COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

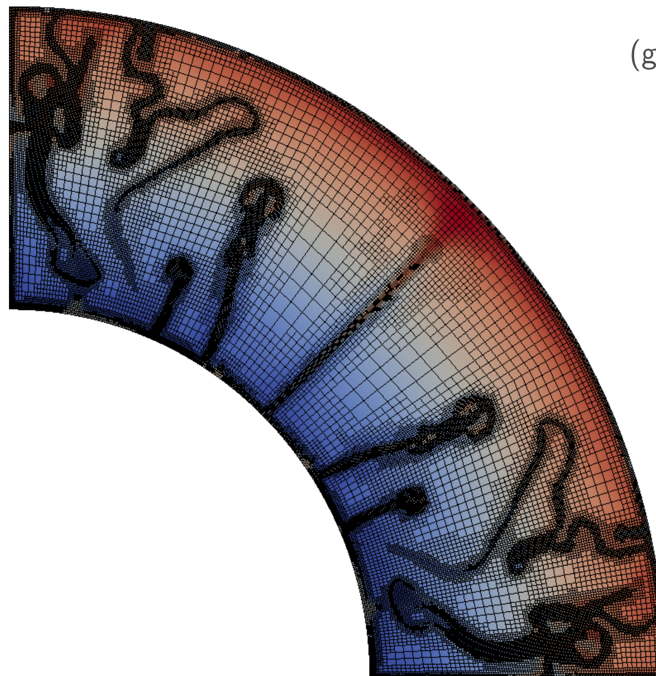
ASPECT

Advanced Solver for Problems in Earth's ConvecTion

User Manual

Version 1.4.0

(generated May 13, 2016)



Wolfgang Bangerth

Timo Heister

with contributions by:

Jacqueline Austermann, Markus Bürg, Juliane Dannberg, William Durkin, Grant Euen, René Gaßmöller, Thomas Geenen, Anne Glerum, Ryan Grove, Eric Heien, Scott King, Martin Kronbichler, Shangxin Liu, Elvira Mulyukova, Jonathan Perry-Houts, Tahiry Rajaonarison, Ian Rose, D. Sarah Stamps, Cedric Thieulot, Iris van Zelst, Siqi Zhang

geodynamics.org

Contents

1 Introduction **6**
1.1 Referencing ASPECT . . . 7
1.2 Acknowledgments . . . 7

2 Equations, models, coefficients **8**
2.1 Basic equations . . . 8
2.1.1 A comment on adiabatic heating . . . 9
2.1.2 Boundary conditions . . . 9
2.1.3 Comments on the final set of equations . . . 10
2.2 Coefficients . . . 10
2.3 Dimensional or non-dimensionalized equations? . . . 12
2.4 Static or dynamic pressure? . . . 14
2.5 Pressure normalization . . . 15
2.6 Initial conditions and the adiabatic pressure/temperature . . . 15
2.7 Compositional fields . . . 16
2.8 Constitutive laws . . . 17
2.9 Numerical methods . . . 18
2.10 Simplifications of the basic equations . . . 19
2.10.1 The Boussinesq approximation: Incompressibility . . . 19
2.10.2 Almost linear models . . . 20
2.10.3 Compressible models . . . 21
2.11 Free surface calculations . . . 21
2.11.1 Arbitrary Lagrangian-Eulerian implementation . . . 21
2.11.2 Free surface stabilization . . . 22
2.12 Nullspace removal . . . 22

3 Installation **23**
3.1 System prerequisites . . . 23
3.2 Software prerequisites . . . 24
3.3 Obtaining ASPECT and initial configuration . . . 25
3.4 Compiling ASPECT and generating documentation . . . 25
3.5 Compiling a static ASPECT executable . . . 26
3.6 Installing and running ASPECT on Mac OS X . . . 26

4 Running ASPECT **27**
4.1 Overview . . . 27
4.2 Selecting between 2d and 3d runs . . . 31
4.3 Debug or optimized mode . . . 32
4.4 Visualizing results . . . 33
4.4.1 Visualization the graphical output using *Visit* . . . 34
4.4.2 Visualizing statistical data . . . 36
4.4.3 Large data issues for parallel computations . . . 39
4.5 Checkpoint/restart support . . . 39
4.6 Making ASPECT run faster . . . 40
4.6.1 Debug vs. optimized mode . . . 40
4.6.2 Adjusting solver tolerances . . . 40
4.6.3 Adjusting solver preconditioner tolerances . . . 41
4.6.4 Using lower order elements for the temperature/compositional discretization . . . 41
4.6.5 Limiting postprocessing . . . 41
4.6.6 Switching off pressure normalization . . . 42

4.6.7 Regularizing models with large coefficient variation 42

5 Run-time input parameters **42**
5.1 Overview 42
5.1.1 The structure of parameter files 42
5.1.2 Categories of parameters 43
5.1.3 A note on the syntax of formulas in input files 44
5.2 Global parameters 45
5.3 Parameters in section `Adiabatic conditions model` 50
5.4 Parameters in section `Boundary composition model` 50
5.5 Parameters in section `Boundary composition model/Ascii data model` 51
5.6 Parameters in section `Boundary composition model/Box` 52
5.7 Parameters in section `Boundary composition model/Box with lithosphere boundary indicators` 53
5.8 Parameters in section `Boundary composition model/Initial composition` 54
5.9 Parameters in section `Boundary composition model/Spherical constant` 55
5.10 Parameters in section `Boundary temperature model` 55
5.11 Parameters in section `Boundary temperature model/Ascii data model` 56
5.12 Parameters in section `Boundary temperature model/Box` 57
5.13 Parameters in section `Boundary temperature model/Box with lithosphere boundary indicators` 58
5.14 Parameters in section `Boundary temperature model/Constant` 59
5.15 Parameters in section `Boundary temperature model/Function` 59
5.16 Parameters in section `Boundary temperature model/Initial temperature` 60
5.17 Parameters in section `Boundary temperature model/Spherical constant` 60
5.18 Parameters in section `Boundary traction model` 61
5.19 Parameters in section `Boundary traction model/Function` 61
5.20 Parameters in section `Boundary velocity model` 62
5.21 Parameters in section `Boundary velocity model/Ascii data model` 62
5.22 Parameters in section `Boundary velocity model/Function` 63
5.23 Parameters in section `Boundary velocity model/GPlates model` 64
5.24 Parameters in section `Checkpointing` 66
5.25 Parameters in section `Compositional fields` 66
5.26 Parameters in section `Compositional initial conditions` 67
5.27 Parameters in section `Compositional initial conditions/Ascii data model` 67
5.28 Parameters in section `Compositional initial conditions/Function` 68
5.29 Parameters in section `Discretization` 69
5.30 Parameters in section `Discretization/Stabilization parameters` 70
5.31 Parameters in section `Free surface` 71
5.32 Parameters in section `Geometry model` 72
5.33 Parameters in section `Geometry model/Box` 73
5.34 Parameters in section `Geometry model/Box with lithosphere boundary indicators` . . . 75
5.35 Parameters in section `Geometry model/Chunk` 77
5.36 Parameters in section `Geometry model/Ellipsoidal chunk` 78
5.37 Parameters in section `Geometry model/Sphere` 80
5.38 Parameters in section `Geometry model/Spherical shell` 80
5.39 Parameters in section `Gravity model` 81
5.40 Parameters in section `Gravity model/Function` 82
5.41 Parameters in section `Gravity model/Radial constant` 82
5.42 Parameters in section `Gravity model/Radial linear` 83
5.43 Parameters in section `Gravity model/Vertical` 83
5.44 Parameters in section `Heating model` 83
5.45 Parameters in section `Heating model/Adiabatic heating` 84
5.46 Parameters in section `Heating model/Constant heating` 84

5.47 Parameters in section Heating model/Function . . . 85
5.48 Parameters in section Heating model/Latent heat . . . 85
5.49 Parameters in section Heating model/Radioactive decay . . . 85
5.50 Parameters in section Heating model/Shear heating . . . 87
5.51 Parameters in section Initial conditions . . . 87
5.52 Parameters in section Initial conditions/Adiabatic . . . 88
5.53 Parameters in section Initial conditions/Adiabatic/Function . . . 90
5.54 Parameters in section Initial conditions/Adiabatic boundary . . . 90
5.55 Parameters in section Initial conditions/Ascii data model . . . 91
5.56 Parameters in section Initial conditions/Function . . . 92
5.57 Parameters in section Initial conditions/Harmonic perturbation . . . 93
5.58 Parameters in section Initial conditions/Inclusion shape perturbation . . . 93
5.59 Parameters in section Initial conditions/S40RTS perturbation . . . 95
5.60 Parameters in section Initial conditions/SAVANI perturbation . . . 96
5.61 Parameters in section Initial conditions/Solidus . . . 98
5.62 Parameters in section Initial conditions/Solidus/Data . . . 98
5.63 Parameters in section Initial conditions/Solidus/Perturbation . . . 99
5.64 Parameters in section Initial conditions/Spherical gaussian perturbation . . . 99
5.65 Parameters in section Initial conditions/Spherical hexagonal perturbation . . . 100
5.66 Parameters in section Material model . . . 101
5.67 Parameters in section Material model/Averaging . . . 106
5.68 Parameters in section Material model/Composition reaction model . . . 106
5.69 Parameters in section Material model/Depth dependent model . . . 108
5.70 Parameters in section Material model/Depth dependent model/Viscosity depth function 109
5.71 Parameters in section Material model/Diffusion dislocation . . . 110
5.72 Parameters in section Material model/Drucker Prager . . . 114
5.73 Parameters in section Material model/Drucker Prager/Viscosity . . . 115
5.74 Parameters in section Material model/Latent heat . . . 115
5.75 Parameters in section Material model/Latent heat melt . . . 119
5.76 Parameters in section Material model/Morency and Doin . . . 124
5.77 Parameters in section Material model/Multicomponent . . . 126
5.78 Parameters in section Material model/Simple compressible model . . . 127
5.79 Parameters in section Material model/Simple model . . . 128
5.80 Parameters in section Material model/Simpler model . . . 130
5.81 Parameters in section Material model/Steinberger model . . . 131
5.82 Parameters in section Mesh refinement . . . 133
5.83 Parameters in section Mesh refinement/Boundary . . . 137
5.84 Parameters in section Mesh refinement/Composition . . . 137
5.85 Parameters in section Mesh refinement/Maximum refinement function . . . 138
5.86 Parameters in section Mesh refinement/Minimum refinement function . . . 139
5.87 Parameters in section Model settings . . . 140
5.88 Parameters in section Postprocess . . . 143
5.89 Parameters in section Postprocess/Command . . . 146
5.90 Parameters in section Postprocess/Depth average . . . 146
5.91 Parameters in section Postprocess/Dynamic Topography . . . 147
5.92 Parameters in section Postprocess/Point values . . . 147
5.93 Parameters in section Postprocess/Tracers . . . 147
5.94 Parameters in section Postprocess/Tracers/Function . . . 151
5.95 Parameters in section Postprocess/Tracers/Generator . . . 152
5.96 Parameters in section Postprocess/Tracers/Generator/Ascii file . . . 152
5.97 Parameters in section Postprocess/Tracers/Generator/Probability density function . 152

5.98 Parameters in section `Postprocess/Tracers/Generator/Uniform box` . . . 153
5.99 Parameters in section `Postprocess/Tracers/Generator/Uniform radial` . . . 154
5.100 Parameters in section `Postprocess/Visualization` . . . 155
5.101 Parameters in section `Postprocess/Visualization/Compositional fields as vectors` . . 159
5.102 Parameters in section `Postprocess/Visualization/Dynamic Topography` . . . 160
5.103 Parameters in section `Postprocess/Visualization/Material properties` . . . 160
5.104 Parameters in section `Postprocess/Visualization/Melt fraction` . . . 160
5.105 Parameters in section `Prescribed Stokes solution` . . . 163
5.106 Parameters in section `Prescribed Stokes solution/Ascii data model` . . . 164
5.107 Parameters in section `Prescribed Stokes solution/Pressure function` . . . 164
5.108 Parameters in section `Prescribed Stokes solution/Velocity function` . . . 165
5.109 Parameters in section `Termination criteria` . . . 166
5.110 Parameters in section `Termination criteria/Steady state velocity` . . . 167
5.111 Parameters in section `Termination criteria/User request` . . . 167

6 Cookbooks **167**
6.1 How to set up computations . . . 168
6.2 Simple setups . . . 169
6.2.1 Convection in a 2d box . . . 169
6.2.2 Convection in a 3d box . . . 181
6.2.3 Convection in a box with prescribed, variable velocity boundary conditions . . . 186
6.2.4 Using passive and active compositional fields . . . 189
6.2.5 Using tracer particles . . . 195
6.2.6 Using a free surface . . . 197
6.2.7 Using a free surface in a model with a crust . . . 199
6.2.8 Averaging material properties . . . 200
6.2.9 Prescribed internal velocity constraints . . . 206
6.2.10 Artificial viscosity smoothing . . . 211
6.2.11 Tracking finite strain . . . 212
6.3 Geophysical setups . . . 214
6.3.1 Simple convection in a quarter of a 2d annulus . . . 216
6.3.2 Simple convection in a spherical 3d shell . . . 222
6.3.3 3D convection with an Earth-like initial condition . . . 226
6.3.4 Using reconstructed surface velocities by GPlates . . . 228
6.3.5 Reproducing rheology of Morency and Doin, 2004 . . . 231
6.3.6 Crustal deformation . . . 235
6.4 Benchmarks . . . 240
6.4.1 Running benchmarks that require code . . . 240
6.4.2 The van Keken thermochemical composition benchmark . . . 242
6.4.3 The SolCx Stokes benchmark . . . 247
6.4.4 The SolKz Stokes benchmark . . . 253
6.4.5 The “inclusion” Stokes benchmark . . . 255
6.4.6 The Burstedde variable viscosity benchmark . . . 257
6.4.7 The “Stokes’ law” benchmark . . . 259
6.4.8 Latent heat benchmark . . . 264
6.4.9 The 2D cylindrical shell benchmarks by Davies et al. . . . 269
6.4.10 The Crameri et al. benchmarks . . . 274

7 Extending ASPECT **279**
7.1 The idea of plugins and the `SimulatorAccess` and `Introspection` classes . . . 280
7.2 How to write a plugin . . . 284
7.3 Materials, geometries, gravitation and other plugin types . . . 285
7.3.1 Material models . . . 285
7.3.2 Heating models . . . 287
7.3.3 Geometry models . . . 287
7.3.4 Gravity models . . . 291
7.3.5 Initial conditions . . . 291
7.3.6 Prescribed velocity boundary conditions . . . 292
7.3.7 Temperature boundary conditions . . . 293
7.3.8 Postprocessors: Evaluating the solution after each time step . . . 294
7.3.9 Visualization postprocessors . . . 297
7.3.10 Mesh refinement criteria . . . 299
7.3.11 Criteria for terminating a simulation . . . 299
7.4 Extending ASPECT through the signals mechanism . . . 300
7.5 Extending the basic solver . . . 303

8 Future plans for ASPECT **304**

9 Finding answers to more questions **305**

References **306**

Index of run-time parameter entries **309**

Index of run-time parameters with section names **314**

1 Introduction

ASPECT — short for Advanced Solver for Problems in Earth's ConvecTion — is a code intended to solve the equations that describe thermally driven convection with a focus on doing so in the context of convection in the earth mantle. It is primarily developed by computational scientists at Texas A&M University based on the following principles:

- *Usability and extensibility:* Simulating mantle convection is a difficult problem characterized not only by complicated and nonlinear material models but, more generally, by a lack of understanding which parts of a much more complicated model are really necessary to simulate the defining features of the problem. To name just a few examples:

  - Mantle convection is often solved in a spherical shell geometry, but the earth is not a sphere – its true shape on the longest length scales is dominated by polar oblateness, but deviations from spherical shape relevant to convection patterns may go down to the length scales of mountain belts, mid-ocean ridges or subduction trenches. Furthermore, processes outside the mantle like crustal depression during glaciations can change the geometry as well.
  - Rocks in the mantle flow on long time scales, but on shorter time scales they behave more like a visco-elasto-plastic material as they break and as their crystalline structure heals again. The mathematical models discussed in Section 2 can therefore only be approximations.
  - If pressures are low and temperatures high enough, rocks melt, leading to all sorts of new and interesting behavior.

  This uncertainty in what problem one actually wants to solve requires a code that is easy to extend by users to support the community in determining what the essential features of convection in the earth mantle are. Achieving this goal also opens up possibilities outside the original scope, such as the simulation of convection in exoplanets or the icy satellites of the gas giant planets in our solar system.

- *Modern numerical methods:* We build ASPECT on numerical methods that are at the forefront of research in all areas – adaptive mesh refinement, linear and nonlinear solvers, stabilization of transport-dominated processes. This implies complexity in our algorithms, but also guarantees highly accurate solutions while remaining efficient in the number of unknowns and with CPU and memory resources.

- *Parallelism:* Many convection processes of interest are characterized by small features in large domains – for example, mantle plumes of a few tens of kilometers diameter in a mantle almost 3,000 km deep. Such problems can not be solved on a single computer but require dozens or hundreds of processors to work together. ASPECT is designed from the start to support this level of parallelism.

- *Building on others' work:* Building a code that satisfies above criteria from scratch would likely require several 100,000 lines of code. This is outside what any one group can achieve on academic time scales. Fortunately, most of the functionality we need is already available in the form of widely used, actively maintained, and well tested and documented libraries, and we leverage these to make ASPECT a much smaller and easier to understand system. Specifically, ASPECT builds immediately on top of the DEAL.II library (see https://www.dealii.org/) for everything that has to do with finite elements, geometries, meshes, etc.; and, through DEAL.II on Trilinos (see http://trilinos.org/) for parallel linear algebra and on P4EST (see http://www.p4est.org/) for parallel mesh handling.

- *Community:* We believe that a large project like ASPECT can only be successful as a community project. Every contribution is welcome and we want to help you so we can improve ASPECT together.

Combining all of these aspects into one code makes for an interesting challenge. We hope to have achieved our goal of providing a useful tool to the geodynamics community and beyond!

Note: ASPECT is a community project. As such, we encourage contributions from the community to improve this code over time. Natural candidates for such contributions are implementations of new plugins as discussed in Section 7.3 since they are typically self-contained and do not require much knowledge of the details of the remaining code. Obviously, however, we also encourage contributions to the core functionality in any form! If you have something that might be of general interest, please contact us.

Note: ASPECT will only solve problems relevant to the community if we get feedback from the community on things that are missing or necessary for what you want to do. Let us know by personal email to the developers, or the mantle convection or `aspect-devel` mailing lists hosted at http://lists.geodynamics.org/cgi-bin/mailman/listinfo/aspect-devel!

1.1 Referencing ASPECT

As with all scientific work, funding agencies have a reasonable expectation that if we ask for continued funding for this work, we need to demonstrate relevance. To this end, we ask that if you publish results that were obtained to some part using ASPECT, you cite the following, canonical reference for this software:

```
@Article{KHB12,
  author =       {M. Kronbichler and T. Heister and W. Bangerth},
  title =        {High Accuracy Mantle Convection Simulation through Modern Numerical Methods},
  journal =      {Geophysics Journal International},
  year =         2012,
  volume =       191,
  pages =        {12--29}}
```

You can refer to the website by citing the following:

```
      @MANUAL{aspectweb,
        title  = {ASPECT: Advanced Solver for Problems in Earth's ConvecTion},
        author = {W. Bangerth and T. Heister and others},
        year   = {2014},
        note   = {\texttt{https://aspect.dealii.org/}},
        url    = {https://aspect.dealii.org/}
      }
```

The manual's proper reference is this:

```
@Manual{aspectmanual,
        title =        {\textsc{ASPECT}: Advanced Solver for Problems in Earth's
                        ConvecTion},
        author =       {W. Bangerth and T. Heister and others},
        organization = {Computational Infrastructure for Geodynamics},
        year =         2014
      }
```

1.2 Acknowledgments

The development of ASPECT has been funded through a variety of grants to the authors. Most immediately, it has been supported through the Computational Infrastructure in Geodynamics (CIG-II) grant (National Science Foundation Award No. EAR-0949446, via The University of California – Davis) but the initial portions have also been supported by the original CIG grant (National Science Foundation Award No.

