectively. As such, this is the appropriate place for documentation concerning the available configuration options of the model and the correct method of employing them. It is also the correct place to document the mathematical formulation of the model and the equations which it solves. Other matters which should be covered include input and output formats, checkpointing and visualisation.

This is not generally the appropriate forum for low-level documentation directed at model developers. Information concerning the finite element method and its implementation in Fluidity should be placed in the Femtools manual while other implementation details could be placed on the AMCG wiki.

A.3 Style guide

A.3.1 Headings

Headings should be typeset with sentence capitalisation. That is to say, only those words which would be capitalised if the heading were an ordinary sentence are capitalised.

A.3.2 Language

The manual is written in British English. This, among many distinctions from our cousins across the Pond means:

- centre not center.
- visualise not visualize.
- licence for the noun, license for the verb.

A.3.3 Labelling

Sections, tables, figures and equations should be labelled consistently.

Sections should be labelled as `\label{sec:unique_section_name}`, tables should be labelled as `\label{tab:unique_table_name}`, figures as `\label{fig:unique_figure_name}`, and equations as `\label{eq:unique_equation_name}`.

Note that all label names should be unique across the manual.

A.3.4 Images

The manual is designed to be compilable to both PDF and html. This creates particular challenges when incorporating images. One approach which is particularly appropriate for diagrams and other images annotated with text or equations is to generate or annotate the figures using xfig. Xfig files may be automatically converted to Postscript for the PDF document and png for the html version. If the “special” attribute is set on text in the xfig document, then that text will be rendered in L^AT_EX. This in particular enables equations to be included in figures in a manner consistent with the equations in the text.

A.3.4.1 Including xfig images

This manual defines the command `\xfig{basename}` which will import `basename.pdf` for pdf output and `basename.png` for web output. Authors should ensure that their xfig file has the name `basename.fig` and that `basename` is added to the `XFIG_IMAGES` variable in the Makefile. This will cause the commands `make fluidity_manual.pdf` and `make fluidity_manual.html` to also generate the pdf and png versions of the figure respectively.

It will be observed that the `\xfig` command does not take any arguments for figure size. This is a deliberate decision designed to ensure that the font size matches between the figure and the text. Figures should instead be appropriately sized in xfig.

A.3.4.2 Including other figures

For other figures, the command `\fig[options]{basename}` is provided. In this case, it is the author's responsibility to provide both `basename.pdf` and `basename.png` files. Please add `basename` to the `IMAGES` variable in the `Makefile`. This will cause the image files to become dependencies of the compiled manual.

If no `basename.png` is available, then the `\pdffig[options]{basename}` should be used instead.

The options provided to `\fig` are passed straight to `\includegraphics` and may therefore include any options which are legal in that context including resizing options.

A.3.5 flml options

The `\option` command is provided to format Fluidity option paths. Options should be formatted according to normal Spud conventions however it will frequently be desirable to show partial option paths not starting from the root. In this circumstance, an ellipsis should be used to show an unknown path component. For example, the mesh element of some prescribed field would be `.../prescribed/mesh` which can be input in \LaTeX as `\option{\ldots/prescribed/mesh}`

A.3.6 Generating pdf and html output

The manual may be compiled to both pdf and html. For the former, type:

```
make fluidity_manual.pdf
```

and for the latter type:

```
make fluidity_manual.html
```

It may sometimes be necessary to introduce content which should only be rendered in one or other output format. For example, long option paths frequently defeat \LaTeX 's line breaking algorithm so it may be necessary to force line breaks in the pdf document. Since the browser is responsible for line breaks in html, it would be inappropriate to force a linebreak in the html output. For this purpose, the manual provides the commands `\ifhtml{content for html}{content for pdf}` as well as the commands `\onlyhtml` and `\onlypdf`. The latter two commands take a single argument which is only rendered for the applicable output.

A.3.7 Representing source code

Source code and commands entered in the shell can be typeset using the `lstlisting` environment. The environment typesets its argument literally so unlike normal \LaTeX , spaces and carriage returns are replicated in the output. The environment takes optional configuration parameters of which the most important is `language` which is used to select the programming language. \LaTeX will highlight the syntax of the chosen language. Inline commands can be typeset using `\lstinline[language=TeX]+command+` substituting any applicable language.

A.3.7.1 Shell commands

For shell commands, `language` should be set to `bash`. For example:

```
\begin{lstlisting}[language=bash]
dham@popper traffic > ls
box.ele   fluidity.err-0  Makefile        traffic_1.vtu   traffic.xml
box.face  fluidity.log-0  src             traffic.flml    vaf.bin
box.node  gmon.out        traffic_0.vtu   traffic.stat    vaf.dat
\end{lstlisting}
```

will be rendered:

```
dham@popper traffic > ls
box.ele   fluidity.err-0  Makefile        traffic_1.vtu   traffic.xml
box.face  fluidity.log-0  src             traffic.flml    vaf.bin
box.node  gmon.out        traffic_0.vtu   traffic.stat    vaf.dat
```

A.3.7.2 Other languages

The other languages which are currently enabled are TeX, for L^AT_EX, Python, Make and XML. Many other languages are supported by the listings package.

A.3.8 Bibliography

Citations from the literature should be included whenever relevant. When formatting entries in the bibliography database, `bibliography.bib`, the preferred key is the first author's surname in lower case followed by the full year of publication. For example `ham2009`. The bibliography database should be sorted alphabetically by key.

The manual uses an author-date citation style which means that it is important to use the correct combination of `\cite`, `\citep` and `\citet`. See the L^AT_EX `natbib` package documentation for more details.

A.3.9 Mathematical notation conventions

A.3.9.1 Continuous Vectors and tensors

There are two conceptually different forms of vector and tensor which occur in the finite element method. The first is for quantities, such as velocity, which are vector-valued in the continuum. These should be typeset in italic bold: \mathbf{u} . The `\vec` command has been redefined for this purpose so a vector quantity named `b` would be typed `\vec{b}`. However, a large number of frequently-used vector quantities have convenience functions pre-defined in `notation.tex`. These have the name `\bmn` where `n` is the symbol to be typeset. Examples include `\bmu` (\mathbf{u}) and `\bmphi` ($\boldsymbol{\varphi}$).