EAR-0426271, via The California Institute of Technology). In addition, the libraries upon which ASPECT builds heavily have been supported through many other grants that are equally gratefully acknowledged.

Please acknowledge CIG as follows:

ASPECT is hosted by the Computational Infrastructure for Geodynamics (CIG) which is supported by the National Science Foundation award NSF-094946.

2 Equations, models, coefficients

2.1 Basic equations

ASPECT solves a system of equations in a $d = 2$- or $d = 3$-dimensional domain Ω that describes the motion of a highly viscous fluid driven by differences in the gravitational force due to a density that depends on the temperature. In the following, we largely follow the exposition of this material in Schubert, Turcotte and Olson [STO01].

Specifically, we consider the following set of equations for velocity $\mathbf{u}$, pressure p and temperature T, as well as a set of advected quantities c_i that we call *compositional fields*:

$$-\nabla \cdot \left[2\eta \left(\varepsilon(\mathbf{u}) - \frac{1}{3}(\nabla \cdot \mathbf{u})\mathbf{1}\right)\right] + \nabla p = \rho \mathbf{g} \qquad \text{in } \Omega, \tag{1}$$

$$\nabla \cdot (\rho \mathbf{u}) = 0 \qquad \text{in } \Omega, \tag{2}$$

$$\begin{aligned} \rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T\right) - \nabla \cdot k \nabla T = \rho H & \\ + 2\eta \left(\varepsilon(\mathbf{u}) - \frac{1}{3}(\nabla \cdot \mathbf{u})\mathbf{1}\right) : \left(\varepsilon(\mathbf{u}) - \frac{1}{3}(\nabla \cdot \mathbf{u})\mathbf{1}\right) & \\ + \alpha T \left(\mathbf{u} \cdot \nabla p\right) & \\ + \rho T \Delta S \left(\frac{\partial X}{\partial t} + \mathbf{u} \cdot \nabla X\right) & \qquad \text{in } \Omega, \end{aligned} \tag{3}$$

$$\frac{\partial c_i}{\partial t} + \mathbf{u} \cdot \nabla c_i = q_i \qquad \text{in } \Omega, i = 1 \dots C \tag{4}$$

where $\varepsilon(\mathbf{u}) = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$ is the symmetric gradient of the velocity (often called the *strain rate*).

In this set of equations, (1) and (2) represent the compressible Stokes equations in which $\mathbf{u} = \mathbf{u}(\mathbf{x}, t)$ is the velocity field and $p = p(\mathbf{x}, t)$ the pressure field. Both fields depend on space $\mathbf{x}$ and time t. Fluid flow is driven by the gravity force that acts on the fluid and that is proportional to both the density of the fluid and the strength of the gravitational pull.

Coupled to this Stokes system is equation (3) for the temperature field $T = T(\mathbf{x}, t)$ that contains heat conduction terms as well as advection with the flow velocity $\mathbf{u}$. The right hand side terms of this equation correspond to

- internal heat production for example due to radioactive decay;
- friction heating;
- adiabatic compression of material;
- phase change.

The last term of the temperature equation corresponds to the latent heat generated or consumed in the process of phase change of material. The latent heat release is proportional to changes in the fraction of

material X that has already undergone the phase transition (also called phase function) and the change of entropy ΔS. This process applies both to solid-state phase transitions and to melting/solidification. Here, ΔS is positive for exothermic phase transitions. As the phase of the material, for a given composition, depends on the temperature and pressure, the latent heat term can be reformulated:

$$\frac{\partial X}{\partial t} + \mathbf{u} \cdot \nabla X = \frac{DX}{Dt} = \frac{\partial X}{\partial T}\frac{DT}{Dt} + \frac{\partial X}{\partial p}\frac{Dp}{Dt} = \frac{\partial X}{\partial T}\left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T\right) + \frac{\partial X}{\partial p}\mathbf{u} \cdot \nabla p.$$

The last transformation results from the assumption that the flow field is always in equilibrium and consequently $\partial p/\partial t = 0$ (this is the same assumption that underlies the fact that equation (1) does not have a term $\partial \mathbf{u}/\partial t$). With this reformulation, we can rewrite (3) in the following way in which it is in fact implemented:

$$\begin{aligned}\left(\rho C_p - \rho T \Delta S \frac{\partial X}{\partial T}\right)\left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T\right) - \nabla \cdot k \nabla T = \rho H & \\ + 2\eta \left(\varepsilon(\mathbf{u}) - \frac{1}{3}(\nabla \cdot \mathbf{u})\mathbf{1}\right) : \left(\varepsilon(\mathbf{u}) - \frac{1}{3}(\nabla \cdot \mathbf{u})\mathbf{1}\right) & \\ + \alpha T \left(\mathbf{u} \cdot \nabla p\right) & \\ + \rho T \Delta S \frac{\partial X}{\partial p}\mathbf{u} \cdot \nabla p & \quad \text{in } \Omega.\end{aligned} \tag{5}$$

The last of the equations above, equation (4), describes the evolution of additional fields that are transported along with the velocity field $\mathbf{u}$ and may react with each other and react to other features of the solution, but that do not diffuse. We call these fields c_i *compositional fields*, although they can also be used for other purposes than just tracking chemical compositions. We will discuss this equation in more detail in Section 2.7.

2.1.1 A comment on adiabatic heating

Other codes and texts sometimes make a simplification to the adiabatic heating term in the previous equation. If you assume the vertical component of the gradient of the *dynamic* pressure to be small compared to the gradient of the *total* pressure (in other words, the gradient is dominated by the gradient of the hydrostatic pressure), then $-\rho\mathbf{g} \approx \nabla \mathbf{p}$, and we have the following relation (the negative sign is due to $\mathbf{g}$ pointing downwards)

$$\alpha T \left(\mathbf{u} \cdot \nabla \mathbf{p}\right) \approx -\alpha \rho T \mathbf{u} \cdot \mathbf{g}.$$

While this simplification is possible, it is not necessary if you have access to the total pressure. ASPECT therefore implements the original term without this simplification.

2.1.2 Boundary conditions

Having discussed (3), let us come to the last one of the original set of equations, (4). It describes the motion of a set of advected quantities $c_i(\mathbf{x}, t), i = 1 \ldots C$. We call these *compositional fields* because we think of them as spatially and temporally varying concentrations of different elements, minerals, or other constituents of the composition of the material that convects. As such, these fields participate actively in determining the values of the various coefficients of these equations. On the other hand, ASPECT also allows the definition of material models that are independent of these compositional fields, making them passively advected quantities. Several of the cookbooks in Section 6 consider compositional fields in this way, i.e., essentially as tracer quantities that only keep track of where material came from.

These equations are augmented by boundary conditions that can either be of Dirichlet-, Neumann, or

tangential type on subsets of the boundary $\Gamma = \partial\Omega$:

$$
\begin{aligned}
\mathbf{u} &= 0 && \text{on } \Gamma_{0,\mathbf{u}}, && (6)\\
\mathbf{u} &= \mathbf{u}_{\text{prescribed}} && \text{on } \Gamma_{\text{prescribed},\mathbf{u}}, && (7)\\
\mathbf{n}\cdot\mathbf{u} &= 0 && \text{on } \Gamma_{\parallel,\mathbf{u}}, && (8)\\
(2\eta\varepsilon(\mathbf{u}) - pI)\mathbf{n} &= \mathbf{t} && \text{on } \Gamma_{\text{traction},\mathbf{u}}, && (9)\\
T &= T_{\text{prescribed}} && \text{on } \Gamma_{D,T}, && (10)\\
\mathbf{n}\cdot k\nabla T &= 0 && \text{on } \Gamma_{N,T}. && (11)\\
c_i &= 0 && \text{on } \Gamma_{in} = \{\mathbf{x} : \mathbf{u}\cdot\mathbf{n} < 0\}. && (12)
\end{aligned}
$$

Here, the boundary conditions for velocity and temperature are subdivided into disjoint parts:

- $\Gamma_{0,\mathbf{u}}$ corresponds to parts of the boundary on which the velocity is fixed to be zero.
- $\Gamma_{\text{prescribed},\mathbf{u}}$ corresponds to parts of the boundary on which the velocity is prescribed to some value (which could also be zero). It is possible to restrict prescribing the velocity to only certain components of the velocity vector.
- $\Gamma_{\parallel,\mathbf{u}}$ corresponds to parts of the boundary on which the velocity may be nonzero but must be parallel to the boundary, with the tangential component undetermined.
- $\Gamma_{\text{traction},\mathbf{u}}$ corresponds to parts of the boundary on which the traction is prescribed to some surface force density (a common application being $\mathbf{t} = -p\mathbf{n}$ if one just wants to prescribe a pressure component). It is possible to restrict prescribing the traction to only certain vector components.
- $\Gamma_{D,T}$ corresponds to places where the temperature is prescribed (for example at the inner and outer boundaries of the earth mantle).
- $\Gamma_{N,T}$ corresponds to places where the temperature is unknown but the heat flux across the boundary is zero (for example on symmetry surfaces if only a part of the shell that constitutes the domain the Earth mantle occupies is simulated).

We require that one of these boundary conditions hold at each point for both velocity and temperature, i.e., $\Gamma_{0,\mathbf{u}} \cup \Gamma_{\text{prescribed},\mathbf{u}} \cup \Gamma_{\parallel,\mathbf{u}} \cup \Gamma_{\text{traction},\mathbf{u}} = \Gamma$ and $\Gamma_{D,T} \cup \Gamma_{N,T} = \Gamma$. No boundary conditions have to be posed for the compositional fields at those parts of the boundary where flow is either tangential to the boundary or points outward.

2.1.3 Comments on the final set of equations

ASPECT solves these equations in essentially the form stated. In particular, the form given in (1) implies that the pressure p we compute is in fact the *total pressure*, i.e., the sum of hydrostatic pressure and dynamic pressure (however, see Section 2.4 for more information on this, as well as the extensive discussion of this issue in [KHB12]). Consequently, it allows the direct use of this pressure when looking up pressure dependent material parameters.

2.2 Coefficients

The equations above contain a significant number of coefficients that we will discuss in the following. In the most general form, many of these coefficients depend nonlinearly on the solution variables pressure p, temperature T and, in the case of the viscosity, on the strain rate $\varepsilon(\mathbf{u})$. If compositional fields $\mathfrak{c} = \{c_1, \dots, c_C\}$ are present (i.e., if $C > 0$), coefficients may also depend on them. Alternatively, they may be parameterized as a function of the spatial variable $\mathbf{x}$. ASPECT allows both kinds of parameterizations.

Note: One of the next versions of ASPECT will actually iterate out nonlinearities in the material description. However, in the current version, we simply evaluate all nonlinear dependence of coefficients at the solution variables from the previous time step or a solution suitably extrapolated from the previous time steps.

Note that below we will discuss examples of the dependence of coefficients on other quantities; which dependence is actually implemented in the code is a different matter. As we will discuss in Sections 5 and 7, some versions of these models are already implemented and can be selected from the input parameter file; others are easy to add to ASPECT by providing self-contained descriptions of a set of coefficients that the rest of the code can then use without a need for further modifications.

Concretely, we consider the following coefficients and dependencies:

- *The viscosity* $\eta = \eta(p, T, \varepsilon(\mathbf{u}), \mathfrak{c}, \mathbf{x})$*:* Units $\text{Pa} \cdot \text{s} = \text{kg}\frac{1}{\text{m}\cdot\text{s}}$.

  The viscosity is the proportionality factor that relates total forces (external gravity minus pressure gradients) and fluid velocities $\mathbf{u}$. The simplest models assume that η is constant, with the constant often chosen to be on the order of 10^{21}Pa s.

  More complex (and more realistic) models assume that the viscosity depends on pressure, temperature and strain rate. Since this dependence is often difficult to quantify, one modeling approach is to make η spatially dependent.

- *The density* $\rho = \rho(p, T, \mathfrak{c}, \mathbf{x})$*:* Units $\frac{\text{kg}}{\text{m}^3}$.

  In general, the density depends on pressure and temperature, both through pressure compression, thermal expansion, and phase changes the material may undergo as it moves through the pressure-temperature phase diagram.

  The simplest parameterization for the density is to assume a linear dependence on temperature, yielding the form $\rho(T) = \rho_{\text{ref}}[1 - \beta(T - T_{\text{ref}})]$ where ρ_{ref} is the reference density at temperature T_{ref} and β is the linear thermal expansion coefficient. For the earth mantle, typical values for this parameterization would be $\rho_{\text{ref}} = 3300\frac{\text{kg}}{\text{m}^3}$, $T_{\text{ref}} = 293\text{K}$, $\beta = 2 \cdot 10^{-5}\frac{1}{\text{K}}$.

- *The gravity vector* $\mathbf{g} = \mathbf{g}(\mathbf{x})$*:* Units $\frac{\text{m}}{\text{s}^2}$.

  Simple models assume a radially inward gravity vector of constant magnitude (e.g., the surface gravity of Earth, $9.81\frac{\text{m}}{\text{s}^2}$), or one that can be computed analytically assuming a homogeneous mantle density.

  A physically self-consistent model would compute the gravity vector as $\mathbf{g} = -\nabla\varphi$ with a gravity potential φ that satisfies $-\Delta\varphi = 4\pi G\rho$ with the density ρ from above and G the universal constant of gravity. This would provide a gravity vector that changes as a function of time. Such a model is not currently implemented.

- *The specific heat capacity* $C_p = C_p(p, T, \mathfrak{c}, \mathbf{x})$*:* Units $\frac{\text{J}}{\text{kg}\cdot\text{K}} = \frac{\text{m}^2}{\text{s}^2\cdot\text{K}}$.

  The specific heat capacity denotes the amount of energy needed to increase the temperature of one kilogram of material by one degree. Wikipedia lists a value of 790 $\frac{\text{J}}{\text{kg}\cdot\text{K}}$ for granite[1] For the earth mantle, a value of 1250 $\frac{\text{J}}{\text{kg}\cdot\text{K}}$ is within the range suggested by the literature.

- *The thermal conductivity* $k = k(p, T, \mathfrak{c}, \mathbf{x})$*:* Units $\frac{\text{W}}{\text{m}\cdot\text{K}} = \frac{\text{kg}\cdot\text{m}}{\text{s}^3\cdot\text{K}}$.

  The thermal conductivity denotes the amount of thermal energy flowing through a unit area for a given temperature gradient. It depends on the material and as such will from a physical perspective depend on pressure and temperature due to phase changes of the material as well as through different mechanisms for heat transport (see, for example, the partial transparency of perovskite, the most abundant material in the earth mantle, at pressures above around 120 GPa [BRV+04]).

[1] See http://en.wikipedia.org/wiki/Specific_heat.

As a rule of thumb for its order of magnitude, Wikipedia quotes values of 1.83–2.90$\frac{\text{W}}{\text{m}\cdot\text{K}}$ for sandstone and 1.73–3.98$\frac{\text{W}}{\text{m}\cdot\text{K}}$ for granite.[2] The values in the mantle are almost certainly higher than this though probably not by much. The exact value is not really all that important: heat transport through convection is several orders of magnitude more important than through thermal conduction.

The thermal conductivity k is often expressed in terms of the *thermal diffusivity* κ using the relation $k = \rho C_p \kappa$.

- *The intrinsic specific heat production $H = H(\mathbf{x})$:* Units $\frac{\text{W}}{\text{kg}} = \frac{\text{m}^2}{\text{s}^3}$.

  This term denotes the intrinsic heating of the material, for example due to the decay of radioactive material. As such, it depends not on pressure or temperature, but may depend on the location due to different chemical composition of material in the earth mantle. The literature suggests a value of $\gamma = 7.4 \cdot 10^{-12}\frac{\text{W}}{\text{kg}}$.

- *The change of entropy ΔS at a phase transition together with the derivatives of the phase function $X = X(p, T, \mathfrak{c}, \mathbf{x})$ with regard to temperature and pressure:* Units $\frac{\text{J}}{\text{kgK}^2}$ $(-\Delta S\frac{\partial X}{\partial T})$ and $\frac{\text{m}^3}{\text{kgK}}$ $(\Delta S\frac{\partial X}{\partial p})$.

  When material undergoes a phase transition, the entropy changes due to release or consumption of latent heat. However, phase transitions occur gradually and for a given chemical composition it depends on temperature and pressure which phase prevails. Thus, the latent heat release can be calculated from the change of entropy ΔS and the derivatives of the phase function $\frac{\partial X}{\partial T}$ and $\frac{\partial X}{\partial p}$. These values have to be provided by the material model, separately for the coefficient $-\Delta S\frac{\partial X}{\partial T}$ on the left-hand side and $\Delta S\frac{\partial X}{\partial p}$ on the right-hand side of the temperature equation. However, they may be either approximated with the help of an analytic phase function, employing data from a thermodynamic database or in any other way that seems appropriate to the user.

2.3 Dimensional or non-dimensionalized equations?

Equations (1)–(3) are stated in the physically correct form. One would usually interpret them in a way that the various coefficients such as the viscosity, density and thermal conductivity η, ρ, κ are given in their correct physical units, typically expressed in a system such as the meter, kilogram, second (MKS) system that is part of the SI system. This is certainly how we envision ASPECT to be used: with geometries, material models, boundary conditions and initial values to be given in their correct physical units. As a consequence, when ASPECT prints information about the simulation onto the screen, it typically does so by using a postfix such as `m/s` to indicate a velocity or `W/m^2` to indicate a heat flux.

> **Note:** For convenience, output quantities are sometimes provided in units meters per *year* instead of meters per *second* (velocities) or in *years* instead of *seconds* (the current time, the time step size); this conversion happens at the time output is generated, and is not part of the solution process. Whether this conversion should happen is determined by the flag "`Use years in output instead of seconds`" in the input file, see Section 5.2. Obviously, this conversion from seconds to years only makes sense if the model is described in physical units rather than in non-dimensionalized form, see below.

That said, in reality, ASPECT has no preferred system of units as long as every material constant, geometry, time, etc., are all expressed in the same system. In other words, it is entirely legitimate to implement geometry and material models in which the dimension of the domain is one, density and viscosity are one, and the density variation as a function of temperature is scaled by the Rayleigh number – i.e., to use the usual non-dimensionalization of the Boussinesq equations. Some of the cookbooks in Section 6 use this non-dimensional form; for example, the simplest cookbook in Section 6.2.1 as well as the SolCx, SolKz

[2]See `http://en.wikipedia.org/wiki/Thermal_conductivity` and `http://en.wikipedia.org/wiki/List_of_thermal_conductivities`.

and inclusion benchmarks in Sections 6.4.3, are such cases. Whenever this is the case, output showing units `m/s` or `W/m^2` clearly no longer have a literal meaning. Rather, the unit postfix must in this case simply be interpreted to mean that the number that precedes the first is a velocity and a heat flux in the second case.