Continuous tensors are represented using a double overbar: $\overline{\overline{\tau}}$. The `\tensor` command is provided for this purpose. Once again, convenience functions are provided for common tensors, this time with the form `\ntens` for example `\tautens` ($\overline{\overline{\tau}}$) and `\ktens` ($\overline{\overline{k}}$).

A.3.9.2 Discrete vectors and matrices

Vectors composed of the value of a field at each node and matrices mapping between discrete spaces should be typeset differently from continuous vectors and tensors. Discrete vectors should be typeset with an underline using the `\dvec` command. Note that the convention in Fluidity is that vector

fields are represented as scalar sums over vector valued basis functions so the correct representation of the discrete velocity vector is $\text{\dvec}\{\text{\bmu}\}(\underline{u})$.

Matrices should be typeset as upright upper case letters. The `\mat` command is available for this purpose. For example `\mat{M}` produces M .

A.3.9.3 Derivatives

The full derivative and the material derivative should be typeset using an upright d and D respectively. The `\d` and `\D` commands are provided for this purpose. There are also a number of functions provided for typesetting derivatives. Each of these functions has one compulsory and one optional argument. The compulsory argument is the function of which the derivative is being taken, the optional argument is the variable with respect to which the derivative is being taken. So, for example `\ppt[q]{y}` gives:

$$\frac{\partial y}{\partial q}$$

While simple `\ppt{}` gives:

$$\frac{\partial}{\partial t}$$

Table A.1 shows the derivative functions available.

command	example
<code>\ddx[]{}{}</code>	$\frac{d}{dx}$
<code>\ddxx[]{}{}</code>	$\frac{d^2}{dx^2}$
<code>\ddt[]{}{}</code>	$\frac{d}{dt}$
<code>\ddtt[]{}{}</code>	$\frac{d^2}{dt^2}$
<code>\ppx[]{}{}</code>	$\frac{\partial}{\partial x}$
<code>\ppxx[]{}{}</code>	$\frac{\partial^2}{\partial x^2}$
<code>\ppt[]{}{}</code>	$\frac{\partial}{\partial t}$
<code>\pptt[]{}{}</code>	$\frac{\partial^2}{\partial t^2}$
<code>\DDx[]{}{}</code>	$\frac{D}{Dx}$
<code>\DDxx[]{}{}</code>	$\frac{D^2}{Dx^2}$
<code>\DDt[]{}{}</code>	$\frac{D}{Dt}$
<code>\DDtt[]{}{}</code>	$\frac{D^2}{Dt^2}$

Table A.1: Functions for correctly typesetting derivatives.

A.3.9.4 Integrals

Integrals in any number of dimensions should be typeset with an integral sign and no measure (*i.e.*, no dx or dV). The domain over which the integral is taken should be expressed as a subscript to the integral sign itself. The integral of ψ over the whole domain will therefore be written as:

$$\int_{\Omega} \psi.$$

A.3.9.5 Units

Units should be typeset in upright font. The \LaTeX package `units` does this for you automatically. The correct syntax is `\unit[value]{unit}`. For example `\unit[5]{m}` produces 5 m. There are a number of convenience functions defined for the manual to make this job easier. These are shown in table A.2. Providing the value as an argument to the unit ensures that the spacing between the value and the unit is correct and will not break over lines. The value is an optional argument so if there is no value, just leave it out. `units` does the right thing in both text and math modes. Other convenience functions can easily be added to `notation.tex`.

command	example
<code>\m[length]</code>	1 m
<code>\km[length]</code>	1 km
<code>\s[time]</code>	1 s
<code>\ms[speed]</code>	1 m s ⁻¹
<code>\mss[accel]</code>	1 m s ⁻²
<code>\K[temp]</code>	1 K
<code>\PSU[salin]</code>	1 PSU
<code>\Pa[press]</code>	1 Pa
<code>\kg[mass]</code>	1 kg
<code>\rads[ang_vel]</code>	1 rad s ⁻¹
<code>\kgmm[density]</code>	1 kg m ⁻²

Table A.2: Convenience functions for physical units

A.3.9.6 Abbreviations in formulae

Abbreviations in formulae should be typeset in upright maths mode using `\mathrm`. For example `F_{\mathrm{wall}}` (F_{wall}).

Appendix B

The Fluidity Python state interface

Fluidity incorporates the Python interpreted programming language as a mechanism for users to customise the model without editing the main Fortran source of the model. There are, in fact, two distinct Python interfaces presented by Fluidity. The first allows users to specify prescribed fields and the initial conditions of prognostic fields by providing a Python function of space and time. This interface is documented in section 8.6.2.2. The present chapter documents the much more comprehensive *Python state* interface which gives the user access to the complete current system state. This may be used to specify diagnostic fields as a function of the values of other fields. In particular, this is used by embedded models such as the biology model to specify the coupling between different model variables.

B.1 System requirements

The Fluidity Python interface requires Python to be installed as well as NumPy, the fundamental Python numerical package. To check that Fluidity has been build with Python, run:

```
fluidity -h
```

and check for the lines:

```
Python support          yes
Numpy support           yes
```

Python will be installed on any modern Unix machine but NumPy may need to be specially installed. Ubuntu and Debian users can do so with the `python-numpy` package.

Fluidity also requires access to its own Python modules which are stored in the `python` directory in the Fluidity source tree. To ensure that these are visible to Fluidity at runtime, users should add this directory to the `PYTHONPATH` environment variable. For example:

```
export PYTHONPATH=<my_fluidity>/python/:$PYTHONPATH
```

where `<my_fluidity>` is the location of the Fluidity source tree.

B.2 Data types

The data classes of most importance to users are `State` and `Field`. Between them, these present all of the field data in Fluidity in a readily accessible way.

node shape	ScalarField values	VectorField values	TensorField values
scalar	scalar	(dim)	(dim, dim)
sequence	(len(node))	(len(node), dim)	(len(node), dim, dim)

Table B.1: The shapes of the return value of `node_val` and the `val` argument to `set` and `addto`. `dim` is the data dimension of the field.

B.2.1 Field objects

The `Field` class defines data types for Fluidity fields. The fields are implemented as wrappers around the internal data structures in Fluidity so the field values are always current and changes to field values are seen by the whole model. Field objects are actually of an appropriate subclass: `ScalarField`, `VectorField` or `TensorField`, however, these classes differ only in the shape of arguments to their data routines.

Field objects support the following methods and attributes:

node_count The number of nodes in the field.

element_count The number of elements in the field.

dimension The data dimension (not for `ScalarField` objects).

node_val(node) Return the value of the field at `node(s)`. If `node` is a scalar then the result is the value of the field at that one node. If `node` is a sequence then the result is the value of the field at each of those nodes. The shape of the result is given for each case below.

set(node, val) Set the value(s) of the field at the `node(s)` specified. If `node` is a scalar then the value of the field at that one node is set. If `node` is a sequence then the value of the field at each of those nodes is set. The shape of `val` must be as given below.

addto(node, val) Add value(s) to the field at the `node(s)` specified. If `node` is a scalar then the value of the field at that one node is modified. If `node` is a sequence then the value of the field at each of those nodes is modified. The shape of `val` must be as given below.

ele_loc(ele_number) Return the number of nodes in element `ele_number`.

ele_nodes(ele_number) Return the indices of the nodes in element `ele_number`.

ele_val(ele_number) Return the value of the field at the nodes in element `ele_number`. This is equivalent to calling `field.node_val(field.ele_nodes(ele_number))`.

ele_region_id(ele_number) Return the `region_id` of element `ele_number`. This can be used to specify diagnostics which only apply over some portion of the domain.

B.2.2 State objects

A `State` is a container for all of the fields in a single material phase. The fields in the material phase are accessed by the names given in the Fluidity options file. `State` objects contain the following data. In each case it is assumed that `s` is an object of class `State`.

scalar_fields This is a dictionary of all the scalar fields in the material phase. For example, if the state contains a field named "Temperature" then it can be accessed with `s.scalar_fields['Temperature']`.

vector.fields This is a dictionary of all the vector fields in the state. For example, the coordinate field can be accessed using `s.vector_fields['Coordinate']`.

tensor.fields This is a dictionary of all the tensor fields in the state. For example, if there is a field called “Viscosity”, it can be accessed as `s.tensor_fields['Viscosity']`.

A useful debugging facility is that a `State` can be printed from within python (`print `s``) which results in a list of the fields and meshes in that state.

B.3 Predefined data

There will always be a variable named `state` of type `states` which will contain all of the fields in the material phase of diagnostic field currently being calculated.

There will also always be a dictionary called `states` which will contain at least the current material phase. If the interface is being used to specify a diagnostic field and the `.../diagnostic/material_phase_support` attribute is set to “multiple”, then all of the material phases will be present in the `states` dictionary. For example, if there are two material phases, “Air” and “Water”, then the air velocity will be present as the field `states['Air'].vector_fields['Velocity']`.

In addition to these states, the variable `field` is preset to the current diagnostic to be set (this does not apply in situations such as the biology model in which multiple fields are to be set in a single Python calculation).

The variables `time` and `dt` are pre-set to the current simulation time and the timestep respectively.

B.4 Importing modules and accessing external data

The Python interpreter run by Fluidity is exactly the same as that used by the Python command line. Therefore essentially anything which is legal in Python is legal in the Fluidity Python interface. In particular, standard and user-defined modules can be imported including modules for reading external data sources. The current directory is automatically added to the Python search path when Fluidity is run so user defined modules placed in the directory from which Fluidity is launched will be found.

B.5 The persistent dictionary

All of the data in the Fluidity interpreter is wiped after every field calculation. Usually this is desirable as it prevents the code for different diagnostics interfering. If it is necessary to store data in the Python interpreter after flow passes back to the main model, these data items can be stored in the dictionary named `persistent` under any key the user chooses.

It is *not* safe to store fields extracted from the states in the persistent dictionary.

B.6 Debugging with an interactive Python session

It is possible to launch an interactive Python session at runtime from within Fluidity. This depends on `iPython` being installed on the system and is achieved by placing the following code in the `flml` file at the point within Python at which you wish to stop:

```
import fluidity_tools
fluidity_tools.shell() ()
```

Note the double brackets `() ()` after `fluidity_tools.shell!` From within this Python session, you can examine or set any variables which are currently visible. As well as using this to debug straightforward syntax errors, you can trap much more complex errors by placing the commands inside an **if** statement or a **try...except** clause. As soon as you leave the Python shell, Fluidity execution will continue.

It is also possible to launch an interactive Python session from within the simple Python interface for prescribed fields, however resuming execution afterwards currently causes a crash.

B.7 Limitations

The Python state interface is essentially driven at fields which are pointwise functions of other fields. This is only straightforward in the situation where all the fields concerned are on the same mesh. Where this is not the case, there are two different work-arounds which can be used.

If the field to be calculated is discontinuous or periodic and it is desired to use continuous and/or non-periodic data in the calculation (for example because the field is a function of position), this can be achieved by looping over the elements and setting the value at each node of each element rather than directly looping over all the nodes in the field. This is possible because the element numbering and the local numbering of nodes within elements is common between meshes of different continuity.

In other cases, such as the case in which a diagnostic field is a function of a field of different polynomial order, it may be necessary to introduce an additional diagnostic field which is the Galerkin projection of the second field onto the same mesh as the diagnostic field.

Appendix C

External libraries

C.1 Introduction

This appendix gives an overview of the external libraries that are required to build and run Fluidity. Fluidity's development strategy has taken a conscious decision to employ external libraries wherever it is possible and beneficial to do so. This industry-standard approach both short-cuts the development process by making use of the work and expertise of external projects, and in most cases provides a better solution than could be implemented by the Fluidity development team.