In other words, whether a computation uses physical or non-dimensional units really depends on the geometry, material, initial and boundary condition description of the particular case under consideration – ASPECT will simply use whatever it is given. Whether one or the other is the more appropriate description is a decision we purposefully leave to the user. There are of course good reasons to use non-dimensional descriptions of realistic problems, rather than to use the original form in which all coefficients remain in their physical units. On the other hand, there are also downsides:

- Non-dimensional descriptions, such as when using the Rayleigh number to indicate the relative strength of convective to diffusive thermal transport, have the advantage that they allow to reduce a system to its essence. For example, it is clear that we get the same behavior if one increases both the viscosity and the thermal expansion coefficient by a factor of two because the resulting Rayleigh number; similarly, if we were to increase the size of the domain by a factor of 2 and thermal diffusion coefficient by a factor of 8. In both of these cases, the non-dimensional equations are exactly the same. On the other hand, the equations in their physical unit form are different and one may not see that the result of this variations in coefficients will be exactly the same as before. Using non-dimensional variables therefore reduces the space of independent parameters one may have to consider when doing parameter studies.

- From a practical perspective, equations (1)–(3) are often ill-conditioned in their original form: the two sides of each equation have physical units different from those of the other equations, and their numerical values are often vastly different.[3] Of course, these values can not be compared: they have different physical units, and the ratios between these values depends on whether we choose to measure lengths in meters or kilometers, for example. Nevertheless, when implementing these equations in software, at one point or another, we have to work with numbers and at this point the physical units are lost. If one does not take care at this point, it is easy to get software in which all accuracy is lost due to round-off errors. On the other hand, non-dimensionalization typically avoids this since it normalizes all quantities so that values that appear in computations are typically on the order of one.

- On the downside, the numbers non-dimensionalized equations produce are not immediately comparable to ones we know from physical experiments. This is of little concern if all we have to do is convert every output number of our program back to physical units. On the other hand, it is more difficult and a source of many errors if this has to be done inside the program, for example, when looking up the viscosity as a pressure-, temperature- and strain-rate-dependent function: one first has to convert pressure, temperature and strain rate from non-dimensional to physical units, look up the corresponding viscosity in a table, and then convert the viscosity back to non-dimensional quantities. Getting this right at every one of the dozens or hundreds of places inside a program and using the correct (but distinct) conversion factors for each of these quantities is both a challenge and a possible source of errors.

- From a mathematical viewpoint, it is typically clear how an equation needs to be non-dimensionalized if all coefficients are constant. However, how is one to normalize the equations if, as is the case in the earth mantle, the viscosity varies by several orders of magnitude? In cases like these, one has to choose a reference viscosity, density, etc. While the resulting non-dimensionalization retains the universality of parameters in the equations, as discussed above, it is not entirely clear that this would also retain the numerical stability if the reference values are poorly chosen.

As a consequence of such considerations, most codes in the past have used non-dimensionalized models. This was aided by the fact that until recently and with notable exceptions, many models had constant

[3] To illustrate this, consider convection in the Earth as a back-of-the-envelope example. With the length scale of the mantle $L = 3 \cdot 10^6\ m$, viscosity $\eta = 10^{24}\ kg/m/s$, density $\rho = 3 \cdot 10^3\ kg/m^3$ and a typical velocity of $U = 0.1\ m/year = 3 \cdot 10^{-9}\ m/s$, we get that the friction term in (1) has size $\eta U/L^2 \approx 3 \cdot 10^2\ kg/m/s^3$. On the other hand, the term $\nabla \cdot (\rho u)$ in the continuity equation (2) has size $\rho U/L \approx 3 \cdot 10^{-12}\ kg/s/m^3$. In other words, their *numerical values* are 14 orders of magnitude apart.

coefficients and the difficulties associated with variable coefficients were not a concern. On the other hand, our goal with ASPECT is for it to be a code that solves realistic problems using complex models and that is easy to use. Thus, we allow users to input models in physical or non-dimensional units, at their discretion. We believe that this makes the description of realistic models simpler. On the other hand, ensuring numerical stability is not something users should have to be concerned about, and is taken care of in the implementation of ASPECT's core (see the corresponding section in [KHB12]).

2.4 Static or dynamic pressure?

One could reformulate equation (1) somewhat. To this end, let us say that we would want to represent the pressure p as the sum of two parts that we will call static and dynamic, $p = p_s + p_d$. If we assume that p_s is already given, then we can replace (1) by

$$-\nabla \cdot 2\eta \nabla \mathbf{u} + \nabla p_d = \rho \mathbf{g} - \nabla p_s.$$

One typically chooses p_s as the pressure one would get if the whole medium were at rest – i.e., as the hydrostatic pressure. This pressure can be computed noting that (1) reduces to

$$\nabla p_s = \rho(p_s, T_s, \mathbf{x})\mathbf{g}$$

in the absence of any motion where T_s is some static temperature field (see also Section 2.6). This, our rewritten version of (1) would look like this:

$$-\nabla \cdot 2\eta \nabla \mathbf{u} + \nabla p_d = [\rho(p, T, \mathbf{x}) - \rho(p_s, T_s, \mathbf{x})]\, \mathbf{g}.$$

In this formulation, it is clear that the quantity that drives the fluid flow is in fact the *buoyancy* caused by the *variation* of densities, not the density itself.

This reformulation has a number of advantages and disadvantages:

- One can notice that in many realistic cases, the dynamic component p_d of the pressure is orders of magnitude smaller than the static component p_s. For example, in the earth, the two are separated by around 6 orders of magnitude at the bottom of the earth mantle. Consequently, if one wants to solve the linear system that arises from discretization of the original equations, one has to solve it a significant degree of accuracy (6–7 digits) to get the dynamic part of the pressure correct to even one digit. This entails a very significant numerical effort, and one that is not necessary if we can split the pressure in a way so that the pre-computed static pressure p_s (or, rather, the density using the static pressure and temperature from which p_s results) absorbs the dominant part and one only has to compute the remaining, dynamic pressure to 2 or 3 digits of accuracy, rather than the corresponding 7–8 for the total pressure.

- On the other hand, the pressure p_d one computes this way is not immediately comparable to quantities that we use to look up pressure-dependent quantities such as the density. Rather, one needs to first find the static pressure as well (see Section 2.6) and add the two together before they can be used to look up material properties or to compare them with experimental results. Consequently, if the pressure a program outputs (either for visualization, or in the internal interfaces to parts of the code where users can implement pressure- and temperature-dependent material properties) is only the dynamic component, then all of the consumers of this information need to convert it into the total pressure when comparing with physical experiments. Since any code implementing realistic material models has a great many of these places, there is a large potential for inadvertent errors and bugs.

- Finally, the definition of a reference density $\rho(p_s, T_s, \mathbf{x})$ derived from static pressures and temperatures is only simple if we have incompressible models and under the assumption that the temperature-induced density variations are small compared to the overall density. In this case, we can choose $\rho(p_s, T_s, \mathbf{x}) = \rho_0$ with a constant reference density ρ_0. On the other hand, for more complicated

models, it is not a priori clear which density to choose since we first need to compute static pressures and temperatures – quantities that satisfy equations that introduce boundary layers, may include phase changes releasing latent heat, and where the density may have discontinuities at certain depths, see Section 2.6.

Thus, if we compute adiabatic pressures and temperatures $\bar{p}_s, \bar{T}_s$ under the assumption of a thermal boundary layer worth 900 Kelvin at the top, and we get a corresponding density profile $\bar{\rho} = \rho(\bar{p}_s, \bar{T}_s, \mathbf{x})$, but after running for a few million years the temperature turns out to be so that the top boundary layer has a jump of only 800 Kelvin with corresponding adiabatic pressures and temperatures $\hat{p}_s, \hat{T}_s$, then a more appropriate density profile would be $\hat{\rho} = \rho(\hat{p}_s, \hat{T}_s, \mathbf{x})$.

The problem is that it may well be that the erroneously computed density profile $\hat{\rho}$ does *not* lead to a separation where $|p_d| \ll |p_s|$ because, especially if the material undergoes phase changes, there will be entire areas of the computational domain in which $|\rho - \hat{\rho}_s| \ll |\rho|$ but $|\rho - \bar{\rho}_s| \not\ll |\rho|$. Consequently the benefits of lesser requirements on the iterative linear solver would not be realized.

We do note that most of the codes available today and that we are aware of split the pressure into static and dynamic parts nevertheless, either internally or require the user to specify the density profile as the difference between the true and the hydrostatic density. This may, in part, be due to the fact that historically most codes were written to solve problems in which the medium was considered incompressible, i.e., where the definition of a static density was simple.

On the other hand, we intend ASPECT to be a code that can solve more general models for which this definition is not as simple. As a consequence, we have chosen to solve the equations as stated originally – i.e., we solve for the *full* pressure rather than just its *dynamic* component. With most traditional methods, this would lead to a catastrophic loss of accuracy in the dynamic pressure since it is many orders of magnitude smaller than the total pressure at the bottom of the earth mantle. We avoid this problem in ASPECT by using a cleverly chosen iterative solver that ensures that the full pressure we compute is accurate enough so that the dynamic pressure can be extracted from it with the same accuracy one would get if one were to solve for only the dynamic component. The methods that ensure this are described in detail in [KHB12] and in particular in the appendix of that paper.

2.5 Pressure normalization

The equations described above, (1)–(3), only determine the pressure p up to an additive constant. On the other hand, since the pressure appears in the definition of many of the coefficients, we need a pressure that has some sort of *absolute* definition. A physically useful definition would be to normalize the pressure in such a way that the average pressure along the "surface" has a prescribed value where the geometry description (see Section 7.3.3) has to determine which part of the boundary of the domain is the "surface" (we call a part of the boundary the "surface" if its depth is "close to zero").

Typically, one will choose this average pressure to be zero, but there is a parameter "`Surface pressure`" in the input file (see Section 5.2) to set it to a different value. One may want to do that, for example, if one wants to simulate the earth mantle without the overlying lithosphere. In that case, the "surface" would be the interface between mantle and lithosphere, and the average pressure at the surface to which the solution of the equations will be normalized should in this case be the hydrostatic pressure at the bottom of the lithosphere.

An alternative is to normalize the pressure in such a way that the *average* pressure throughout the domain is zero or some constant value. This is not a useful approach for most geodynamics applications but is common in benchmarks for which analytic solutions are available. Which kind of normalization is chosen is determined by the "`Pressure normalization`" flag in the input file, see Section 5.2.

2.6 Initial conditions and the adiabatic pressure/temperature

Equations (1)–(3) require us to pose initial conditions for the temperature, and this is done by selecting one of the existing models for initial conditions in the input parameter file, see Section 5.51. The equations

themselves do not require that initial conditions are specified for the velocity and pressure variables (since there are no time derivatives on these variables in the model).

Nevertheless, a nonlinear solver will have difficulty converging to the correct solution if we start with a completely unphysical pressure for models in which coefficients such as density ρ and viscosity η depend on the pressure and temperature. To this end, ASPECT computes pressure and temperature fields $p_{\rm ad}(z), T_{\rm ad}(z)$ that satisfy adiabatic conditions:

$$\rho C_p \frac{\mathrm{d}}{\mathrm{d}z} T_{\rm ad}(z) = \frac{\partial \rho}{\partial T} T_{\rm ad}(z) g_z, \tag{13}$$

$$\frac{\mathrm{d}}{\mathrm{d}z} p_{\rm ad}(z) = \rho g_z, \tag{14}$$

where strictly speaking g_z is the magnitude of the vertical component of the gravity vector field, but in practice we take the magnitude of the entire gravity vector.

These equations can be integrated numerically starting at $z = 0$, using the depth dependent gravity field and values of the coefficients $\rho = \rho(p, T, z), C_p = C_p(p, T, z)$. As starting conditions at $z = 0$ we choose a pressure $p_{\rm ad}(0)$ equal to the average surface pressure (often chosen to be zero, see Section 2.5), and an adiabatic surface temperature $T_{\rm ad}(0)$ that is also selected in the input parameter file.

Note: The adiabatic surface temperature is often chosen significantly higher than the actual surface temperature. For example, on earth, the actual surface temperature is on the order of 290 K, whereas a reasonable adiabatic surface temperature is maybe 1200 K. The reason is that the bulk of the mantle is more or less in thermal equilibrium with a thermal profile that corresponds to the latter temperature, whereas the very low actual surface temperature and the very high bottom temperature at the core-mantle boundary simply induce a thermal boundary layer. Since the temperature and pressure profile we compute using the equations above are simply meant to be good starting points for nonlinear solvers, it is important to choose this profile in such a way that it covers most of the mantle well; choosing an adiabatic surface temperature of 290 K would yield a temperature and pressure profile that is wrong almost throughout the entire mantle.

2.7 Compositional fields

The last of the basic equations, (4), describes the evolution of a set of variables $c_i(\mathbf{x}, t), i = 1 \dots C$ that we typically call *compositional fields* and that we often aggregate into a vector $\mathfrak{c}$.

Compositional fields were originally intended to track what their name suggest, namely the chemical composition of the convecting medium. In this interpretation, they composition is a quantity that is simply advected along passively, i.e., it would satisfy the equation

$$\frac{\partial \mathfrak{c}}{\partial t} + \mathbf{u} \cdot \nabla \mathfrak{c} = 0.$$

However, these compositional fields participate in determining the values of the various coefficients as discussed in Section 2.2.

That said, over time compositional fields have shown to be a much more useful tool than originally intended. For example, they can be used to track where material comes from and goes to (see Section 6.2.4) and, if one allows for a reaction rate $\mathfrak{q}$ on the right hand side,

$$\frac{\partial \mathfrak{c}}{\partial t} + \mathbf{u} \cdot \nabla \mathfrak{c} = \mathfrak{q},$$

then one can also model interaction between species – for example to simulate phase changes where one compositional field indicating a particular phase transforms into another phase depending on pressure and temperature, or where several phases combine to other phases. Inside the material model, the interaction is given by `reaction_term` which is defined as $\triangle t \cdot \mathfrak{q}$.

Modeling reactions between different compositional fields often involves finding an equilibrium state between state between different fields because chemical reactions happen on a much faster time scale than transport. In other words, one then often assumes that there is a $\mathfrak{c}^*(p,T)$ so that

$$\mathfrak{q}(p,T,\varepsilon(\mathbf{u}),\mathfrak{c}^*(p,T)) = 0.$$

Consequently, the material model methods that deal with source terms for the compositional fields need to compute an *increment* $\Delta\mathfrak{c}$ to the previous value of the compositional fields so that the sum of the previous values and the increment equals $\mathfrak{c}^*$. This is opposed to the usual approach of evaluating the right hand side term $\mathfrak{q}$, which corresponds to a *rate*, instead of an increment.

On the other hand, there are other uses of compositional fields that do not actually have anything to do with quantities that can be considered related to compositions. For example, one may define a field that tracks the grain size of rocks. If the strain rate is high, then the grain size decreases as the rocks break. If the temperature is high enough, then grains heal and their size increases again. Such "damage" models would then call for an equation of the form (assuming one uses only a single compositional field)

$$\frac{\partial c}{\partial t} + \mathbf{u}\cdot\nabla c = q(T,c),$$

where in the simplest case one could postulate

$$q(T,c) = -\alpha c + \beta \max\{T - T_{\text{healing}}, 0\}c.$$

One would then use this compositional field in the definition of the viscosity of the material: more damage means lower viscosity because the rocks are weaker.

In cases like this, there is only a single compositional field and it is not in permanent equilibrium. Consequently, the increment implementations of material models in ASPECT need to compute is typically the rate $q(T,c)$ times the time step. In other words, if you compute a reaction rate inside the material model you need to multiply it by the time step size before returning the value.

Compositional fields have proven to be surprisingly versatile tools to model all sorts of components of models that go beyond the simple Stokes plus temperature set of equations. Play with them!

2.8 Constitutive laws

Equation (1) describes buoyancy-driven flow in an isotropic fluid where strain rate is related to stress by a scalar (possibly spatially variable) multiplier, η. For some material models it is useful to generalize this relationship to anisotropic materials, or other exotic constitutive laws. For these cases ASPECT can optionally include a generalized, fourth-order tensor field as a material model state variable which changes equation (1) to

$$-\nabla\cdot\left[2\eta\left(C\varepsilon(\mathbf{u}) - \frac{1}{3}(tr(C\varepsilon(\mathbf{u})))\mathbf{1}\right)\right] + \nabla p = \rho\mathbf{g} \qquad \text{in } \Omega \tag{15}$$

and the shear heating term in equation (3) to

$$\begin{aligned} &\dots \\ +2\eta\left(C\varepsilon(\mathbf{u}) - \frac{1}{3}(tr(C\varepsilon(\mathbf{u})))\mathbf{1}\right) : &\left(\varepsilon(\mathbf{u}) - \frac{1}{3}(\nabla\cdot\mathbf{u})\mathbf{1}\right) \\ &\dots \end{aligned} \tag{16}$$

where $C = C_{ijkl}$ is defined by the material model. For physical reasons, C needs to be a symmetric rank-4 tensor: i.e., when multiplied by a symmetric (strain rate) tensor of rank 2 it needs to return another symmetric tensor of rank 2. In mathematical terms, this means that $C_{ijkl} = C_{jikl} = C_{ijlk} = C_{jilk}$. Energy considerations also require that C is positive definite: i.e., for any $\varepsilon \neq 0$, the scalar $\varepsilon : (C\varepsilon)$ must be positive.

This functionality can be optionally invoked by any material model that chooses to define a C field, and falls back to the default case ($C = \mathbb{I}$) if no such field is defined. It should be noted that η still appears in equations (15) and (16). C is therefore intended to be thought of as a "director" tensor rather than a replacement for the viscosity field, although in practice either interpretation is okay.

2.9 Numerical methods

There is no shortage in the literature for methods to solve the equations outlined above. The methods used by ASPECT use the following, interconnected set of strategies in the implementation of numerical algorithms:

- *Mesh adaptation:* Mantle convection problems are characterized by widely disparate length scales (from plate boundaries on the order of kilometers or even smaller, to the scale of the entire earth). Uniform meshes can not resolve the smallest length scale without an intractable number of unknowns. Fully adaptive meshes allow resolving local features of the flow field without the need to refine the mesh globally. Since the location of plumes that require high resolution change and move with time, meshes also need to be adapted every few time steps.

- *Accurate discretizations:* The Boussinesq problem upon which most models for the earth mantle are based has a number of intricacies that make the choice of discretization non-trivial. In particular, the finite elements chosen for velocity and pressure need to satisfy the usual compatibility condition for saddle point problems. This can be worked around using pressure stabilization schemes for low-order discretizations, but high-order methods can yield better accuracy with fewer unknowns and offer more reliability. Equally important is the choice of a stabilization method for the highly advection-dominated temperature equation. ASPECT uses a nonlinear artificial diffusion method for the latter.

- *Efficient linear solvers:* The major obstacle in solving the Boussinesq system is the saddle-point nature of the Stokes equations. Simple linear solvers and preconditioners can not efficiently solve this system in the presence of strong heterogeneities or when the size of the system becomes very large. ASPECT uses an efficient solution strategy based on a block triangular preconditioner utilizing an algebraic multigrid that provides optimal complexity even up to problems with hundreds of millions of unknowns.

- *Parallelization of all of the steps above:* Global mantle convection problems frequently require extremely large numbers of unknowns for adequate resolution in three dimensional simulations. The only realistic way to solve such problems lies in parallelizing computations over hundreds or thousands of processors. This is made more complicated by the use of dynamically changing meshes, and it needs to take into account that we want to retain the optimal complexity of linear solvers and all other operations in the program.

- *Modularity of the code:* A code that implements all of these methods from *scratch* will be unwieldy, unreadable and unusable as a community resource. To avoid this, we build our implementation on widely used and well tested libraries that can provide researchers interested in extending it with the support of a large user community. Specifically, we use the DEAL.II library [BHK07, BHK12] for meshes, finite elements and everything discretization related; the TRILINOS library [HBH+05, H+11] for scalable and parallel linear algebra; and P4EST [BWG11] for distributed, adaptive meshes. As a consequence, our code is freed of the mundane tasks of defining finite element shape functions or dealing with the data structures of linear algebra, can focus on the high-level description of what is supposed to happen, and remains relatively compact. The code will also automatically benefit from improvements to the underlying libraries with their much larger development communities. ASPECT is extensively documented to enable other researchers to understand, test, use, and extend it.