C.2 List of external libraries and software

Fluidity requires the following libraries and supporting software to build and run:

- Fortran 90 and C++ compiler (tested gfortran 4.5 and 4.6, Intel 11.1 for versions above 11.1.073)
- BLAS (tested netlib, ATLAS, MKL)
- LAPACK (tested netlib, ATLAS, MKL)
- XML2 (tested version 2.6)
- MPI2 implementation (tested OpenMPI version 1.4.3)
- PETSc (tested versions 3.0 and 3.1)
- ParMetis (tested version 3.2)
- ARPACK (tested version 96)
- Python (tested version 2.7)
- NumPy (tested version 1.6.1)
- SciPy (tested version 0.9.0)
- Trang (tested version 20030619, any recent version should work)
- VTK (tested version 5.6)
- Zoltan (tested version 3.501)

Fluidity recommends also making available the following:

- CGNS (tested version 2.5)
- NetCDF (tested version 4.0)
- UDUnits (tested version 2.1.23)
- Bazaar (tested version 2.4.0)

C.3 Installing required libraries on Debian or Ubuntu

By far the easiest way to obtain all the supporting libraries and other software required to build and run Fluidity is to make use of the Fluidity packages available from the Launchpad fluidity-core PPA. These are available for as many Ubuntu versions as Fluidity currently supports; at the time of writing, 10.10 (Maverick) and newer, though support for 10.10 will end as of April 2012 when central Ubuntu support ends.

BE AWARE: AMCG packages are provided for use at your own risk and without warranty. You should ensure that any packages installed from external repositories are not going to adversely affect your system before installing them!

To access the repository containing the Fluidity support packages, you will need to run:

```
sudo apt-add-repository -y ppa:fluidity-core/ppa
```

You will then need to update your system and install the fluidity-dev package, which depends on all the other software required for building Fluidity:

```
sudo apt-get update
sudo apt-get install fluidity-dev
```

To benefit from the environment modules supplied from AMCG you may want to add the following lines to your `/etc/bash.bashrc` file:

```
if [ -f /usr/share/Modules/init/bash ]; then
    . /usr/share/Modules/init/bash
fi
```

New bash shells should automatically inherit the modules environment now, and you should be able to type:

```
module load petsc-gcc4
```

to enable use of PETSc, required for building Fluidity.

C.4 Manual install of external libraries and software

Competent systems administrators should find it relatively straightforward to install the supporting software and external libraries required by Fluidity on most modern UNIX systems and compute clusters. The following instructions are intended to help with this process, offering hints and tips to speed up the deployment process.

In most cases, modern Linux systems will supply some if not most of the required packages without needing to resort to compiling them from source.

C.4.1 Build environment

The following compile instructions assume that you have set up a basic bash environment containing a few key environment variables. Set `WORKING` to be the root of your working area which the subsequent variables will refer to:

```
export WORKING="/path/to/my/data"

export PATH="$WORKING/fluidity/bin:$PATH"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$WORKING/fluidity/lib"
export CFLAGS="-L$WORKING/fluidity/lib"
export FFLAGS="-L$WORKING/fluidity/lib"
export CPPFLAGS="-I$WORKING/fluidity/include"
export LDFLAGS="-L$WORKING/fluidity/lib"
```

csh users will want to alter all export commands to the corresponding setenv syntax throughout this appendix.

Throughout this section where standard configure, make, and install is referred to it is assumed to mean running the following commands:

```
./configure --prefix=$WORKING/fluidity
make
make install
```

Where the source directory for a package is referred to it is assumed to mean the root directory created when the package is uncompressed.

C.4.2 Compilers

The Fluidity build process requires working Fortran 90 and C++ compilers.

Fluidity has been tested with gfortran ≥ 4.4 and Intel 11.1 for versions $\geq 11.1.073$. It is not supported for gfortran ≤ 4.3 or Intel $\leq 11.1.073$, and if using gfortran some features are not available except for gfortran ≥ 4.5 .

Unsupported compilers generally have incorrect Fortran 90 implementations for which bug reports have been submitted and implemented in later versions where applicable. Bug reports have been submitted for Portland group compilers but not yet implemented. Fluidity is not yet tested against Intel 12.x compilers.

Fluidity has been tested with both GNU and Intel C++ compilers at corresponding versions to the tested and known-good Fortran 90 compilers.

If you do not already have compilers suitable for building Fluidity, GCC is freely available and is possible to build from source with sufficient disk space and time.

C.4.2.1 GMP and MPFR

GMP (tested for Fluidity with gmp-5.0.2) and MPFR (tested for Fluidity with mpfr-2.4.2) are needed for the GCC 4.x build if you do not already have them. Download GMP from <http://gmplib.org/> and build it in the source directory, appending `--enable-cxx` to the standard configure, then running the standard make and install.

Once GMP has been installed, download MPFR from <http://www.mpfr.org/mpfr-current/>

and build it in the source directory, appending `--with-gmp=\$WORKING/fluidity` to the standard configure, then running the standard make and install.

If you have no compilers at all, you may need to download GMP and MPFR as pre-built binaries. If you have any compilers, even if not ones which support building Fluidity, you should be able to build GMP and MPFR, and then go on to build GCC.

C.4.2.2 GCC

GCC (tested for Fluidity with gcc-4.4.1, known-working with Ubuntu builds of 4.5 and 4.6) can be downloaded from the UK mirror at <http://gcc-uk.internet.bs/> Before the build, make sure that the GMP and MPFR libraries are on LD_LIBRARY_PATH or the stage 1 configure will fail even if `--with-[gmp|mpfr]` is supplied.

Also note that the build needs to be in a target build directory, NOT in the source directory, or again the build will fail with definition conflicts against the system includes.

Finally, the java build appears to be buggy in (at least) 4.4 and is enabled by default, but Fluidity only needs c, c++, and fortran, so just specify those.

Make a target build directory which is OUTSIDE the source tree, then build with the following configure:

```
/path/to/gcc/source/configure --prefix=$WORKING/fluidity
--with-gmp=$WORKING/fluidity --with-mpfr=$WORKING/fluidity
--enable-languages=c,c++,fortran
```

followed by the standard make and install.

The GCC4 build has not been tested by the Fluidity builders in an environment with no compilers at all and thus the ability or otherwise of the build to bootstrap itself is not known.

C.4.2.3 OpenMPI

Finally, you'll need an MPI implementation to wrap your compiler for the Fluidity build, which lets you spawn parallel runs from the compiled Fluidity binary. Any full MPI implementation should be sufficient, though Fluidity is generally tested using OpenMPI. Please note that Clustervision-supplied clusters generally ship with broken MPI C++ support and will need attention before Fluidity can be compiled. Fluidity is no longer generally supported as non-MPI code as it is assumed that serial runs will be precursors to large parallel runs and will be built with MPI enabled for later use.

OpenMPI (tested for Fluidity with openmpi-1.4.3) can be downloaded from <http://www.open-mpi.org/software/ompi/v1.4/>. It is built in the source directory with the standard configure, make, and install.

C.4.3 Python

Python is widely used within Fluidity for user-defined functions and for diagnostic tools and problem setup, and currently tested up to Python version 2.7. Earlier Python version may be suitable for use but may lack later functionality. Python extensions required are: setuptools for Fluidity builds, Python-4suite and Python-XML for options file parsing, and NumPy for custom function use within Fluidity.

If you do not have a working version of Python it can be built from source.

C.4.3.1 Readline

Readline (tested version 6.0) is not strictly needed for Python to build but is very handy if you want to make use of things like Python command history. Download readline from <http://ftp.gnu.org/pub/gnu/readline/> and then add `-fPIC` to `CFLAGS` and `FFLAGS` for the duration of this build with:

```
export CFLAGS="$CFLAGS_-fPIC"  
export FFLAGS="$FFLAGS_-fPIC"
```

These can be returned to their default values after the readline build. Building with `-fPIC` shouldn't be necessary but seems to be required by the later Python build.

Build readline in the source directory, appending `--disable-shared` to the standard configure and then running the standard make and install.

C.4.3.2 Python

Python (tested version 2.6) can be downloaded from <http://www.python.org/download/> and built in the source directory with the standard configure, make, and install, adding `--enable-shared` to the configure flags.

C.4.3.3 Setuptools

Setuptools (tested version 0.6c11) can be downloaded from <http://pypi.python.org/pypi/setuptools> and installed from the appropriate egg file with:

```
sh ./setuptools-0.6c11-py2.6.egg --prefix=$WORKING/fluidity
```

Use the appropriate filename for the Python egg you have downloaded.

C.4.3.4 NumPy

NumPy (tested version 1.6.0) can be installed through `easy_install` which was provided by setuptools, using:

```
easy_install numpy
```

C.4.4 Numerical Libraries

BLAS and LAPACK are required for efficient linear algebra methods within Fluidity, and are tested with the netlib, ATLAS, and MKL implementations, though any standard BLAS or LAPACK implementation should be sufficient for Fluidity. PETSc is required to provide matrix solvers, and ParMetis is required for mesh partitioning and sparse matrix operations.

C.4.4.1 BLAS

BLAS can be downloaded from <http://www.netlib.org/blas/> (for netlib BLAS), <http://sourceforge.net/projects/math-atlas/files/> (for ATLAS), or combined with commercially available compilers such as MKL from Intel.

BLAS is built in the source directory after editing the following entries in the `make.inc` file:

```

FORTRAN = gfortran
LOADER  = gfortran
OPTS    = -O3 -fPIC

```

Then run:

```

make
cp blas_LINUX.a $WORKING/fluidity/lib/libblas.a

```

C.4.4.2 LAPACK

LAPACK can be downloaded from <http://www.netlib.org/lapack/> (for netlib LAPACK), <http://sourceforge.net/projects/math-atlas/files/> (for ATLAS), or combined with commercially available compilers such as MKL from Intel.

LAPACK is built in the source directory. First, make a copy of `make.inc.example`:

```
cp make.inc.example make.inc
```

Edit it to set:

```

FORTRAN = gfortran -fimplicit-none -g -fPIC
BLASLIB = /path/to/your/libraries/libblas.a

```

Then:

```

make
cp lapack_LINUX.a $WORKING/fluidity/lib/liblapack.a

```

C.4.4.3 ParMetis

ParMetis (tested for Fluidity with version 3.2.0) is required for mesh partitioning and sparse matrix operations, and can be downloaded from <http://glaros.dtc.umn.edu/gkhome/fsroot/sw/parmetis/OLD>

ParMetis is built in the source directory with:

```

make
cp lib*.a $WORKING/fluidity/lib
cp parmetis.h $WORKING/fluidity/include

```

Note that Fluidity is NOT currently tested with ParMETIS 4.0.0, and further note that ParMETIS is subject to licensing conditions for commercial users. Commercial users should contact the University of Minnesota's Office for Technology Commercialization directly.

Fluidity is in the process of transitioning away from using ParMETIS to avoid this restrictive commercial licensing, and by release 4.2 should have an alternative partitioner which is free to use for commercial partners.

C.4.4.4 PETSc

PETSc (currently tested for Fluidity with version 3.1p8) is required for efficient solver methods within Fluidity. PETSc can be downloaded from <http://www.mcs.anl.gov/petsc/petsc-2/download/index.html> and built in the source directory. First, set `PETSC_DIR` in the source directory:

```
export PETSC_DIR=$PWD
```

Then configure with the following all on one line:

```
./configure --prefix=$WORKING/fluidity --with-mpi-shared=1 --with-shared=1
  --with-debugging=0 --with-parmetis=1 --download-parmetis=1 --with-hypre=1
  --download-hypre=1 --with-prometheus=1 --download-prometheus=1
  --with-fortran-interfaces=1
```

When configure completes, it should supply you with a 'make all' command line, including various configuration variables. Copy and paste this into your terminal and run it. Once it completes, it will supply you with a further 'make install' command line containing other variables; do the same with this, copy and pasting it into your terminal and running it.

Finally, reset the environment variables back to normal:

```
export PETSC_DIR=$WORKING/fluidity
```

NOTE: If you see problems with shared libraries not building correctly, make sure you have built BLAS and LAPACK with `-fPIC`.

C.4.5 VTK and supporting software

VTK is required for Fluidity data output, and currently tested to version 5.6. If you do not already have them installed, you will need Cmake (tested for version 2.8.5, download from <http://www.cmake.org/cmake/resources/software.html>), as well as Tcl and Tk (tested for version 8.5.10, download from <http://www.tcl.tk/software/tcltk/download.html>). All three packages are built with the standard configure, make, and install, Cmake first in its source directory, then Tcl followed by Tk in each package's respective unix/ subdirectory of their main source directory.

VTK (tested version 5.6.1) can be download from <http://www.vtk.org/VTK/resources/software.html>

The following build is for a non-graphical install of Fluidity- ie, one for a cluster, not one for a workstation expected to run diamond. In the situation that diamond is required, `VTK_USE_RENDERING` must be enabled and dependencies on GTK+ satisfied which are provided on the vast majority of modern Linux systems. A description of how to satisfy these dependencies from scratch is beyond the scope of this appendix. When building VTK, it is recommended that shared libraries are enabled, and that `VTKpython` is enabled. The Fluidity configure script should be tolerant of local variations in terms of VTK libraries either being supported internally by VTK or supported through system libraries.