Rather than detailing the various techniques upon which ASPECT is built, we refer to the paper by Kronbichler, Heister and Bangerth [KHB12] that gives a detailed description and rationale for the various building blocks.

2.10 Simplifications of the basic equations

There are two common variations to equations (1)–(3) that are frequently used and that make the system much simpler to solve and analyze: assuming that the fluid is incompressible (the Boussinesq approximation) and a linear dependence of the density on the temperature with constants that are otherwise independent of the solution variables. These are discussed in the following; ASPECT has run-time parameters that allow both of these simpler models to be used.

2.10.1 The Boussinesq approximation: Incompressibility

The original Boussinesq approximation assumes that the density can be considered constant in all occurrences in the equations with the exception of the buoyancy term on the right hand side of (1). The primary result of this assumption is that the continuity equation (2) will now read

$$\nabla \cdot \mathbf{u} = 0.$$

This makes the equations *much* simpler to solve: First, because the divergence operation in this equation is the transpose of the gradient of the pressure in the momentum equation (1), making the system of these two equations symmetric. And secondly, because the two equations are now linear in pressure and velocity (assuming that the viscosity η and the density ρ are considered fixed). In addition, one can drop all terms involving $\nabla \cdot \mathbf{u}$ from the left hand side of the momentum equation (1) as well as from the shear heating term on the right hand side of (3); while dropping these terms does not affect the solution of the equations, it makes assembly of linear systems faster. In addition, in the incompressible case, one needs to neglect the adiabatic heating term $\frac{\partial \rho}{\partial T} T \mathbf{u} \cdot \mathbf{g}$ on the right hand side of (3).

From a physical perspective, the assumption that the density is constant in the continuity equation but variable in the momentum equation is of course inconsistent. However, it is justified if the variation is small since the momentum equation can be rewritten to read

$$-\nabla \cdot 2\eta\varepsilon(\mathbf{u}) + \nabla p_d = (\rho - \rho_0)\mathbf{g},$$

where p_d is the *dynamic* pressure and ρ_0 is the constant reference density. This makes it clear that the true driver of motion is in fact the *deviation* of the density from its background value, however small this value is: the resulting velocities are simply proportional to the density variation, not to the absolute magnitude of the density.

As such, the Boussinesq approximation can be justified. On the other hand, given the real pressures and temperatures at the bottom of the earth mantle, it is arguable whether the density can be considered to be almost constant. Most realistic models predict that the density of mantle rocks increases from somewhere around 3300 at the surface to over 5000 kilogram per cubic meters at the core mantle boundary, due to the increasing lithostatic pressure. While this appears to be a large variability, if the density changes slowly with depth, this is not in itself an indication that the Boussinesq approximation will be wrong. To this end, consider that the continuity equation can be rewritten as $\frac{1}{\rho}\nabla \cdot (\rho \mathbf{u}) = 0$, which we can multiply out to obtain

$$\nabla \cdot \mathbf{u} + \frac{1}{\rho}\mathbf{u} \cdot \nabla \rho = 0.$$

The question whether the Boussinesq approximation is valid is then whether the second term (the one omitted in the Boussinesq model) is small compared to the first. To this end, consider that the velocity can change completely over length scales of maybe 10 km, so that $\nabla \cdot \mathbf{u} \approx \|u\|/10\text{km}$. On the other hand, given a smooth dependence of density on pressure, the length scale for variation of the density is the entire earth mantle, i.e., $\frac{1}{\rho}\mathbf{u} \cdot \nabla \rho \approx \|u\| 0.5/3000\text{km}$ (given a variation between minimal and maximal density of 0.5 times the density itself). In other words, for a smooth variation, the contribution of the compressibility to the continuity equation is very small. This may be different, however, for models in which the density changes rather abruptly, for example due to phase changes at mantle discontinuities.

In summary, models that use the approximation of incompressibility solve the following set of equations instead of (1)–(3):

$$-\nabla \cdot [2\eta\varepsilon(\mathbf{u})] + \nabla p = \rho \mathbf{g} \qquad \text{in } \Omega, \tag{17}$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega, \tag{18}$$

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot k \nabla T = \rho H + 2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u}) \qquad \text{in } \Omega, \tag{19}$$

where the coefficients $\eta, \rho, \mathbf{g}, C_p$ may possible depend on the solution variables.

Note: As we will see in Section 7, it is easy to add new material models to ASPECT. Each model can decide whether it wants to use the Boussinesq approximation or not. The description of the models in Section 5.66 also gives an answer which of the models already implemented uses the approximation or considers the material sufficiently compressible to go with the fully compressible continuity equation.

2.10.2 Almost linear models

A further simplification can be obtained if one assumes that all coefficients with the exception of the density do not depend on the solution variables but are, in fact, constant. In such models, one typically assumes that the density satisfies a relationship of the form $\rho = \rho(T) = \rho_0(1 - \beta(T - T_0))$ with a small thermal expansion coefficient β and a reference density ρ_0 that is attained at temperature T_0. Since the thermal expansion is considered small, this naturally leads to the following variant of the Boussinesq model discussed above:

$$-\nabla \cdot [2\eta\varepsilon(\mathbf{u})] + \nabla p = \rho_0(1 - \beta(T - T_0))\mathbf{g} \qquad \text{in } \Omega,$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega,$$

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot k \nabla T = \rho H + 2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u}) \qquad \text{in } \Omega,$$

If the gravitational acceleration $\mathbf{g}$ results from a gravity potential φ via $\mathbf{g} = -\nabla\varphi$, then one can rewrite the equations above in the following, commonly used form:[4]

$$-\nabla \cdot [2\eta\varepsilon(\mathbf{u})] + \nabla p_d = -\beta\rho_0 T \mathbf{g} \qquad \text{in } \Omega, \tag{20}$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega, \tag{21}$$

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot k \nabla T = \rho H + 2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u}) \qquad \text{in } \Omega, \tag{22}$$

where $p_d = p + \rho_0(1 + \beta T_0)\varphi$ is the dynamic pressure, as opposed to the total pressure $p = p_d + p_s$ that also includes the hydrostatic pressure $p_s = -\rho_0(1 + \beta T_0)\varphi$. Note that the right hand side forcing term in (20) is now only the deviation of the gravitational force from the force that would act if the material were at temperature T_0.

Under the assumption that all other coefficients are constant, one then arrives at equations in which the only nonlinear terms are the advection term, $\mathbf{u} \cdot \nabla T$, and the shear friction, $2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u})$, in the temperature equation (22). This facilitates the use of a particular class of time stepping scheme in which one does not solve the whole set of equations at once, iterating out nonlinearities as necessary, but instead in each time step solves first the Stokes system with the previous time step's temperature, and then uses the so-computed velocity to solve the temperature equation. These kind of time stepping schemes are often referred to as *IMPES* methods (they originate in the porous media flow community, where the acronym stands for *Im*plicit *P*ressure, *E*xplicit *S*aturation). For details see [KHB12].

[4] Note, however, that ASPECT does not solve the equations in the form given in (20)–(22). Rather, it takes the original form with the real density, not the variation of the density. That said, you can use the formulation (20)–(22) by implementing a material model (see Section 7.3.1) in which the density in fact has the form $\rho(T) = \beta\rho_0 T$ even though this is not physical.

2.10.3 Compressible models

In the compressible case, the conservation of mass equation in equation (2) becomes $\nabla \cdot (\rho \mathbf{u}) = 0$ instead of $\nabla \cdot \mathbf{u} = 0$, which is nonlinear and no longer symmetric to the ∇p term in the force balance equation (1), making solving and preconditioning the resulting linear and nonlinear systems difficult. To make this work in ASPECT, we consequently reformulate this equation. Dividing by ρ and applying the product rule of differentiation gives

$$\frac{1}{\rho}\nabla \cdot (\rho \mathbf{u}) = \nabla \cdot \mathbf{u} + \frac{1}{\rho}\nabla \rho \cdot \mathbf{u}.$$

We will now make two basic assumptions: First, the variation of the density $\rho(p, T, \mathbf{x}, \mathfrak{c})$ is dominated by the dependence on the (total) pressure; in other words, $\nabla \rho \approx \frac{\partial \rho}{\partial p}\nabla p$. This assumption is primarily justified by the fact that, in the Earth mantle, the density increases by at least 50% between Earth crust and the core-mantle boundary due to larger pressure there. Secondly, we assume that the pressure is dominated by the static pressure, which can be written as $\nabla p \approx \nabla p_s \approx \rho \mathbf{g}$. This is essentially motivated by the slowness of the movement in the Earth or, alternatively, based on the fact that the viscosity is so large. This finally allows us to write

$$\frac{1}{\rho}\nabla \rho \cdot \mathbf{u} \approx \frac{1}{\rho}\frac{\partial \rho}{\partial p}\nabla p \cdot \mathbf{u} \approx \frac{1}{\rho}\frac{\partial \rho}{\partial p}\nabla p_s \cdot \mathbf{u} \approx \frac{1}{\rho}\frac{\partial \rho}{\partial p}\rho \mathbf{g} \cdot \mathbf{u}$$

so we get

$$\nabla \cdot \mathbf{u} = -\frac{1}{\rho}\frac{\partial \rho}{\partial p}\rho \mathbf{g} \cdot \mathbf{u} \tag{23}$$

where $\frac{1}{\rho}\frac{\partial \rho}{\partial p}$ is often referred to as the compressibility.

In the implementation used in ASPECT, this equation replaces (2). It has the advantage that it retains the symmetry of the Stokes equations if we can treat the right hand side of (23) as known. We do so by evaluating ρ and $\mathbf{u}$ using the solution from the last time step (or values extrapolated from previous time steps).

2.11 Free surface calculations

In reality the boundary conditions of a convecting Earth are not no-slip or free slip (i.e., no normal velocity). Instead, we expect that a free surface is a more realistic approximation, since air and water should not prevent the flow of rock upward or downward. This means that we require zero stress on the boundary, or $\sigma \cdot \mathbf{n} = 0$, where $\sigma = 2\eta\varepsilon(\mathbf{u})$. In general there will be flow across the boundary with this boundary condition. To conserve mass we must then advect the boundary of the domain in the direction of fluid flow. Thus, using a free surface necessitates that the mesh be dynamically deformable.

2.11.1 Arbitrary Lagrangian-Eulerian implementation

The question of how to handle the motion of the mesh with a free surface is challenging. Eulerian meshes are well behaved, but they do not move with the fluid motions, which makes them difficult for use with free surfaces. Lagrangian meshes do move with the fluid, but they quickly become so distorted that remeshing is required. ASPECT implements an Arbitrary Lagrangian-Eulerian (ALE) framework for handling motion of the mesh. The ALE approach tries to retain the benefits of both the Lagrangian and the Eulerian approaches by allowing the mesh motion $\mathbf{u}_m$ to be largely independent of the fluid. The mass conservation condition requires that $\mathbf{u}_m \cdot \mathbf{n} = \mathbf{u} \cdot \mathbf{n}$ on the free surface, but otherwise the mesh motion is unconstrained, and should be chosen to keep the mesh as well behaved as possible.

ASPECT uses a Laplacian scheme for calculating the mesh velocity. The mesh velocity is calculated by solving

$$
\begin{aligned}
-\Delta \mathbf{u}_m &= 0 && \text{in } \Omega, && (24)\\
\mathbf{u}_m &= (\mathbf{u}\cdot\mathbf{n})\,\mathbf{n} && \text{on } \partial\Omega_{\text{free surface}}, && (25)\\
\mathbf{u}_m\cdot\mathbf{n} &= 0 && \text{on } \partial\Omega_{\text{free slip}}, && (26)\\
\mathbf{u}_m &= 0 && \text{on } \partial\Omega_{\text{Dirichlet}}. && (27)
\end{aligned}
$$

After this mesh velocity is calculated, the mesh vertices are time-stepped explicitly. This scheme has the effect of choosing a minimally distorting perturbation to the mesh. Because the mesh velocity is no longer zero in the ALE approach, we must then correct the Eulerian advection terms in the advection system with the mesh velocity (see, e.g. [DHPRF04]). For instance, the temperature equation (22) becomes

$$\rho C_p \left(\frac{\partial T}{\partial t} + (\mathbf{u} - \mathbf{u}_m)\cdot \nabla T \right) - \nabla \cdot k \nabla T = \rho H \quad \text{in } \Omega.$$

2.11.2 Free surface stabilization

Small disequilibria in the location of a free surface can cause instabilities in the surface position and result in a "sloshing" instability. This may be countered with a quasi-implicit free surface integration scheme described in [KMM10]. This scheme enters the governing equations as a small stabilizing surface traction that prevents the free surface advection from overshooting its true position at the next time step. ASPECT implements this stabilization, the details of which may be found in [KMM10].

An example of a simple model which uses a free surface may be found in Section 6.2.6.

2.12 Nullspace removal

The Stokes equation (1) only involves symmetric gradients of the velocity, and as such the velocity is determined only up to rigid-body motions (that is to say, translations and rotations). For many simulations the boundary conditions will fully specify the velocity solution, but for some combinations of geometries and boundary conditions the solution will still be underdetermined. In the language of linear algebra, the Stokes system may have a nullspace.

Usually the user will be able to determine beforehand whether their problem has a nullspace. For instance, a model in a spherical shell geometry with free-slip boundary conditions at the top and bottom will have a rigid-body rotation in its nullspace (but not translations, as the boundary conditions do not allow flow through them). That is to say, the solver may be able to come up with a solution to the Stokes operator, but that solution plus an arbitrary rotation is also an equally valid solution.

Another example is a model in a Cartesian box with periodic boundary conditions in the x-direction, and free slip boundaries on the top and bottom. This setup has arbitrary translations along the x-axis in its nullspace, so any solution plus an arbitrary x-translation is also a solution.

A solution with some small power in these nullspace modes should not affect the physics of the simulation. However, the timestepping of the model is based on evaluating the maximum velocities in the solution, and having unnecessary motions can severely shorten the time steps that ASPECT takes. Furthermore, rigid body motions can make postprocessing calculations and visualization more difficult to interpret.

ASPECT allows the user to specify if their model has a nullspace. If so, any power in the nullspace is calculated and removed from the solution after every timestep. There are two varieties of nullspace removal implemented: removing net linear/angular momentum, and removing net translations/rotations.

For removing linear momentum we search for a constant velocity vector $\mathbf{c}$ such that

$$\int_\Omega \rho(\mathbf{u} - \mathbf{c}) = 0$$

This may be solved by realizing that $\int_\Omega \rho \mathbf{u} = \mathbf{p}$, the linear momentum, and $\int_\Omega \rho = M$, the total mass of the model. Then we find

$$\mathbf{c} = \mathbf{p}/M$$

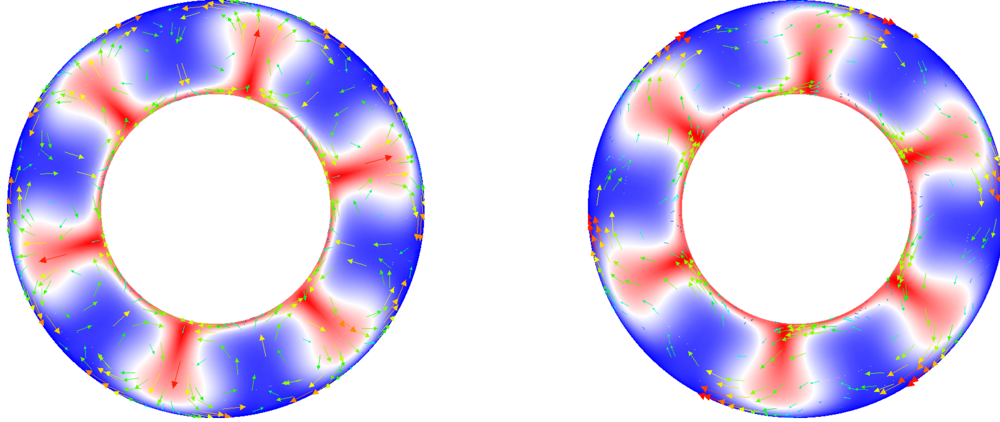


Figure 1: *Example of nullspace removal. On the left the nullspace (a rigid rotation) is removed, and the velocity vectors accurately show the mantle flow. On the right there is a significant clockwise rotation to the velocity solution which is making the more interesting flow features difficult to see.*

which is subtracted off of the velocity solution.

Removing the angular momentum is similar, though a bit more complicated. We search for a rotation vector ω such that

$$\int_\Omega \rho(\mathbf{x} \times (\mathbf{u} - \omega \times \mathbf{x})) = 0$$

Recognizing that $\int_\Omega \rho\mathbf{x} \times \mathbf{u} = \mathbf{H}$, the angular momentum, and $\int_\Omega \rho\mathbf{x} \times \omega \times \mathbf{x} = \mathbf{I} \cdot \omega$, the moment of inertia dotted into the sought-after vector, we can solve for ω:

$$\omega = \mathbf{I^{-1}} \cdot \mathbf{H}$$

A rotation about the rotation vector ω is then subtracted from the velocity solution.

Removing the net translations/rotations are identical to their momentum counterparts, but for those the density is dropped from the formulae. For most applications the density should not vary so wildly that there will be an appreciable difference between the two varieties, though removing linear/angular momentum is more physically motivated.

The user can flag the nullspace for removal by setting the `Remove nullspace` option, as described in Section 5.87. Figure 1 shows the result of removing angular momentum from a convection model in a 2D annulus with free-slip velocity boundary conditions.

3 Installation

This is a brief explanation of how to install all the required software and ASPECT itself.

3.1 System prerequisites

In order to install ASPECT and its dependencies, you should have your system set up to be able to compile and link programs. Additionally, ASPECT needs a number of widely used libraries that are available for most operating systems. Therefore, you will need compilers for C, C++ and Fortran, the GNU make system, the CMake build system, and the libraries and header files of BLAS, LAPACK and zlib, which is used for compressing the output data. To use more than one process for your computations you will need to install a MPI library, its headers and the necessary executables to run MPI programs. There are some optional packages for additional features, like the HDF5 libraries for additional output formats, PETSC for alternative solvers, and Numdiff for checking ASPECT's test results with reasonable accuracy, but these

are not strictly required, and in some operating systems they are not available as packages but need to be compiled from scratch. Finally, for obtaining a recent development version of ASPECT you will need the git version control system.

An exemplary command to obtain all required packages on Ubuntu 14.04 would be:

```
sudo apt-get install build-essential \
                     cmake \
                     gcc \
                     g++ \
                     gfortran \
                     git \
                     libblas-dev \
                     liblapack-dev \
                     libopenmpi-dev \
                     numdiff \
                     openmpi-bin \
                     zlib1g-dev
```

3.2 Software prerequisites

ASPECT builds on a few other libraries that are widely used in the computational science area and that provide most of the lower-level functionality such as finite element descriptions or parallel linear algebra. Specifically, it builds on DEAL.II which in turn uses Trilinos and P4EST. These need to be installed first before you can compile and run ASPECT. All of these libraries can readily be installed in a user's home directory, without the need to modify the overall system directories.

The following steps should guide you through the installation of these prerequisites:

1. *Trilinos:* Trilinos can be downloaded from http://trilinos.org/download/. At the current time we recommend Trilinos Version 11.4.x.[5] For installation instructions see the deal.II README file on installing Trilinos. Note that you have to configure with MPI by using

   ```
   TPL_ENABLE_MPI:BOOL=ON
   ```

   in the call to cmake. After that, run `make install`.

2. P4EST*:* Download and install P4EST as described in the deal.II p4est installation instructions. This is done using the `p4est-setup.sh`; do not use the P4EST stand-alone installation instructions.

3. DEAL.II*:* The current version of ASPECT requires DEAL.II version 8.2 or later. This version can be downloaded and installed from `https://www.dealii.org/download.html`.