At runtime, the environment variables `VTK_INCLUDE` and `VTK_LIBS` will need to be set to point at your VTK install, and the library directory added to `LD_LIBRARY_PATH`.

For the build described here, these would be set to:

```
export VTK_INCLUDE="$WORKING/fluidity/include/vtk-5.6"
export VTK_LIBS="$WORKING/fluidity/lib/vtk-5.6"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$WORKING/fluidity/lib/vtk-5.6"
```

If you are building from source, VTK should be built in a separate build directory which is not inside the source hierarchy. In the source directory run:

```
mkdir ../VTK-build
cd ../VTK-build
ccmake ../VTK/
```

Then type 'c' and edit the resulting rules screen to:

```
BUILD_EXAMPLES           OFF
BUILD_SHARED_LIBS        ON
BUILD_TESTING            ON
CMAKE_BACKWARDS_COMPATIBILITY  2.4
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX     /path/to/WORKING/fluidity
VTK_DATA_ROOT            VTK_DATA_ROOT-NOTFOUND
VTK_USE_GEOVIS           OFF
VTK_USE_INFOVIS          OFF
VTK_USE_N_WAY_ARRAYS    OFF
VTK_USE_PARALLEL        OFF
VTK_USE_RENDERING       OFF
VTK_USE_VIEWS           OFF
VTK_WRAP_JAVA           OFF
VTK_WRAP_PYTHON        ON
VTK_WRAP_TCL            OFF
```

Then:

- Type 'c' a first time to configure
- Type 'c' a second time to configure
- Type 'g' to generate and quit

Finally, run the standard make and install.

C.4.6 Supporting Libraries

The above libraries should be sufficient for the most basic Fluidity builds, but, depending on local requirements, other external libraries may be required. Brief details and suggestions for avoiding common problems are given here, but package instructions should be referred to for full build details.

C.4.6.1 XML2

XML2 is required for parsing Fluidity's flml parameter file format, and is tested with version 2.7.8. It can be downloaded from <ftp://xmlsoft.org/libxml2/>

C.4.6.2 ARPACK

ARPACK is required for solving large eigenvalue problems, and is tested with version 96 with paths configured for the local site. It can be downloaded from <http://www.caam.rice.edu/software/ARPACK/>

C.4.6.3 NetCDF

NetCDF is required for reading datafiles in NetCDF format, and is tested with version 4.1.3. It is recommended to be configured with *f77*, *f90*, *c*, *cxx*, and utilities enabled, and with `--enable-shared` added to the standard configure.

NetCDF can be downloaded from <http://www.unidata.ucar.edu/downloads/netcdf/>

C.4.6.4 UDUnits

UDUnits is required for physical unit conversions, and is tested with version 2.1.23. Note there is a common issue with hand-building this package where `CPPFLAGS` needs to be correctly set with a `-D` option for the relevant Fortran environment. This commonly leads to an error during the build when not set. See for example <http://www.unidata.ucar.edu/downloads/udunits/>.

Setting

```
export CPPFLAGS=-Df2cFortran
```

should be sufficient for a GCC-based build on Linux.

UDUnits can be downloaded from <http://www.unidata.ucar.edu/downloads/udunits/index.jsp>

C.4.6.5 Trang

Trang is required for parsing Fluidity's flml schema, and is tested with 20030619 but any recent version should be sufficient. Trang can be downloaded from <http://www.thaiopensource.com/relaxng/trang.html>

C.4.6.6 Bazaar (bzt)

Bazaar is recommended as a general tool for accessing the Fluidity code repository, and tested to version 2.4.0. Bazaar can be downloaded from <https://launchpad.net/bzt/+download>

C.4.6.7 CGNS

CGNS is required when any CGNS-format datafiles are being read by Fluidity, and is currently tested to version 2.5.4. It can be downloaded from <http://cgns.sourceforge.net/download.html>.

Appendix D

Troubleshooting

We have several sources of information in case you run into trouble installing or running Fluidity.

Firstly, check if your question is answered in this manual. An overview how to correctly set up Fluidity is given in chapter 1. Questions about configuring Fluidity are covered in chapter 8.

Secondly, have a look at our "Cookbook for Fluidity " (http://amcg.ese.ic.ac.uk/index.php?title=Cook_Book). This webpage provides examples of how to set up particular types of problems in Fluidity.

If none of these documents answered your question, we highly encourage you to get in contact with us by sending us an email to fluidity@imperial.ac.uk.

Appendix E

Mesh formats

This chapter describes the information contained in a mesh file and the two mesh formats that can be read by Fluidity: the gmsh and triangle format. For an overview and further pointers on how to generate these meshes see [chapter 6](#).

E.1 Mesh data

A mesh describes the computational domain in which a simulation takes place. Regardless of the mesh file format in use, the information conveyed is essentially the same.

E.1.1 Node location

The locations of the element vertices are recorded. Usually, these have the same dimension as the topological dimension of the mesh elements. Fluidity does not currently support configurations such as shells in which the node location dimension differs from the element topology dimension.

E.1.2 Element topology

The mesh is composed of elements. In one dimension these will be intervals with each interval joining two nodes. In two dimensions, triangles or quadrilaterals are supported with the elements joining three or four nodes respectively. In three dimensions, the elements can be tetrahedra or hexahedra and will join four or eight nodes.

The element topology will store the indices of the nodes which make up each of the elements in the mesh.

E.1.3 Facets

Facets form the surface of elements. In one dimension, the facets of an element are its bounding nodes. In two dimensions, the facets are the edge intervals while the facets of a three-dimensional tetrahedral element are triangles and those of a hexahedral element are quadrilaterals. External mesh formats tend to only supply facet topology information for facets on the surface of each domain. For each facet specified, the node indices of that facet will be given. These surface facets are used in combination with surface IDs to specify the regions over which boundary conditions should be applied.

E.1.4 Surface IDs

Numbers can be assigned to label particular facets (boundary nodes, edges or faces in 1, 2 or 3 dimensions respectively) in order to set boundary conditions or other parameters. This number can then be used to specify which surface a particular boundary condition should be applied to in Fluidity.