4. *Configuring and compiling* DEAL.II*:* Now it is time to configure DEAL.II. To this end, follow the DEAL.II installation instructions. Note that DEAL.II recently made the switch to cmake, so the configuration changed. Make sure you enable MPI. A typical command line would look like this:

   ```
   mkdir build
   cd build
   cmake -DDEAL_II_WITH_MPI=ON \
         -DCMAKE_INSTALL_PREFIX=/u/username/deal-installed/ \
         -DTRILINOS_DIR=/u/username/trilinos-11.4.1/ \
         -DP4EST_DIR=/u/username/p4est-0.3.4.1/ \
         ../deal.II
   ```

[5]Other versions of Trilinos like 10.6.x and 10.8.x have bugs that make these versions unusable for our purpose. The DEAL.II ReadMe file provides a list of versions that are known to work without bugs with DEAL.II.

if the Trilinos and p4est packages have been installed in the subdirectory `/u/username/`. Make sure the configuration succeeds and detects the MPI compilers correctly. For more information see the documentation of deal.II.

Now you are ready to compile deal.II by running `make install`. If you have multiple processor cores, feel free to do `make install -jN` where `N` is the number of processors in your machine to accelerate the process.

5. *Testing your installation:* Test that your installation works by running the `step-32` example that you can find in `$DEAL_II_DIR/examples/step-32`. Compile by running `make` and run with `mpirun -n 2 ./step-32`.

6. You may now want to set the environment variable[6] `DEAL_II_DIR` to the directory where you *installed* deal.II.

3.3 Obtaining ASPECT and initial configuration

The development version of ASPECT can be downloaded by executing the command

```
git clone https://github.com/geodynamics/aspect.git
```

If `$DEAL_II_DIR` points to your deal.II installation, you can configure ASPECT by running

```
cmake .
```

in the `aspect` directory created by the `git clone` command above. If you did not set `$DEAL_II_DIR` you have to supply cmake with the location:

```
cmake -DDEAL_II_DIR=/u/username/deal-installed/ .
```

An alternative would be to configure ASPECT as an out-of-source build. Similar to the configuration of deal.II, you would need to create a build directory and specify an install directory using -DCMAKE_INSTALL_PREFIX. The instructions in the following sections assume an in-source build, so you need to adapt the location of the ASPECT binary.

3.4 Compiling ASPECT and generating documentation

After downloading ASPECT and having built the libraries it builds on, you can compile it by typing

```
make
```

on the command line (or `make -jN` if you have multiple processors in your machine, where `N` is the number of processors). This builds the ASPECT executable which will reside in the main directory and will be named `./aspect`. If you intend to modify ASPECT for your own experiments, you may want to also generate documentation about the source code. This can be done using the command

```
cd doc; make
```

which assumes that you have the `doxygen` documentation generation tool installed. Most Linux distributions have packages for `doxygen`. The result will be the file `doc/doxygen/index.html` that is the starting point for exploring the documentation.

[6] For bash this would be adding the line `export DEAL_II_DIR=/path/to/deal-installed/` to the file `~/.bashrc`. Then close the terminal and open it again to activate the change.

3.5 Compiling a static ASPECT executable

ASPECT defaults to a dynamically linked executable, which saves disk space and build time. In some circumstances however, it is preferred to generate a statically linked executable that incorporates all used libraries. This need may arise on large clusters on which libraries and loaded modules/variables on login nodes may be different from the ones available on compute nodes. The general build procedure in such a case equals the above explained instructions with the following differences:

1. *Trilinos*: Add the following lines to your cmake call:

   ```
   -DBUILD_SHARED_LIBS=OFF
   -DTPL_FIND_SHARED_LIBS=OFF
   ```

2. P4EST*:* Change items "–enable-shared" to "–enable-static" in p4est-setup.sh lines 83 and 97.

3. DEAL.II*:* Add the following lines to your call to cmake:

   ```
   -DDEAL_II_STATIC_EXECUTABLE=ON
   ```

4. ASPECT*:* If everything above is set up correctly, there is no need for any configuration change to ASPECT's build procedure. You should see the following cmake output from ASPECT:

   ```
   -- Creating a statically linked executable
   -- Disabling dynamic loading of plugins from the input file
   ```

The here mentioned build was tested on a Cray XC30 cluster, which was set up for default static compiling and linking. On machines that default to dynamic linking additional compiler and/or linker flags may be required (e.g. "-fPIC" / "–static"). In case of questions send an email to the mailing list.

3.6 Installing and running ASPECT on Mac OS X

OS X has some eccentricities which can complicate installation of ASPECT. Currently the easiest and most reliable way to run ASPECT under Mac OS X Mavericks (10.9) and Yosemite (10.10) is to install and run the binary package for DEAL.II. The step-by-step process is described in detail, with screenshots, here: https://wiki.geodynamics.org/_media/software:aspect:aspect_yosemite_20150529.pdf

1. Install Cmake.

   CMake is a cross-platform, open-source build system that can be downloaded from http://www.cmake.org. After installation of CMake.app, the terminal command for cmake will be

   ```
   /Applications/CMake.app/Contents/bin/cmake
   ```

2. Download and install the parallel DEAL.II. This is the binary package for Mac OS .dmg file.

   ```
   open  https://github.com/dealii/dealii/releases/download/v8.2.1/dealii-8.2.1-parallel-bundle.dmg
   ```

   Open the downloaded disk image, and drag DEAL.II.app into the Applications folder. To start the DEAL.II app, double click the icon in the Applications folder or use the open command:

   ```
   open /Applications/deal.II.app
   ```

   DEAL.II app opens a terminal window and displays a DEAL.II message. DEAL.II.app will install all required libraries for ASPECT (p4est, parMeTiS, and Trilinos) and will include the environment variables needed to use these libraries.

3. Download the ASPECT source from the git repository.

```
cd $HOME/src
git clone https://github.com/geodynamics/aspect.git
```

Note: you **MUST** build and run ASPECT in the terminal window started by deal.II.

4. Build ASPECT

   (a) Go to the directory where you wish to install ASPECT and run the following commands:

   ```
   cmake .
   ```

   This should display something like:

   ```
   Project  aspect  set up with  deal.II-8.2.1  found at /Applications/deal.II.app/Contents/Resources
   ```

   (b) Run make:

   ```
   bash-3.2$ make
   Scanning dependencies of target aspect
   [  0%] Building CXX object CMakeFiles/aspect.dir/source/adiabatic_conditions/initial_profile.cc.o
   [  0%] Building CXX object CMakeFiles/aspect.dir/source/adiabatic_conditions/interface.cc.o
   ...
   Linking CXX executable aspect
   [100%] Built target aspect
   bash-3.2$ ls -l aspect
   -rwxr-xr-x  1 <name>  staff  19131292 May  7 15:02 aspect
   ```

   There may be warnings from the compiler, but if the ASPECT target is created then it was successful.

   (c) By default, ASPECT compiles the debug version of the code. To compile the optimized version:

   ```
   make release
   ```

   (d) Run make test

   ```
   make test
   ```

5. Run ASPECT.

   A reminder: you **must** run ASPECT on the terminal window which is opened by deal.II.app.
   To start ASPECT using MPI for parallelization, from the directory where you installed ASPECT:

   ```
   mpirun -np <# of processes> ./aspect <parameter file>
   ```

   To check quickly whether you are running ASPECT on the deal.II.app terminal, check the location of the `mpirun` command:

   ```
   bash-3.2$ which mpirun
   /Applications/deal.II.app/Contents/Resources/opt/openmpi-1.6.5/bin/mpirun
   ```

4 Running ASPECT

4.1 Overview

After compiling ASPECT as described above, you should have an executable file in the main directory. It can be called as follows:

```
./aspect parameter-file.prm
```

or, if you want to run the program in parallel, using something like

```
mpirun -np 32 ./aspect parameter-file.prm
```

to run with 32 processors. In either case, the argument denotes the (path and) name of a file that contains input parameters.[7] When you download ASPECT, there are a number of sample input files in the `cookbooks` directory, corresponding to the examples discussed in Section 6, and input files for some of the benchmarks discussed in Section 6.4 are located in the `benchmarks` directory. A full description of all parameters one can specify in these files is given in Section 5.

Running ASPECT with an input file will produce output that will look something like this (numbers will all be different, of course):

```
Number of active cells: 1,536 (on 5 levels)
Number of degrees of freedom: 20,756 (12,738+1,649+6,369)

*** Timestep 0:  t=0 years

   Rebuilding Stokes preconditioner...
   Solving Stokes system... 30+3 iterations.
   Solving temperature system... 8 iterations.

Number of active cells: 2,379 (on 6 levels)
Number of degrees of freedom: 33,859 (20,786+2,680+10,393)

*** Timestep 0:  t=0 years

   Rebuilding Stokes preconditioner...
   Solving Stokes system... 30+4 iterations.
   Solving temperature system... 8 iterations.

   Postprocessing:
     Writing graphical output: output/solution-00000
     RMS, max velocity:        0.0946 cm/year, 0.183 cm/year
     Temperature min/avg/max:  300 K, 3007 K, 6300 K
     Inner/outer heat fluxes:  1.076e+05 W, 1.967e+05 W

*** Timestep 1:  t=1.99135e+07 years

   Solving Stokes system... 30+3 iterations.
   Solving temperature system... 8 iterations.

   Postprocessing:
     Writing graphical output: output/solution-00001
     RMS, max velocity:        0.104 cm/year, 0.217 cm/year
     Temperature min/avg/max:  300 K, 3008 K, 6300 K
     Inner/outer heat fluxes:  1.079e+05 W, 1.988e+05 W
```

[7] As a special case, if you call ASPECT with an argument that consists of two dashes, "`--`", then the arguments will be read from the standard input stream of the program. In other words, you could type the input parameters into your shell window in this case (though that would be cumbersome, ASPECT would seem to hang until you finish typing all of your input into the window and then terminating the input stream by typing `Ctrl-D`). A more common case would be to use Unix pipes so that the default input of ASPECT is the output of another program, as in a command like `cat parameter-file.prm.in | mypreprocessor | ./aspect --`, where `mypreprocessor` would be a program of your choice that somehow transforms the file `parameter-file.prm.in` into a valid input file, for example to systematically vary one of the input parameters.

If you want to run ASPECT in parallel, you can do something like `cat parameter-file.prm.in | mypreprocessor | mpirun -np 4 ./aspect --`. In cases like this, `mpirun` only forwards the output of `mypreprocessor` to the first of the four MPI processes, which then sends the text to all other processors.

```
*** Timestep 2:  t=3.98271e+07 years

   Solving Stokes system... 30+3 iterations.
   Solving temperature system... 8 iterations.

   Postprocessing:
     RMS, max velocity:       0.111 cm/year, 0.231 cm/year
     Temperature min/avg/max: 300 K, 3008 K, 6300 K
     Inner/outer heat fluxes: 1.083e+05 W, 2.01e+05 W

*** Timestep 3:  t=5.97406e+07 years

...
```

This output was produced by a parameter file that, among other settings, contained the following values (we will discuss many such input files in Section 6:

```
set Dimension                     = 2
set End time                      = 2e9
set Output directory              = output

subsection Geometry model
  set Model name                  = spherical shell
end

subsection Mesh refinement
  set Initial global refinement   = 4
  set Initial adaptive refinement = 1
end

subsection Postprocess
  set List of postprocessors      = all
end
```

In other words, these run-time parameters specify that we should start with a geometry that represents a spherical shell (see Sections 5.32 and 5.38 for details). The coarsest mesh is refined 4 times globally, i.e., every cell is refined into four children (or eight, in 3d) 4 times. This yields the initial number of 1,536 cells on a mesh hierarchy that is 5 levels deep. We then solve the problem there once and, based on the number of adaptive refinement steps at the initial time set in the parameter file, use the solution so computed to refine the mesh once adaptively (yielding 2,379 cells on 6 levels) on which we start the computation over at time $t = 0$.

Within each time step, the output indicates the number of iterations performed by the linear solvers, and we generate a number of lines of output by the postprocessors that were selected (see Section 5.88). Here, we have selected to run all postprocessors that are currently implemented in ASPECT which includes the ones that evaluate properties of the velocity, temperature, and heat flux as well as a postprocessor that generates graphical output for visualization.

While the screen output is useful to monitor the progress of a simulation, its lack of a structured output makes it not useful for later plotting things like the evolution of heat flux through the core-mantle boundary. To this end, ASPECT creates additional files in the output directory selected in the input parameter file (here, the `output/` directory relative to the directory in which ASPECT runs). In a simple case, this will look as follows:

```
aspect> ls -l output/
total 780
-rw------- 1 b   9863 Dec  1 15:13 parameters.prm
-rw------- 1 b 306562 Dec  1 15:13 solution-00000.0000.vtu
```

```
-rw------- 1 b  97057 Nov 30 05:58 solution-00000.0001.vtu
...
-rw------- 1 b   1061 Dec  1 15:13 solution-00000.pvtu
-rw------- 1 b     35 Dec  1 15:13 solution-00000.visit
-rw------- 1 b 306530 Dec  1 15:13 solution-00001.0000.vtu
-rw------- 1 b   1061 Dec  1 15:13 solution-00001.pvtu
-rw------- 1 b     35 Dec  1 15:13 solution-00001.visit
...
-rw-r--r-- 1 b    997 Dec  1 15:13 solution.pvd
-rw-r--r-- 1 b    997 Dec  1 15:13 solution.visit
-rw------- 1 b    924 Dec  1 15:13 statistics
```

The purpose of these files is as follows:

- *A listing of all run-time parameters:* The `output/parameters.prm` file contains a complete listing of all run-time parameters. In particular, this includes the one that have been specified in the input parameter file passed on the command line, but it also includes those parameters for which defaults have been used. It is often useful to save this file together with simulation data to allow for the easy reproduction of computations later on.

- *Graphical output files:* One of the postprocessors you select when you say "all" in the parameter files is the one that generates output files that represent the solution at certain time steps. The screen output indicates that it has run at time step 0, producing output files of the form `output/solution-00000`. At the current time, the default is that ASPECT generates this output in VTK format[8] as that is widely used by a number of excellent visualization packages and also supports parallel visualization.[9] If the program has been run with multiple MPI processes, then the list of output files will look as shown above, with the base `solution-x.y` denoting that this the `x`th time we create output files and that the file was generated by the `y`th processor.

  VTK files can be visualized by many of the large visualization packages. In particular, the Visit and ParaView programs, both widely used, can read the files so created. However, while VTK has become a de-facto standard for data visualization in scientific computing, there doesn't appear to be an agreed upon way to describe which files jointly make up for the simulation data of a single time step (i.e., all files with the same `x` but different `y` in the example above). Visit and Paraview both have their method of doing things, through `.pvtu` and `.visit` files. To make it easy for you to view data, ASPECT simply creates both kinds of files in each time step in which graphical data is produced.

  The final two files of this kind, `solution.pvd` and `solution.visit`, are files that describes to Paraview and Visit, respectively, which `solution-xxxx.pvtu` and `solution-xxxx.yyyy.vtu` jointly form a complete simulation. In the former case, the file lists the `.pvtu` files of all timesteps together with the simulation time to which they correspond. In the latter case, it actually lists all `.vtu` that belong to one simulation, grouped by the timestep they correspond to. To visualize an entire simulation, not just a single time step, it is therefore simplest to just load one of these files, depending on whether you use Paraview or Visit.[10]

  For more on visualization, see also Section 4.4.

[8]The output is in fact in the VTU version of the VTK file format. This is the XML-based version of this file format in which contents are compressed. Given that typical file sizes for 3d simulation are substantial, the compression saves a significant amount of disk space.

[9]The underlying DEAL.II package actually supports output in around a dozen different formats, but most of them are not very useful for large-scale, 3d, parallel simulations. If you need a different format than VTK, you can select this using the run-time parameters discussed in Section 5.100.

[10]At the time of writing this, current versions of Visit (starting with version 2.5.1) actually have a bug that prevents them from successfully reading the `solution.visit` or `solution-xxxx.visit` files – Visit believes that each of these files corresponds to an individual time step, rather than that a whole group of files together form one time step. This bug is not fixed in Visit 2.6.3, but may be fixed in later versions.

- *A statistics file:* The `output/statistics` file contains statistics collected during each time step, both from within the simulator (e.g., the current time for a time step, the time step length, etc.) as well as from the postprocessors that run at the end of each time step. The file is essentially a table that allows for the simple production of time trends. In the example above, it looks like this:

```
# 1: Time step number
# 2: Time (years)
# 3: Iterations for Stokes solver
# 4: Time step size (year)
# 5: Iterations for temperature solver
# 6: Visualization file name
# 7: RMS velocity (m/year)
# 8: Max. velocity (m/year)
# 9: Minimal temperature (K)
# 10: Average temperature (K)
# 11: Maximal temperature (K)
# 12: Average nondimensional temperature (K)
# 13: Core-mantle heat flux (W)
# 14: Surface heat flux (W)
0 0.0000e+00 33 2.9543e+07 8                    "" 0.0000 0.0000   0.0000    0.0000 ...
0 0.0000e+00 34 1.9914e+07 8 output/solution-00000 0.0946 0.1829 300.0000 3007.2519 ...
1 1.9914e+07 33 1.9914e+07 8 output/solution-00001 0.1040 0.2172 300.0000 3007.8406 ...
2 3.9827e+07 33 1.9914e+07 8                    "" 0.1114 0.2306 300.0000 3008.3939 ...
```

  The actual columns you have in your statistics file may differ from the ones above, but the format of this file should be obvious. Since the hash mark is a comment marker in many programs (for example, `gnuplot` ignores lines in text files that start with a hash mark), it is simple to plot these columns as time series. Alternatively, the data can be imported into a spreadsheet and plotted there.

  Note: As noted in Section 2.3, ASPECT can be thought of as using the meter-kilogram-second (MKS, or SI) system. Unless otherwise noted, the quantities in the output file are therefore also in MKS units.

  A simple way to plot the contents of this file is shown in Section 4.4.2.

- *Output files generated by other postprocessors:* Similar to the `output/statistics` file, several of the existing postprocessors one can select from the parameter file generate their data in their own files in the output directory. For example, ASPECT can write depth-average statistics into `output/depth_average.gnuplot`. This is done by the "depth average" postprocessor and the user can control how often this file is updated, as well as what graphical file format to use (if anything other than `gnuplot` is desired).

  By default, the data is written in text format that can be easily displayed by e.g. gnuplot. For an example, see Figure 2. The plot shows how an initially linear temperature profile forms upper and lower boundary layers.

4.2 Selecting between 2d and 3d runs

ASPECT can solve both two- and three-dimensional problems. You select which one you want by putting a line like the following into the parameter file (see Section 5):

```
set Dimension                         = 2
```

Internally, dealing with the dimension builds on a feature in DEAL.II, upon which ASPECT is based, that is called *dimension-independent programming.* In essence, what this does is that you write your code

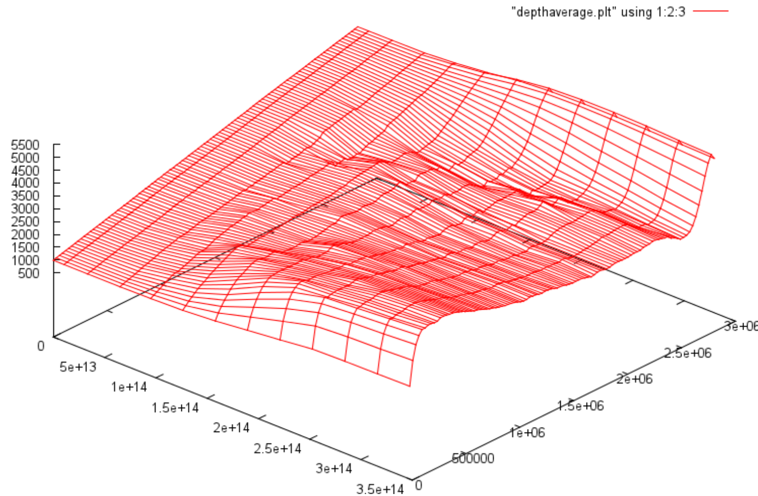


Figure 2: *Example output for depth average statistics. On the left axis are 13 time steps, on the right is the depth (from the top at 0 to the bottom of the mantle on the far right), and the upwards pointing axis is the average temperature. This plot is generated by gnuplot, but the depth averages can be written in many other output formats as well, if preferred (see Section 5.90).*

only once in a way so that the space dimension is a variable (or, in fact, a template parameter) and you can compile the code for either 2d or 3d. The advantage is that codes can be tested and debugged in 2d where simulations are relatively cheap, and the same code can then be re-compiled and executed in 3d where simulations would otherwise be prohibitively expensive for finding bugs; it is also a useful feature when scoping out whether certain parameter settings will have the desired effect by testing them in 2d first, before running them in 3d. This feature is discussed in detail in the deal.II tutorial program step-4. Like there, all the functions and classes in ASPECT are compiled for both 2d and 3d. Which dimension is actually called internally depends on what you have set in the input file, but in either case, the machine code generated for 2d and 3d results from the same source code and should, thus, contain the same set of features and bugs. Running in 2d and 3d should therefore yield comparable results. Be prepared to wait much longer for computations to finish in the latter case, however.

4.3 Debug or optimized mode

ASPECT utilizes a deal.II feature called *debug mode*. By default, ASPECT uses debug mode, i.e., it calls a version of the deal.II library that contain lots of checks for the correctness of function arguments, the consistency of the internal state of data structure, etc. If you program with deal.II, for example to extend ASPECT, it has been our experience over the years that, by number, most programming errors are of the kind where one forgets to initialize a vector, one accesses data that has not been updated, one tries to write into a vector that has ghost elements, etc. If not caught, the result of these bugs is that parts of the program use invalid data (data written into ghost elements is not communicated to other processors), that operations simply make no sense (adding vectors of different length), that memory is corrupted (writing past the end of an array) or, in rare and fortunate cases, that the program simply crashes.

Debug mode is designed to catch most of these errors: It enables some 7,300 assertions (as of late 2011) in deal.II where we check for errors like the above and, if the condition is violated, abort the program with a detailed message that shows the failed check, the location in the source code, and a stacktrace how the program got there. The downside of debug mode is, of course, that it makes the program much slower – depending on application by a factor of 4–10. An example of the speedup one can get is shown in Section 6.2.1.

ASPECT by default uses debug mode because most users will want to play with the source code, and

because it is also a way to verify that the compilation process worked correctly. If you have verified that the program runs correctly with your input parameters, for example by letting it run for the first 10 time steps, then you can switch to optimized mode by compiling ASPECT with the command[11]

```
make release
```

and then compile using

```
make
```

To switch back to debug mode type:

```
make debug
```

> **Note:** It goes without saying that if you make significant modifications to the program, you should do the first runs in debug mode to verify that your program still works as expected.

4.4 Visualizing results

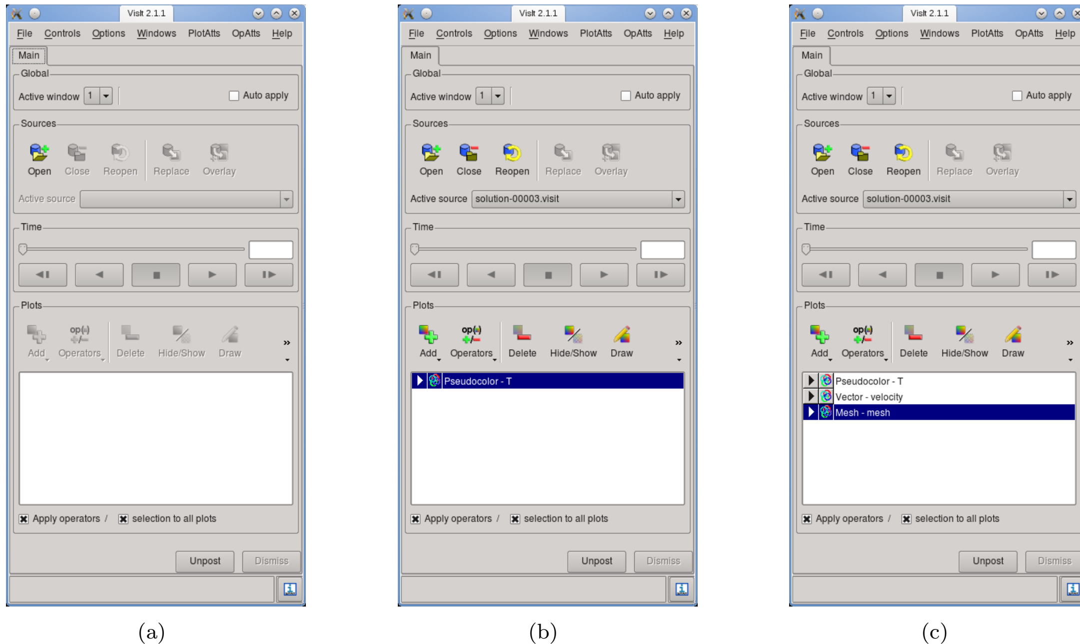
Among the postprocessors that can be selected in the input parameter file (see Sections 4.1 and 5.100) are some that can produce files in a format that can later be used to generate a graphical visualization of the solution variables $\mathbf{u}, p$ and T at select time steps, or of quantities derived from these variables (for the latter, see Section 7.3.9).

By default, the files that are generated are in VTU format, i.e., the XML-based, compressed format defined by the VTK library, see `http://public.kitware.com/VTK/`. This file format has become a broadly accepted pseudo-standard that many visualization program support, including two of the visualization programs used most widely in computational science: Visit (see `https://visit.llnl.gov/`) and ParaView (see `http://www.paraview.org/`). The VTU format has a number of advantages beyond being widely distributed:

- It allows for compression, keeping files relatively small even for sizeable computations.
- It is a structured XML format, allowing other programs to read it without too much trouble.
- It has a degree of support for parallel computations where every processor would only write that part of the data to a file that this processor in fact owns, avoiding the need to communicate all data to a single processor that then generates a single file. This requires a master file for each time step that then contains a reference to the individual files that together make up the output of a single time step. Unfortunately, there doesn't appear to be a standard for these master records; however, both ParaView and Visit have defined a format that each of these programs understand and that requires placing a file with ending `.pvtu` or `.visit` into the same directory as the output files from each processor. Section 4.1 gives an example of what can be found in the output directory.

> **Note:** You can select other formats for output than VTU, see the run-time parameters in Section 5.100. However, none of the numerous formats currently implemented in DEAL.II other than the VTK/VTU formats allows for splitting up data over multiple files in case of parallel computations, thus making subsequent visualization of the entire volume impossible. Furthermore, given the amount of data ASPECT can produce, the compression that is part of the VTU format is an important part of keeping data manageable.

[11] Note that this procedure also changed with the switch to cmake.



(a) (b) (c)

Figure 3: *Main window of Visit, illustrating the different steps of adding content to a visualization.*

4.4.1 Visualization the graphical output using *Visit*

In the following, let us discuss the process of visualizing a 2d computation using Visit. The steps necessary for other visualization programs will obviously differ but are, in principle, similar.

To this end, let us consider a simulation of convection in a box-shaped, 2d region (see the "cookbooks" section, Section 6, and in particular Section 6.2.1 for the input file for this particular model). We can run the program with 4 processors using

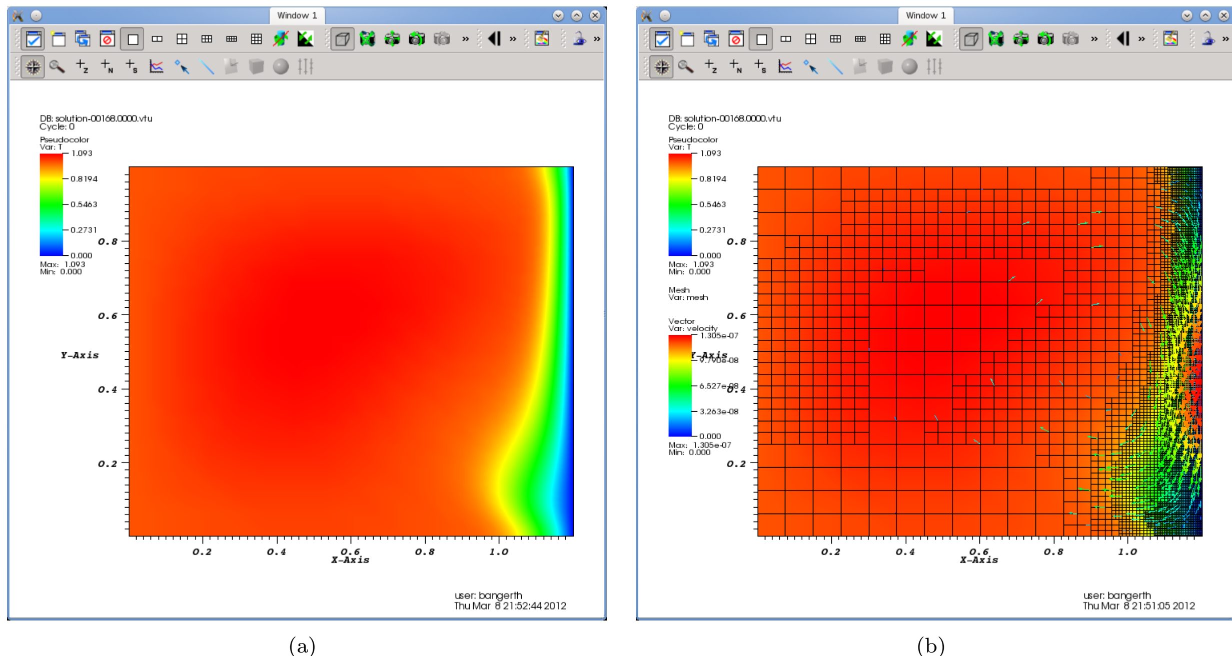
```
mpirun -np 4 ./aspect cookbooks/convection-box.prm
```

Letting the program run for a while will result in several output files as discussed in Section 4.1 above.

In order to visualize one time step, follow these steps:[12]

- *Selecting input files:* As mentioned above, in parallel computations we usually generate one output file per processor in each time step for which visualization data is produced (see, however, Section 4.4.3). To tell Visit which files together make up one time step, ASPECT creates a `solution-NNNNN.visit` file in the output directory. To open it, start Visit, click on the "Open" button in the "Sources" area of its main window (see Fig. 3(a)) and select the file you want. Alternatively, you can also select files using the "File > Open" menu item, or hit the corresponding keyboard short-cut. After adding an input source, the "Sources" area of the main window should list the selected file name.
- *Selecting what to plot:* ASPECT outputs all sorts of quantities that characterize the solution, such as temperature, pressure, velocity, and many others on demand (see Section 5.100). Once an input file has been opened, you will want to add graphical representations of some of this data to the still empty canvas. To this end, click on the "Add" button of the "Plots" area. The resulting menu provides a number of different kinds of plots. The most important for our purpose are: (i) "Pseudocolor" allows the visualization of a scalar field (e.g., temperature, pressure, density) by using a color field. (ii)

[12]The instructions and screenshots were generated with Visit 2.1. Later versions of Visit differ slightly in the arrangement of components of the graphical user interface, but the workflow and general idea remains unchanged.



(a) (b)

Figure 4: *Display window of Visit, showing a single plot and one where different data is overlaid.*

"Vector" displays a vector-valued field (e.g., velocity) using arrows. (iii) "Mesh" displays the mesh. The "Contour", "Streamline" and "Volume" options are also frequently useful, in particular in 3d.

Let us choose the "Pseudocolor" item and select the temperature field as the quantity to plot. Your main window should now look as shown in Fig. 3(b). Then hit the "Draw" button to make Visit generate data for the selected plots. This will yield a picture such as shown in Fig. 4(a) in the display window of Visit.

- *Overlaying data:* Visit can overlay multiple plots in the same view. To this end, add another plot to the view using again the "Add" button to obtain the menu of possible plots, then the "Draw" button to actually draw things. For example, if we add velocity vectors and the mesh, the main window looks as in Fig. 3(c) and the main view as in Fig. 4(b).

- *Adjusting how data is displayed:* Without going into too much detail, if you double click onto the name of a plot in the "Plots" window, you get a dialog in which many of the properties of this plot can be adjusted. Further details can be changed by using "Operators" on a plot.

- *Making the output prettier:* As can be seen in Fig. 4, Visit by default puts a lot of clutter around the figure – the name of the user, the name of the input file, color bars, axes labels and ticks, etc. This may be useful to explore data in the beginning but does not yield good pictures for presentations or publications. To reduce the amount of information displayed, go to the "Controls > Annotations" menu item to get a dialog in which all of these displays can be selectively switched on and off.

- *Saving figures:* To save a visualization into a file that can then be included into presentations and publications, go to the menu item "File > Save window". This will create successively numbered files in the directory from which Visit was started each time a view is saved. Things like the format used for these files can be chosen using the "File > Set save options" menu item. We have found that one can often get better looking pictures by selecting the "Screenshot" method in this dialog.

More information on all of these topics can be found in the Visit documentation, see https://visit.llnl.gov/. We have also recorded video lectures demonstrating this process interactively at http://www.youtube.com/watch?v=3ChnUxqtt08 for Visit, and at http://www.youtube.com/watch?v=w-65jufR-bc for Paraview.

4.4.2 Visualizing statistical data

In addition to the graphical output discussed above, ASPECT produces a statistics file that collects information produced during each time step. For the remainder of this section, let us assume that we have run ASPECT with the input file discussed in Section 6.2.1, simulating convection in a box. After running ASPECT, you will find a file called `statistics` in the output directory that, at the time of writing this, looked like this: This file has a structure that looks (at the time of writing this section) like this:

```
# 1: Time step number
# 2: Time (seconds)
# 3: Number of mesh cells
# 4: Number of Stokes degrees of freedom
# 5: Number of temperature degrees of freedom
# 6: Iterations for temperature solver
# 7: Iterations for Stokes solver
# 8: Velocity iterations in Stokes preconditioner
# 9: Schur complement iterations in Stokes preconditioner
# 10: Time step size (seconds)
# 11: RMS velocity (m/s)
# 12: Max. velocity (m/s)
# 13: Minimal temperature (K)
# 14: Average temperature (K)
# 15: Maximal temperature (K)
# 16: Average nondimensional temperature (K)
# 17: Outward heat flux through boundary with indicator 0 ("left") (W)
# 18: Outward heat flux through boundary with indicator 1 ("right") (W)
# 19: Outward heat flux through boundary with indicator 2 ("bottom") (W)
# 20: Outward heat flux through boundary with indicator 3 ("top") (W)
# 21: Visualization file name
 0 0.0000e+00 256 2467 1089  0 29 30 29 1.2268e-02 1.79026783e+00 2.54322608e+00
 1 1.2268e-02 256 2467 1089 32 29 30 30 3.7388e-03 5.89844152e+00 8.35160076e+00
 2 1.6007e-02 256 2467 1089 20 28 29 29 2.0239e-03 1.09071922e+01 1.54298908e+01
 3 1.8031e-02 256 2467 1089 15 27 28 28 1.3644e-03 1.61759153e+01 2.28931189e+01
 4 1.9395e-02 256 2467 1089 13 26 27 27 1.0284e-03 2.14465789e+01 3.03731397e+01
 5 2.0424e-02 256 2467 1089 11 25 26 26 8.2812e-04 2.66110761e+01 3.77180480e+01
```

In other words, it first lists what the individual columns mean with a hash mark at the beginning of the line and then has one line for each time step in which the individual columns list what has been explained above.[13]

This file is easy to visualize. For example, one can import it as a whitespace separated file into a spreadsheet such as Microsoft Excel or OpenOffice/LibreOffice Calc and then generate graphs of one column against another. Or, maybe simpler, there is a multitude of simple graphing programs that do not need the overhead of a full fledged spreadsheet engine and simply plot graphs. One that is particularly simple to use and available on every major platform is `Gnuplot`. It is extensively documented at http://www.gnuplot.info/.

[13] With input files that ask for initial adaptive refinement, the first time step may appear twice because we solve on a mesh that is globally refined and we then start the entire computation over again on a once adaptively refined mesh (see the parameters in Section 5.82 for how to do that).

Gnuplot is a command line program in which you enter commands that plot data or modify the way data is plotted. When you call it, you will first get a screen that looks like this:

```
/home/user/aspect/output gnuplot

        G N U P L O T
        Version 4.6 patchlevel 0    last modified 2012-03-04
        Build System: Linux x86_64

        Copyright (C) 1986-1993, 1998, 2004, 2007-2012
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type set to 'qt'
gnuplot>
```

At the prompt on the last line, you can then enter commands. Given the description of the individual columns given above, let us first try to plot the heat flux through boundary 2 (the bottom boundary of the box), i.e., column 19, as a function of time (column 2). This can be achieved using the following command:

```
plot "statistics" using 2:19
```

The left panel of Fig. 5 shows what Gnuplot will display in its output window. There are many things one can configure in these plots (see the Gnuplot manual referenced above). For example, let us assume that we want to add labels to the x- and y-axes, use not just points but lines and points for the curves, restrict the time axis to the range $[0, 0.2]$ and the heat flux axis to $[-10 : 10]$, plot not only the flux through the bottom but also through the top boundary (column 20) and finally add a key to the figure, then the following commands achieve this:

```
set xlabel "Time"
set ylabel "Heat flux"
set style data linespoints
plot [0:0.2][-10:10] "statistics" using 2:19 title "Bottom boundary", \
                     "statistics" using 2:20 title "Top boundary"
```

If a line gets too long, you can continue it by ending it in a backslash as above. This is rarely used on the command line but useful when writing the commands above into a script file, see below. We have done it here to get the entire command into the width of the page.

For those who are lazy, Gnuplot allows to abbreviate things in many different ways. For example, one can abbreviate most commands. Furthermore, one does not need to repeat the name of an input file if it is the same as the previous one in a plot command. Thus, instead of the commands above, the following abbreviated form would have achieved the same effect:

```
se xl "Time"
se yl "Heat flux"
se sty da lp
pl [:0.2][-10:10] "statistics" us 2:19 t "Bottom boundary", "" us 2:20 t "Top boundary"
```

This is of course unreadable at first but becomes useful once you become more familiar with the commands offered by this program.

Once you have gotten the commands that create the plot you want right, you probably want to save it into a file. Gnuplot can write output in many different formats. For inclusion in publications, either eps or png are the most common. In the latter case, the commands to achieve this are

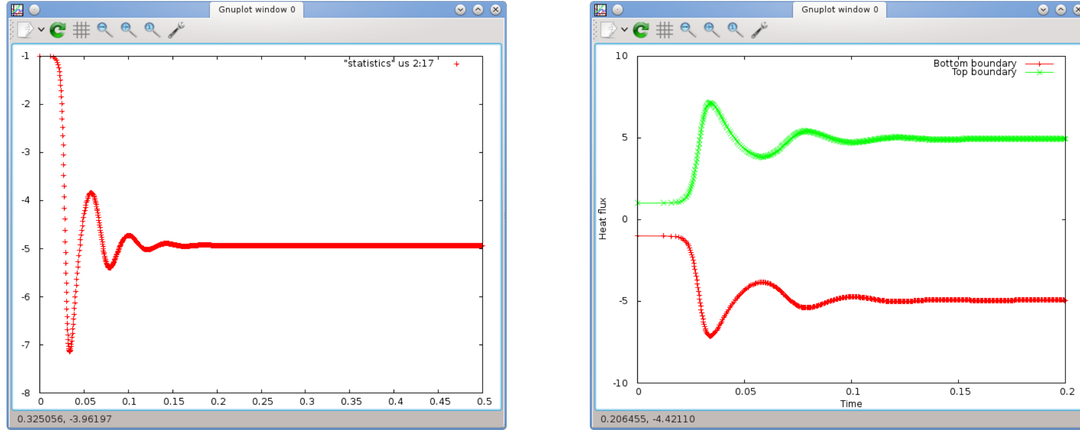


Figure 5: *Visualizing the statistics file obtained from the example in Section 6.2.1 using* **Gnuplot**: *Output using simple commands.*

```
set terminal png
set output "heatflux.png"
replot
```

The last command will simply generate the same plot again but this time into the given file. The result is a graphics file similar to the one shown in Fig. 7 on page 176.