E.1.5 Region IDs

These are analogous to surface IDs, however they are associated with elements rather than facets. Region IDs may be used in Fluidity to specify different initial conditions or material properties to different parts of the domain.

E.2 The triangle format

The *triangle* format is a ASCII file format originally designed for 2D triangle meshes, but it can be easily extended to different dimensions and more complex geometries. Fluidity supports a version of the triangle format which supports 1D, 2D and 3D meshes. The following table shows the supported combinations of element dimension and geometry.

Dimension	Geometry	Number of vertices per element
1D	Line	2
2D	Triangles	3
2D	Quadrilateral	4
3D	Tetrahedra	4
3D	Hexahedra	8

A complete triangle mesh consists of three files: one file defining the nodes of the mesh, one file describing the elements (for example triangles in 2D) and one file defining the boundary parts of the mesh.

The triangle file format is very simple. Since the data is stored in ASCII, any text editor can be used to edit the files. Lines starting with # will be interpreted as a comment by Fluidity. The filename should end with either `.node`, `.ele`, `.bound`, `.edge` or `.face`. The structure of these files will now be explained:

.node file This file holds the coordinates of the nodes. The file structure is:

First line

```
<total number of vertices> <dimension (must be 1,2 or 3)> 0 0
```

Remaining lines

```
<vertex number> <x> [<y> [<z>]]
```

where x, y and z are the coordinates.

Vertices must be numbered consecutively, starting from one.

.ele file Saves the elements of the mesh. The file structure is:

First line:

```
<total number of elements> <nodes per element> 1
```

Remaining lines:

```
<element number> <node> <node> <node> ... <region id>
```

The elements must be numbered consecutively, starting from one. Nodes are indices into the corresponding .node file. For example in case of describing a 2D triangle mesh, the first three nodes are the corner vertices. The region ID can be used by Fluidity to set conditions on different parts of the mesh, see section [E.1.5](#).

.bound file This file is only generated for one-dimensional meshes. It records the boundary points and assigns surface IDs to them. The file structure is:

First line:

```
<total number of boundary points> 1
```

Remaining lines:

```
<boundary point number> <node> <surface id>
```

The boundary points must be numbered consecutively, starting from one. Nodes are indices into the corresponding .node file. The surface ID is used by Fluidity to specify parts of the surface where different boundary conditions will be applied, see section [E.1.4](#).

.edge file This file is only generated for two-dimensional meshes. It records the edges and assigns surface IDs to part of the mesh surface. The file structure is:

First line:

```
<total number of edges> 1
```

Remaining lines:

```
<edge number> <node> <node> ... <surface id>
```

The edges must be numbered consecutively, starting from one. Nodes are indices into the corresponding .node file. The surface ID is used by Fluidity to specify parts of the surface where different boundary conditions will be applied, see section [E.1.4](#).

.face file This file is only generated for three-dimensional meshes. It records the faces and assigns surface IDs to part of the mesh surface. The file structure is:

First line:

```
<total number of faces> 1
```

Remaining lines:

```
<face number> <node> <node> <node> ... <surface id>
```

The faces must be numbered consecutively, starting from one. Nodes are indices into the corresponding .node file. The surface ID is used by Fluidity to specify parts of the surface where different boundary conditions will be applied, see section [E.1.4](#).

To clarify the file format, a simple 1D example is shown: The following .node file defines 6 equidistant nodes from 0.0 to 5.0

```
# example.node
6 1 0 0
1 0.0
2 1.0
3 2.0
4 3.0
5 4.0
6 5.0
```

The `.ele` file connects these nodes to 5 lines. Two regions will be defined with the IDs 1 and 2.

```
# example.ele
5 2 1
1 1 2 1
2 2 3 1
3 3 4 1
4 4 5 2
5 5 6 2
```

Finally, the `.bound` file tags the very left and very right nodes as boundary points and assigns the surface IDs 1 and 2 to them.

```
# example.bound
2 1
1 1 1
2 6 2
```

E.3 The Gmsh format

Fluidity contains support for the Gmsh format. Gmsh is a mesh generator freely available on the Web at (<http://geuz.org/gmsh/>), and is included in Linux distributions such as Ubuntu.

Unlike triangle files, Gmsh meshes are contained within one file, which have the extension `.msh`. The file contents may be either binary or ASCII.

This section briefly describes the Gmsh format, and is only intended to serve as a short introduction. If you need further information, please read the official Gmsh documentation (<http://geuz.org/gmsh/doc/texinfo/gmsh.pdf>). Typically Gmsh files used in Fluidity contain three parts: a header, a section for nodes, and one for elements. These are explained in more detail below.

The header

This contains Gmsh file version information, and indicates whether the main data is in ASCII or binary format. This will typically look like:

```
$MeshFormat
2.1 0 8
[Extra data here in binary mode]
$EndMeshFormat
```

From the listing above we can tell that:

- the Gmsh format version is 2.1
- it is in ASCII, as indicated by the 0 (1=binary)
- the byte size of double precision is 8

In addition, in binary mode the integer 1 is written as 4 raw bytes, to check that the endianness of the Gmsh file and the system are the same (*you will rarely have to worry about this*)

The nodes section

The next section contains node data, viz:

```
$Nodes
number_of_nodes
[node data]
$EndNodes
```

The `[node data]` part contains the listing of nodes, with ID, followed by x , y , and z coordinates. This part will be in binary when binary mode has been selected. Note that even with 2D problems, there will be a zeroed z coordinate.

The elements section

The elements section contains information on both facets and regular elements. It also varies between binary and ASCII formats. The ASCII version is:

```
$Elements
element1_id element_type number_of_tags tag_list node_number_list
element2_id ...
...
...
$EndElements
```

Tags are integer properties ascribed to the element. In Fluidity, usually we are only concerned with the first one, the physical ID. This can mean one of two things:

- A region ID - in the case of elements
- A surface ID - in the case of facets

Since Gmsh doesn't explicitly label facets or regular elements as such, internally Fluidity works this out from type: eg, if there a mesh consists of tetrahedra and triangles, then triangles must be the facets.