> **Note:** After setting output to a file, *all* following plot commands will want to write to this file. Thus, if you want to create more plots after the one just created, you need to reset output back to the screen. On Linux, this is done using the command `set terminal X11`. You can then continue experimenting with plots and when you have the next plot ready, switch back to output to a file.

What makes `Gnuplot` so useful is that it doesn't just allow entering all these commands at the prompt. Rather, one can write them all into a file, say `plot-heatflux.gnuplot`, and then, on the command line, call

```
gnuplot plot-heatflux.gnuplot
```

to generate the `heatflux.png` file. This comes in handy if one wants to create the same plot for multiple simulations while playing with parameters of the physical setup. It is also a very useful tool if one wants to generate the same kind of plot again later with a different data set, for example when a reviewer requested additional computations to be made for a paper or if one realizes that one has forgotten or misspelled an axis label in a plot.[14]

`Gnuplot` has many many more features we have not even touched upon. For example, it is equally happy to produce three-dimensional graphics, and it also has statistics modules that can do things like curve fits, statistical regression, and many more operations on the data you provide in the columns of an input file. We will not try to cover them here but instead refer to the manual at `http://www.gnuplot.info/`. You can also get a good amount of information by typing `help` at the prompt, or a command like `help plot` to get help on the `plot` command.

[14] In my own work, I usually save the ASPECT input file, the `statistics` output file and the `Gnuplot` script along with the actual figure I want to include in a paper. This way, it is easy to either re-run an entire simulation, or just tweak the graphic at a later time. Speaking from experience, you will not believe how often one wants to tweak a figure long after it was first created. In such situations it is outstandingly helpful if one still has both the actual data as well as the script that generated the graphic.

4.4.3 Large data issues for parallel computations

Among the challenges in visualizing the results of parallel computations is dealing with the large amount of data. The first bottleneck this presents is during run-time when ASPECT wants to write the visualization data of a time step to disk. Using the compressed VTU format, ASPECT generates on the order of 10 bytes of output for each degree of freedom in 2d and more in 3d; thus, output of a single time step can run into the range of gigabytes that somehow have to get from compute nodes to disk. This stresses both the cluster interconnect as well as the data storage array.

There are essentially two strategies supported by ASPECT for this scenario:

- If your cluster has a fast interconnect, for example Infiniband, and if your cluster has a fast, distributed file system, then ASPECT can produce output files that are already located in the correct output directory (see the options in Section 5.2) on the global file system. ASPECT uses MPI I/O calls to this end, ensuring that the local machines do not have to access these files using slow NFS-mounted global file systems.
- If your cluster has a slow interconnect, e.g., if it is simply a collection of machines connected via Ethernet, then writing data to a central file server may block the rest of the program for a while. On the other hand, if your machines have fast local storage for temporary file systems, then ASPECT can write data first into such a file and then move it in the background to its final destination while already continuing computations. To select this mode, set the appropriate variables discussed in Section 5.100. Note, however, that this scheme only makes sense if every machine on which MPI processes run has fast local disk space for temporary storage.

Note: An alternative would be if every processor directly writes its own files into the global output directory (possibly in the background), without the intermediate step of the temporary file. In our experience, file servers are quickly overwhelmed when encountering a few hundred machines wanting to open, fill, flush and close their own file via NFS mounted file system calls, sometimes completely blocking the entire cluster environment for extended periods of time.

4.5 Checkpoint/restart support

If you do long runs, especially when using parallel computations, there are a number of reasons to periodically save the state of the program:

- If the program crashes for whatever reason, the entire computation may be lost. A typical reason is that a program has exceeded the requested wallclock time allocated by a batch scheduler on a cluster.
- Most of the time, no realistic initial conditions for strongly convecting flow are available. Consequently, one typically starts with a somewhat artificial state and simply waits for a long while till the convective state enters the phase where it shows its long-term behavior. However, getting there may take a good amount of CPU time and it would be silly to always start from scratch for each different parameter setting. Rather, one would like to start such parameter studies with a saved state that has already passed this initial, unphysical, transient stage.

To this end, ASPECT creates a set of files in the output directory (selected in the parameter file) every N time steps (controlled by the number of steps or wall time as specified in `subsection Checkpointing`, see Section 5.24) in which the entire state of the program is saved so that a simulation can later be continued at this point. The previous checkpoint files will then be deleted. To resume operations from the last saved state, you need to set the `Resume computation` flag in the input parameter file to `true`, see Section 5.2.

Note: It is not imperative that the parameters selected in the input file are exactly the same when resuming a program from a saved state than what they were at the time when this state was saved. For example, one may want to choose a different parametrization of the material law, or add or remove postprocessors that should be run at the end of each time step. Likewise, the end time, the times at which some additional mesh refinement steps should happen, etc., can be different.
Yet, it is clear that some other things can't be changed: For example, the geometry model that was used to generate the coarse mesh and describe the boundary must be the same before and after resuming a computation. Likewise, you can not currently restart a computation with a different number of processors than initially used to checkpoint the simulation. Not all invalid combinations are easy to detect, and ASPECT may not always realize immediate what is going on if you change a setting that can't be changed. However, you will almost invariably get nonsensical results after some time.

4.6 Making ASPECT run faster

When developing ASPECT, we are guided by the principle that the default for all settings should be *safe*. In particular, this means that you should get errors when something goes wrong, the program should not let you choose an input file parameter so that it doesn't make any sense, and we should solve the equations to best ability without cutting corners. The goal is that when you start working with ASPECT that we give you the best answer we can. The downside is that this also makes ASPECT run slower than may be possible. This section describes ways of making ASPECT run faster – assuming that you know what you are doing and are making conscious decisions.

4.6.1 Debug vs. optimized mode

Both DEAL.II and ASPECT by default have a great deal of internal checking to make sure that the code's state is valid. For example, if you write a new postprocessing plugin (see Section 7.1)) in which you need to access the solution vector, then DEAL.II's `Vector` class will make sure that you are only accessing elements of the vector that actually exist and are available on the current machine if this is a parallel computation. We do so because it turns out that by far the most bugs one introduces in programs are of the kind where one tries to do something that obviously doesn't make sense (such as accessing vector element 101 when it only has 100 elements). These kinds of bugs are more frequent than implementing a wrong algorithm, but they are fortunately easy to find if you have a sufficient number of assertions in your code. The downside is that assertions cost run time.

As mentioned above, the default is to have all of these assertions in the code to catch those places where we may otherwise silently access invalid memory locations. However, once you have a plugin running and verified that your input file runs without problems, you can switch off all of these checks by switching from debug to optimized mode. This means re-compiling ASPECT and linking against a version of the DEAL.II library without all of these internal checks. Because this is the first thing you will likely want to do, we have already discussed how to do all of this in Section 4.3.

4.6.2 Adjusting solver tolerances

At the heart of every time step lies the solution of linear systems for the Stokes equations, the temperature field, and possibly for compositional fields. In essence, each of these steps requires us to solve a linear system of the form $Ax = b$ which we do through iterative solvers, i.e., we try to find a sequence of approximations $x^{(k)}$ where $x^{(k)} \to x = A^{-1}b$. This iteration is terminated at iteration k if the approximation is "close enough" to the exact solution. The solvers we use this determine this by testing after every iteration whether the *residual*, $r^{(k)} = A(x - x^{(k)}) = b - Ax^{(k)}$, satisfies $\|r^{(k)}\| \leq \varepsilon \|r^{(0)}\|$ where ε is called the (relative) *tolerance*.

Obviously, the smaller we choose ε, the more accurate the approximation $x^{(k)}$ will be. On the other hand, it will also take more iterations and, consequently, more CPU time to reach the stopping criterion with a

smaller tolerance. The default value of these tolerances are chosen so that the approximation is typically sufficient. You can make ASPECT run faster if you choose these tolerances larger. The parameters you can adjust are all listed in Section 5.2 and are located at the top level of the input file. In particular, the parameters you want to look at are `Linear solver tolerance`, `Temperature solver tolerance` and `Composition solver tolerance`.

All this said, it is important to understand the consequences of choosing tolerances larger. In particular, if you choose tolerances too large, then the difference between the exact solution of a linear system x and the approximation $x^{(k)}$ may become so large that you do not get an accurate output of your model any more. A rule of thumb in choosing tolerances is to start with a small value and then increase the tolerance until you come to a point where the output quantities start to change significantly. This is the point where you will want to stop.

4.6.3 Adjusting solver preconditioner tolerances

To solve the Stokes equations it is necessary to lower the condition number of the Stokes matrix by preconditioning it. In ASPECT a right preconditioner $Y^{-1} = \begin{pmatrix} \widetilde{A^{-1}} & -\widetilde{A^{-1}}B^T\widetilde{S^{-1}} \\ 0 & \widetilde{S^{-1}} \end{pmatrix}$ is used to precondition the system, where $\widetilde{A^{-1}}$ is the approximate inverse of the A block and $\widetilde{S^{-1}}$ is the approximate inverse of the Schur complement matrix. Matrix $\widetilde{A^{-1}}$ and $\widetilde{S^{-1}}$ are calculated through a CG solve, which requires a tolerance to be set. In comparison with the solver tolerances of the previous section, these parameters are relatively safe to use, since they only change the preconditioner, but can speed up or slow down solving the Stokes system considerably.

In practice $\widetilde{A^{-1}}$ takes by far the most time to compute, but is also very important in conditioning the system. The accuracy of the computation of $\widetilde{A^{-1}}$ is controlled by the parameter `Linear solver A block tolerance` which has a default value of $1e-2$. Setting this tolerance to a less strict value will result in more outer iterations, since the preconditioner is not as good, but the amount of time to compute $\widetilde{A^{-1}}$ can drop significantly resulting in a reduced total solve time. The cookbook crustal deformation (Section 6.3.6) for example can be computed much faster by setting the `Linear solver A block tolerance` to $5e-1$. The calculation of $\widetilde{S^{-1}}$ is usually much faster and the conditioning of the system is less sensitive to the parameter `Linear solver S block tolerance`, but for some problems it might be worth it to investigate.

4.6.4 Using lower order elements for the temperature/compositional discretization

The default settings of ASPECT use quadratic finite elements for the velocity. Given that the temperature and compositional fields essentially (up to material parameters) satisfy advection equations of the kind $\partial_t T + \mathbf{u}\cdot\nabla T = \ldots$, it seems appropriate to also use quadratic finite elemen shape functions for the temperature and compositional fields.

However, this is not mandatory. If you do not care about high accuracy in these fields and are mostly interested in the velocity or pressure field, you can select lower-order finite elements in the input file. The polynomial degrees are controlled with the parameters in the *discretization* section of the input file, see Section 5.29, in particular by `Temperature polynomial degree` and `Composition polynomial degree`.

As with the other parameters discussed above and below, it is worthwhile comparing the results you get with different values of these parameters when making a decision whether you want to save on accuracy in order to reduce compute time. An example of how this choice affects the accuracy you get is discussed in Section 6.2.1.

4.6.5 Limiting postprocessing

ASPECT has a lot of postprocessing capabilities, from generating graphical output to computing average temperatures or temperature fluxes. To see what all is possible, take a look at the `List of postprocessors` parameter that can be set in the input file, see Section 5.88.

Many of these postprocessors take a non-negligible amount of time. How much they collectively use can be inferred from the timing report ASPECT prints periodically among its output, see for example the output shown in Section 6.2.1. So, if your computations take too long, consider limiting which postprocessors you run to those you really need. Some postprocessors – for example those that generate graphical output, see Section 5.100 – also allow you to run them only once every once in a while, rather than at every time step.

4.6.6 Switching off pressure normalization

In most practically relevant cases, the Stokes equations (1)–(2) only determine the pressure up to a constant because only the pressure gradient appears in the equations, not the actual value of it. However, unlike this "mathematical" pressure, we have a very specific notion of the "physical" pressure: namely a well-defined quantity that at the surface of Earth equals the air pressure, which compared to the hydrostatic pressure inside Earth is essentially zero.

As a consequence, the default in ASPECT is to normalize the computed "mathematical" pressure in such a way that either the mean pressure at the surface is zero (where the geometry model describes where the "surface" is, see Section 7.3.3), or that the mean pressure in the domain is zero. This normalization is important if your model describes densities, viscosities and other quantities in dependence of the pressure – because you almost certainly had the "physical" pressure in mind, not some unspecified "mathematical" one. On the other hand, if you have a material model in which the pressure does not enter, then you don't need to normalize the pressure at all – simply go with whatever the solver provides. In that case, you can switch off pressure normalization by looking at the `Pressure normalization` parameter at the top level of the input file, see Section 5.2.

4.6.7 Regularizing models with large coefficient variation

Models with large jumps in viscosity and other coefficients present significant challenges to both discretizations and solvers. In particular, they can lead to very long solver times. Section 6.2.8 presents parameters that can help regularize models and these typically also include significant improvements in run-time.

5 Run-time input parameters

5.1 Overview

What ASPECT computes is driven by two things:

- The models implemented in ASPECT. This includes the geometries, the material laws, or the initial conditions currently supported. Which of these models are currently implemented is discussed below; Section 7 discusses in great detail the process of implementing additional models.

- Which of the implemented models is selected, and what their run-time parameters are. For example, you could select a model that prescribes constant coefficients throughout the domain from all the material models currently implemented; you could then select appropriate values for all of these constants. Both of these selections happen from a parameter file that is read at run time and whose name is specified on the command line. (See also Section 4.1.)

In this section, let us give an overview of what can be selected in the parameter file. Specific parameters, their default values, and allowed values for these parameters are documented in the following subsections. An index with page numbers for all run-time parameters can be found on page 309.

5.1.1 The structure of parameter files

Most of the run-time behavior of ASPECT is driven by a parameter file that looks in essence like this:

```
set Dimension                          = 2
set Resume computation                 = false
set End time                           = 1e10
set CFL number                         = 1.0
set Output directory                   = bin

subsection Mesh refinement
  set Initial adaptive refinement = 1
  set Initial global refinement   = 4
end

subsection Material model
  set Model name                       = simple

  subsection Simple model
    set Reference density              = 3300
    set Reference temperature          = 293
    set Viscosity                      = 5e24
  end
end

...
```

Some parameters live at the top level, but most parameters are grouped into subsections. An input parameter file is therefore much like a file system: a few files live in the root directory; others are in a nested hierarchy of sub-directories. And just as with files, parameters have both a name (the thing to the left of the equals sign) and a content (what's to the right).

All parameters you can list in this input file have been *declared* in ASPECT. What this means is that you can't just list anything in the input file, and expect that entries that are unknown are simply ignored. Rather, if your input file contains a line setting a parameter that is unknown, you will get an error message. Likewise, all declared parameters have a description of possible values associated with them – for example, some parameters must be non-negative integers (the number of initial refinement steps), can either be true or false (whether the computation should be resumed from a saved state), or can only be a single element from a selection (the name of the material model). If an entry in your input file doesn't satisfy these constraints, it will be rejected at the time of reading the file (and not when a part of the program actually accesses the value and the programmer has taken the time to also implement some error checking at this location). Finally, because parameters have been declared, you do not *need* to specify a parameter in the input file: if a parameter isn't listed, then the program will simply use the default provided when declaring the parameter.

> **Note:** In cases where a parameter requires a significant amount of text, you can end a line in the input file with a backslash. This indicates that the following line will simply continue to be part of the text of the current line, in the same way as the C/C++ preprocessor expands lines that end in backslashes.

5.1.2 Categories of parameters

The parameters that can be provided in the input file can roughly be categorized into the following groups:

- Global parameters (see Section 5.2): These parameters determine the overall behavior of the program. Primarily they describe things like the output directory, the end time of the simulation, or whether the computation should be resumed from a previously saved state.
- Parameters for certain aspects of the numerical algorithm: These describe, for example, the specifics of the spatial discretization. In particular, this is the case for parameters concerning the polynomial degree

of the finite element approximation (Section 5.29), some details about the stabilization (Section 5.30), and how adaptive mesh refinement is supposed to work (Section 5.82).

- Parameters that describe certain global aspects of the equations to be solved: This includes, for example, a description if certain terms in the model should be omitted or not. See Section 5.87 for the list of parameters in this category.
- Parameters that characterize plugins: Certain behaviors of ASPECT are described by what we call *plugins* – self-contained parts of the code that describe one particular aspect of the simulation. An example would be which of the implemented material models to use, and the specifics of this material model. The sample parameter file above gives an indication of how this works: within a subsection of the file that pertains to the material models, one can select one out of several plugins (or, in the case of the postprocessors, any number, including none, of the available plugins), and one can then specify the specifics of this model in a sub-subsection dedicated to this particular model.

  A number of components of ASPECT are implemented via plugins. Some of these, together with the sections in which their parameters are declared, are the following:

  - The material model: Sections 5.66 and following.
  - The geometry: Sections 5.32 and following.
  - The gravity description: Sections 5.39 and following.
  - Initial conditions for the temperature: Sections 5.51 and following.
  - Temperature boundary conditions: Sections 5.10 and following.
  - Postprocessors: Sections 5.88 and following for most postprocessors, section 5.100 and following for postprocessors related to visualization.

The details of parameters in each of these categories can be found in the sections linked to above. Some of them will also be used in the cookbooks in Section 6.

5.1.3 A note on the syntax of formulas in input files

Input files have different ways of describing certain things to ASPECT. For example, you could select a plugin for the temperature initial values that prescribes a constant temperature, or a plugin that implements a particular formula for these initial conditions in C++ in the code of the plugin, or a plugin that allows you to describe this formula in a symbolic way in the input file (see Section 5.51). An example of this latter case is this snippet of code discussed in Section 6.2.2:

```
subsection Initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,y,z
    set Function constants  = p=0.01, L=1, pi=3.1415926536, k=1
    set Function expression = (1.0-z) - p*cos(k*pi*x/L)*sin(pi*z)*y^3
  end
end
```

The formulas you can enter here need to use a syntax that is understood by the functions and classes that interpret what you write. Internally, this is done using the muparser library, see `http://muparser.beltoforion.de/`. The syntax is mostly self-explanatory in that it allows to use the usual symbols `x`, `y` and `z` to reference coordinates (unless a particular plugin uses different variables, such as the depth), the symbol `t` for time in many situations, and allows you to use all of the typical mathematical functions such as sine and cosine. Another common case is an if-statement that has the general form `if(condition,true-expression,false-expression)`. For more examples of the syntax understood, reference the documentation of the muparser library linked to above.

5.2 Global parameters

- *Parameter name:* `Additional shared libraries`

  Value:

  Default:

  Description: A list of names of additional shared libraries that should be loaded upon starting up the program. The names of these files can contain absolute or relative paths (relative to the directory in which you call ASPECT). In fact, file names that are do not contain any directory information (i.e., only the name of a file such as <myplugin.so> will not be found if they are not located in one of the directories listed in the `LD_LIBRARY_PATH` environment variable. In order to load a library in the current directory, use <./myplugin.so> instead.