Internal use

Fluidity occasionally augments GMSH files for its own internal use. It does this in two cases. Firstly, when generating periodic meshes through `periodise` (section 6.6.5), the fourth element tag is reserved to label elements at the periodic boundary. Secondly, when checkpointing extruded meshes, a new section called `$NodeData` is created at the end of the GMSH file; this contains essential node column ID information.

Index

- absorption term 14, 126
- adaptivity options 148, 150
- advection-diffusion equation 13
 - continuous Galerkin 32
 - control volume 43
 - discontinuous Galerkin 36
 - discretisation options 123
 - weak form 30
 - discretised 31
- algebraic multigrid (AMG) 66
- ARPACK 270
- basis function 31
- Bassi-Rebay 43
- bazaar 2
- biology 139
- BLAS 267
- body forces
 - astronomical tides 25
- boundary conditions 134
 - boundary tides 24
 - bulk parameterisations 24
 - Dirichlet 14, 19, 135
 - strongly imposed 36
 - weakly imposed 35
 - drag 136
 - flux 136
 - free stress 19
 - free surface 136
 - generic length scale model 139
 - k- ε model 139
 - Momentum 19
 - NEMO data 137
 - Neumann 15, 135
 - weakly imposed 35
 - no normal flow 137
 - ocean 110, 138
 - prescribed stress 19
 - scalar 14
 - setting 135
 - synthetic eddy method 138
 - traction 19
 - wetting and drying 136
 - wind stress 136
 - zero flux 135
- Boussinesq
 - approximation 23
 - discretised 57
 - equations 23
 - buoyancy 22
 - bzr *see* bazaar
 - installing 271
 - CGNS 271
 - checkpointing 112
 - restarting 113
 - Compact Discontinuous Galerkin 43
 - compilers 265
 - compressible fluid model 155
 - conjugate gradient 128
 - conjugate gradient method 65
 - conservation
 - equation 15
 - mass 15
 - momentum 15
 - continuity equation 23
 - continuous Galerkin
 - see* Galerkin
 - continuous 39
 - control volume
 - advection 43
 - convergence criteria 66
 - Coriolis 20
 - β -plane 22
 - f -plane 22
 - options 116
 - density 22
 - reference 23
 - detectors
 - Lagrangian 67
 - options 113
 - Diamond 10
 - diffusion
 - discontinuous Galerkin 40
 - Bassi-Rebay 43
 - CDG 43
 - diffusivity
 - eddy 69, 72
 - dimension 109
 - discontinuous Galerkin
 - see* Galerkin

- discontinuous 39
- Entering bathymetry data into fluidity using Gmsh 94
- equation of state 17, 18
 - linear 18, 130
 - options 130
 - Pade approximation 18, 130
 - stiffened gas 131
- errors
 - linear solver 129
- field 120
 - constant 121
 - from Nemo 123
 - input 122
 - Python function 122, 259
 - values 121
- free surface 19
 - boundary condition 136
 - weak form 63
 - Wetting and drying 20
- Galerkin
 - continuous 30
 - advection 32
 - discontinuous 30
 - advection 36
 - diffusion 40
 - slope limiters 39
 - methods 30
 - Petrov- 33
 - projection 30
- Gauss-Seidel iteration 64
- generic length scale model 69
 - boundary conditions 139
- GLS *see* generic length scale model
- GMRES 65, 128
- gmsh 93
- gravity 116
- grid *see* mesh, *see* mesh
- initial conditions
 - setting 121
- Jacobi iteration 64
- $k-\varepsilon$ model 72, 132
 - boundary conditions 139
- Krylov subspace methods 65
- Lagrangian trajectories 67
- LAPACK 268
- large eddy simulation 74
- libraries
 - installing externals from source 264
 - installing on debian and ubuntu 264
- linear momentum 22
- linear solvers 63, 64
 - convergence criteria 66, 128
 - iterative 64
 - Krylov subspace methods 65
 - options 128
 - preconditioners 65, 128
- mass lumping 59
- mesh
 - coordinate mesh 90
 - cubical 90
 - derived 118
 - extruded 90
 - file formats 276
 - generation 89, 275
 - input 117
 - meshing tools 91
 - fldecomp 92
 - firedcomp 92
 - gmsh 93
 - mesh conversion 91
 - mesh creation 91
 - mesh verification 91
 - periodise 93
 - Terreno 93
 - nodes 90
 - output 111
 - periodic 91, 118
 - simplicial 90
- momentum equation 15, 19, 23
 - discretisation options 123
 - discretised 56
- multi-material flow 25
- multi-phase flow 26
- multigrid 66, 128
- multigrid methods 66
- NetCDF 270
- numpy 267
- OpenMPI 266
 - options
 - names 108
 - syntax 107
- P_1 31
- Péclet number
 - grid 32
- parallel 10
 - mesh decomposition 92
- ParMetis 268
- penalty parameter
 - CDG 125

interior penalty method	125	advection subcycling	55
periodic domain	91, 118	step	114
Petrov-Galerkin	33	θ -scheme	55
PETSc	128, 268	traction force	19
P_N	30	traditional approximation	21
preconditioners	65	Trang	271
pressure		trial function	30
discretisation	57	turbulence model	69, 72, 74, 132
pressure		UDUnits	271
balance	61	viscosity	
CG with CV tested continuity	61, 127	eddy	69, 72
correction	58	visualisation	157
geostrophic balance	127	vtk	111
hydrostatic	22	installing	269
null space	127	vtu	157
options	127	weak form	30
perturbation	23	wind forcing	136
Python		XML2	270
detector positions	113	zoltan	105, 150
installing	266		
prescribed field values	122		
state interface	259		
quadrature			
options	110		
reaction term	14		
region ID	89, 123, 276		
Reynolds stress	69, 72		
Reynolds Transport theorem	15		
sediments	140		
slope limiters	39		
Sobolev space	30		
solvers	<i>see</i> linear solvers		
source term	14, 126		
spherical earth	110		
sponge regions	126		
stabilisation			
advection	32		
balancing diffusion	32		
discontinuous Galerkin	39		
Petrov-Galerkin	33		
stat file	113, 188		
strain	17		
stress	17		
subversion	2, 4		
surface ID	89, 135, 276, 277		
symmetric positive definite (SPD)	65		
Terreno	93		
test function	30		
θ -scheme	55		
tides	139		
time			