  The typical use of this parameter is so that you can implement additional plugins in your own directories, rather than in the ASPECT source directories. You can then simply compile these plugins into a shared library without having to re-compile all of ASPECT. See the section of the manual discussing writing extensions for more information on how to compile additional files into a shared library.

  Possible values: [List list of [FileName (Type: input)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Adiabatic surface temperature`

  Value: 0

  Default: 0

  Description: In order to make the problem in the first time step easier to solve, we need a reasonable guess for the temperature and pressure. To obtain it, we use an adiabatic pressure and temperature field. This parameter describes what the 'adiabatic' temperature would be at the surface of the domain (i.e. at depth zero). Note that this value need not coincide with the boundary condition posed at this point. Rather, the boundary condition may differ significantly from the adiabatic value, and then typically induce a thermal boundary layer.

  For more information, see the section in the manual that discusses the general mathematical model.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `CFL number`

  Value: 1.0

  Default: 1.0

  Description: In computations, the time step k is chosen according to $k = c \min_K \frac{h_K}{\|u\|_{\infty,K} p_T}$ where h_K is the diameter of cell K, and the denominator is the maximal magnitude of the velocity on cell K times the polynomial degree p_T of the temperature discretization. The dimensionless constant c is called the CFL number in this program. For time discretizations that have explicit components, c must be less than a constant that depends on the details of the time discretization and that is no larger than one. On the other hand, for implicit discretizations such as the one chosen here, one can choose the time step as large as one wants (in particular, one can choose $c > 1$) though a CFL number significantly larger than one will yield rather diffusive solutions. Units: None.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Composition solver tolerance`

  Value: 1e-12

  Default: 1e-12

  Description: The relative tolerance up to which the linear system for the composition system gets solved. See 'linear solver tolerance' for more details.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Dimension`

  Value: 2

  Default: 2

  Description: The number of space dimensions you want to run this program in. ASPECT can run in 2 and 3 space dimensions.

  Possible values: [Integer range 2...4 (inclusive)]

- *Parameter name:* `End time`

  Value: 5.69e+300

  Default: 5.69e+300

  Description: The end time of the simulation. The default value is a number so that when converted from years to seconds it is approximately equal to the largest number representable in floating point arithmetic. For all practical purposes, this equals infinity. Units: Years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Linear solver A block tolerance`

  Value: 1e-2

  Default: 1e-2

  Description: A relative tolerance up to which the approximate inverse of the A block of the Stokes system is computed. This approximate A is used in the preconditioning used in the GMRES solver.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Linear solver S block tolerance`

  Value: 1e-6

  Default: 1e-6

  Description: A relative tolerance up to which the approximate inverse of the S block (Schur complement matrix, $S = BA^{-1}B^T$) of the Stokes system is computed. This approximate inverse of the S block is used in the preconditioning used in the GMRES solver.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Linear solver tolerance`

  Value: 1e-7

  Default: 1e-7

  Description: A relative tolerance up to which the linear Stokes systems in each time or nonlinear step should be solved. The absolute tolerance will then be $\|Mx_0 - F\| \cdot \mathrm{tol}$, where $x_0 = (0, p_0)$ is the initial guess of the pressure, M is the system matrix, F is the right-hand side, and tol is the parameter specified here. We include the initial guess of the pressure to remove the dependency of the tolerance on the static pressure. A given tolerance value of 1 would mean that a zero solution vector is an acceptable solution since in that case the norm of the residual of the linear system equals the norm of the right hand side. A given tolerance of 0 would mean that the linear system has to be solved exactly, since this is the only way to obtain a zero residual.

  In practice, you should choose the value of this parameter to be so that if you make it smaller the results of your simulation do not change any more (qualitatively) whereas if you make it larger, they do. For most cases, the default value should be sufficient. In fact, a tolerance of 1e-4 might be accurate enough.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Max nonlinear iterations`

  Value: 10

  Default: 10

  Description: The maximal number of nonlinear iterations to be performed.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Max nonlinear iterations in pre-refinement`

  Value: 2147483647

  Default: 2147483647

  Description: The maximal number of nonlinear iterations to be performed in the pre-refinement steps. This does not include the last refinement step before moving to timestep 1. When this parameter has a larger value than max nonlinear iterations, the latter is used.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Maximum time step`

  Value: 5.69e+300

  Default: 5.69e+300

  Description: Set a maximum time step size for the solver to use. Generally the time step based on the CFL number should be sufficient, but for complicated models or benchmarking it may be useful to limit the time step to some value. The default value is a value so that when converted from years into seconds it equals the largest number representable by a floating point number, implying an unlimited time step.Units: Years or seconds, depending on the "Use years in output instead of seconds" parameter.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Nonlinear solver scheme`

  Value: IMPES

  Default: IMPES

  Description: The kind of scheme used to resolve the nonlinearity in the system. 'IMPES' is the classical IMplicit Pressure Explicit Saturation scheme in which ones solves the temperatures and Stokes equations exactly once per time step, one after the other. The 'iterated IMPES' scheme iterates this decoupled approach by alternating the solution of the temperature and Stokes systems. The 'iterated Stokes' scheme solves the temperature equation once at the beginning of each time step and then iterates out the solution of the Stokes equation. The 'Stokes only' scheme only solves the Stokes system and ignores compositions and the temperature equation (careful, the material model must not depend on the temperature; mostly useful for Stokes benchmarks). The 'Advection only'scheme only solves the temperature and other advection systems and instead of solving for the Stokes system, a prescribed velocity and pressure is used

  Possible values: [Selection IMPES—iterated IMPES—iterated Stokes—Stokes only—Advection only]

- *Parameter name:* `Nonlinear solver tolerance`

  Value: 1e-5

  Default: 1e-5

  Description: A relative tolerance up to which the nonlinear solver will iterate. This parameter is only relevant if Nonlinear solver scheme is set to 'iterated Stokes' or 'iterated IMPES'.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Number of cheap Stokes solver steps`

  Value: 30

  Default: 30

  Description: As explained in the ASPECT paper (Kronbichler, Heister, and Bangerth, GJI 2012) we first try to solve the Stokes system in every time step using a GMRES iteration with a poor but cheap preconditioner. By default, we try whether we can converge the GMRES solver in 30 such iterations before deciding that we need a better preconditioner. This is sufficient for simple problems with constant viscosity and we never need the second phase with the more expensive preconditioner. On the other hand, for more complex problems, and in particular for problems with strongly varying viscosity, the 30 cheap iterations don't actually do very much good and one might skip this part right away. In that case, this parameter can be set to zero, i.e., we immediately start with the better but more expensive preconditioner.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Output directory`

  Value: output

  Default: output

  Description: The name of the directory into which all output files should be placed. This may be an absolute or a relative path.

  Possible values: [DirectoryName]

- *Parameter name:* `Pressure normalization`

  Value: surface

  Default: surface

  Description: If and how to normalize the pressure after the solution step. This is necessary because depending on boundary conditions, in many cases the pressure is only determined by the model up to a constant. On the other hand, we often would like to have a well-determined pressure, for example for table lookups of material properties in models or for comparing solutions. If the given value is 'surface', then normalization at the end of each time steps adds a constant value to the pressure in such a way that the average pressure at the surface of the domain is zero; the surface of the domain is determined by asking the geometry model whether a particular face of the geometry has a zero or small 'depth'. If the value of this parameter is 'volume' then the pressure is normalized so that the domain average is zero. If 'no' is given, the no pressure normalization is performed.

  Possible values: [Selection surface—volume—no]

- *Parameter name:* `Resume computation`

  Value: false

  Default: false

  Description: A flag indicating whether the computation should be resumed from a previously saved state (if true) or start from scratch (if false). If auto is selected, models will be resumed if there is an existing checkpoint file, otherwise started from scratch.

  Possible values: [Selection true—false—auto]

- *Parameter name:* `Start time`

  Value: 0

  Default: 0

Description: The start time of the simulation. Units: Years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Surface pressure`

  Value: 0

  Default: 0

  Description: The mathematical equations that describe thermal convection only determine the pressure up to an arbitrary constant. On the other hand, for comparison and for looking up material parameters it is important that the pressure be normalized somehow. We do this by enforcing a particular average pressure value at the surface of the domain, where the geometry model determines where the surface is. This parameter describes what this average surface pressure value is supposed to be. By default, it is set to zero, but one may want to choose a different value for example for simulating only the volume of the mantle below the lithosphere, in which case the surface pressure should be the lithostatic pressure at the bottom of the lithosphere.

  For more information, see the section in the manual that discusses the general mathematical model.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Temperature solver tolerance`

  Value: 1e-12

  Default: 1e-12

  Description: The relative tolerance up to which the linear system for the temperature system gets solved. See 'linear solver tolerance' for more details.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Timing output frequency`

  Value: 100

  Default: 100

  Description: How frequently in timesteps to output timing information. This is generally adjusted only for debugging and timing purposes. If the value is set to zero it will also output timing information at the initiation timesteps.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Use conduction timestep`

  Value: false

  Default: false

  Description: Mantle convection simulations are often focused on convection dominated systems. However, these codes can also be used to investigate systems where heat conduction plays a dominant role. This parameter indicates whether the simulator should also use heat conduction in determining the length of each time step.

  Possible values: [Bool]

- *Parameter name:* `Use direct solver for Stokes system`

  Value: false

  Default: false

Description: If set to true the linear system for the Stokes equation will be solved using Trilinos klu, otherwise an iterative Schur complement solver is used. The direct solver is only efficient for small problems.

Possible values: [Bool]

- *Parameter name:* `Use years in output instead of seconds`

  Value: true

  Default: true

  Description: When computing results for mantle convection simulations, it is often difficult to judge the order of magnitude of results when they are stated in MKS units involving seconds. Rather, some kinds of results such as velocities are often stated in terms of meters per year (or, sometimes, centimeters per year). On the other hand, for non-dimensional computations, one wants results in their natural unit system as used inside the code. If this flag is set to 'true' conversion to years happens; if it is 'false', no such conversion happens. Note that when 'true', some input such as prescribed velocities should also use years instead of seconds.

  Possible values: [Bool]

5.3 Parameters in section `Adiabatic conditions model`

- *Parameter name:* `Model name`

  Value: initial profile

  Default: initial profile

  Description: Select one of the following models:

  'initial profile': A model in which the adiabatic profile is calculated once at the start of the model run. The gravity is assumed to be in depth direction and the composition is evaluated at reference points, no lateral averaging is performed. All material parameters are used from the material model plugin.

  Possible values: [Selection initial profile]

5.4 Parameters in section `Boundary composition model`

- *Parameter name:* `Model name`

  Value: unspecified

  Default: unspecified

  Description: Select one of the following models:

  'ascii data': Implementation of a model in which the boundary composition is derived from files containing data in ascii format. Note the required format of the input data: The first lines may contain any number of comments if they begin with '#', but one of these lines needs to contain the number of grid points in each dimension as for example '# POINTS: 3 3'. The order of the data columns has to be 'x', 'composition1', 'composition2', etc. in a 2d model and 'x', 'y', 'composition1', 'composition2', etc., in a 3d model, according to the number of compositional fields, which means that there has to be a single column for every composition in the model. Note that the data in the input files need to be sorted in a specific order: the first coordinate needs to ascend first, followed by the second in order to assign the correct data to the prescribed coordinates.If you use a spherical model, then the data will still be handled as Cartesian, however the assumed grid changes. 'x' will be replaced by the radial distance of the point to the bottom of the model, 'y' by the azimuth angle and 'z' by the polar angle measured positive from the north pole. The grid will be assumed to be a latitude-longitude grid. Note that the order of spherical coordinates is 'r', 'phi', 'theta' and not 'r', 'theta', 'phi', since this allows for dimension independent expressions.

‘box’: A model in which the composition is chosen constant on all the sides of a box.

‘box with lithosphere boundary indicators’: A model in which the composition is chosen constant on all the sides of a box. Additional boundary indicators are added to the lithospheric parts of the vertical boundaries. This model is to be used with the 'Two Merged Boxes' Geometry Model.

‘initial composition’: A model in which the composition at the boundary is chosen to be the same as given in the initial conditions.

Because this class simply takes what the initial composition had described, this class can not know certain pieces of information such as the minimal and maximal composition on the boundary. For operations that require this, for example in postprocessing, this boundary composition model must therefore be told what the minimal and maximal values on the boundary are. This is done using parameters set in section “Boundary composition model/Initial composition”.

‘spherical constant’: A model in which the composition is chosen constant on the inner and outer boundaries of a spherical shell or chunk. Parameters are read from subsection 'Spherical constant'.

Possible values: [Selection ascii data—box—box with lithosphere boundary indicators—initial composition—spherical constant—unspecified]

5.5 Parameters in section `Boundary composition model/Ascii data model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/boundary-composition/ascii-data/test/

  Default: $ASPECT_SOURCE_DIR/data/boundary-composition/ascii-data/test/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: box_2d_%s.%d.txt

  Default: box_2d_%s.%d.txt

  Description: The file name of the material data. Provide file in format: (Velocity file name).%s%d where %s is a string specifying the boundary of the model according to the names of the boundary indicators (of a box or a spherical shell).%d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

- *Parameter name:* `Data file time step`

  Value: 1e6

  Default: 1e6

  Description: Time step between following velocity files. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years. The default is one million, i.e., either one million seconds or one million years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Decreasing file order`

  Value: false

  Default: false

  Description: In some cases the boundary files are not numbered in increasing but in decreasing order (e.g. 'Ma BP'). If this flag is set to 'True' the plugin will first load the file with the number 'First velocity file number' and decrease the file number during the model run.

  Possible values: [Bool]

- *Parameter name:* `First data file model time`

  Value: 0

  Default: 0

  Description: Time from which on the velocity file with number 'First velocity file number' is used as boundary condition. Previous to this time, a no-slip boundary condition is assumed. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `First data file number`

  Value: 0

  Default: 0

  Description: Number of the first velocity file to be loaded when the model time is larger than 'First velocity file model time'.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

  Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction. Another way to use this factor is to convert units of the input files. The unit is assumed to bem/s or m/yr depending on the 'Use years in output instead of seconds' flag. If you provide velocities in cm/yr set this factor to 0.01.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.6 Parameters in section `Boundary composition model/Box`

- *Parameter name:* `Bottom composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the bottom boundary (at minimal y-value in 2d, or minimal z-value in 3d). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Left composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the left boundary (at minimal x-value). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Right composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the right boundary (at maximal x-value). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Top composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the top boundary (at maximal y-value in 2d, or maximal z-value in 3d). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.7 Parameters in section `Boundary composition model/Box with lithosphere boundary indicators`

- *Parameter name:* `Bottom composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the bottom boundary (at minimal y-value in 2d, or minimal z-value in 3d). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Left composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the left boundary (at minimal x-value). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Left composition lithosphere`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the left boundary (at minimal x-value). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Right composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the right boundary (at maximal x-value). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Right composition lithosphere`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the right boundary (at maximal x-value). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Top composition`

  Value:

  Default:

  Description: A comma separated list of composition boundary values at the top boundary (at maximal y-value in 2d, or maximal z-value in 3d). This list must have as many entries as there are compositional fields. Units: none.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.8 Parameters in section `Boundary composition model/Initial composition`

- *Parameter name:* `Maximal composition`

  Value: 1

  Default: 1

  Description: Maximal composition. Units: none.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimal composition`

  Value: 0

  Default: 0

  Description: Minimal composition. Units: none.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.9 Parameters in section `Boundary composition model/Spherical constant`

- *Parameter name:* `Inner composition`

  Value: 1

  Default: 1

  Description: Composition at the inner boundary (core mantle boundary). Units: none.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Outer composition`

  Value: 0

  Default: 0

  Description: Composition at the outer boundary (lithosphere water/air). Units: none.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.10 Parameters in section `Boundary temperature model`

- *Parameter name:* `Model name`

  Value: box

  Default: unspecified

  Description: Select one of the following models:

  'ascii data': Implementation of a model in which the boundary data is derived from files containing data in ascii format. Note the required format of the input data: The first lines may contain any number of comments if they begin with '#', but one of these lines needs to contain the number of grid points in each dimension as for example '# POINTS: 3 3'. The order of the data columns has to be 'x', 'Temperature [K]' in a 2d model and 'x', 'y', 'Temperature [K]' in a 3d model, which means that there has to be a single column containing the temperature. Note that the data in the input files need to be sorted in a specific order: the first coordinate needs to ascend first, followed by the second in order to assign the correct data to the prescribed coordinates. If you use a spherical model, then the data will still be handled as Cartesian, however the assumed grid changes. 'x' will be replaced by the radial distance of the point to the bottom of the model, 'y' by the azimuth angle and 'z' by the polar angle measured positive from the north pole. The grid will be assumed to be a latitude-longitude grid. Note that the order of spherical coordinates is 'r', 'phi', 'theta' and not 'r', 'theta', 'phi', since this allows for dimension independent expressions.

  'box': A model in which the temperature is chosen constant on all the sides of a box.

  'box with lithosphere boundary indicators': A model in which the temperature is chosen constant on all the sides of a box. Additional boundary indicators are added to the lithospheric parts of the vertical boundaries. This model is to be used with the 'Two Merged Boxes' Geometry Model.

  'constant': A model in which the temperature is chosen constant on a given boundary indicator. Parameters are read from the subsection 'Constant'.

  'function': Implementation of a model in which the boundary temperature is given in terms of an explicit formula that is elaborated in the parameters in section "Boundary temperature model—Function".

  Since the symbol t indicating time may appear in the formulas for the prescribed temperatures, it is interpreted as having units seconds unless the global input parameter "Use years in output instead of seconds" is set, in which case we interpret the formula expressions as having units year.

  Because this class simply takes what the function calculates, this class can not know certain pieces of information such as the minimal and maximal temperature on the boundary. For operations that require this, for example in postprocessing, this boundary temperature model must therefore be told

what the minimal and maximal values on the boundary are. This is done using parameters set in section "Boundary temperature model/Initial temperature".

The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

'initial temperature': A model in which the temperature at the boundary is chosen to be the same as given in the initial conditions.

Because this class simply takes what the initial temperature had described, this class can not know certain pieces of information such as the minimal and maximal temperature on the boundary. For operations that require this, for example in postprocessing, this boundary temperature model must therefore be told what the minimal and maximal values on the boundary are. This is done using parameters set in section "Boundary temperature model/Initial temperature".

'spherical constant': A model in which the temperature is chosen constant on the inner and outer boundaries of a spherical shell, ellipsoidal chunk or chunk. Parameters are read from subsection 'Spherical constant'.

Possible values: [Selection ascii data—box—box with lithosphere boundary indicators—constant—function—initial temperature—spherical constant—unspecified]

5.11 Parameters in section `Boundary temperature model/Ascii data model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/boundary-temperature/ascii-data/test/

  Default: $ASPECT_SOURCE_DIR/data/boundary-temperature/ascii-data/test/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: box_2d_%s.%d.txt

  Default: box_2d_%s.%d.txt

  Description: The file name of the material data. Provide file in format: (Velocity file name).%s%d where %s is a string specifying the boundary of the model according to the names of the boundary indicators (of a box or a spherical shell).%d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

- *Parameter name:* `Data file time step`

  Value: 1e6

  Default: 1e6

  Description: Time step between following velocity files. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years. The default is one million, i.e., either one million seconds or one million years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Decreasing file order`

  Value: false

  Default: false

  Description: In some cases the boundary files are not numbered in increasing but in decreasing order (e.g. 'Ma BP'). If this flag is set to 'True' the plugin will first load the file with the number 'First velocity file number' and decrease the file number during the model run.

  Possible values: [Bool]

- *Parameter name:* `First data file model time`

  Value: 0

  Default: 0

  Description: Time from which on the velocity file with number 'First velocity file number' is used as boundary condition. Previous to this time, a no-slip boundary condition is assumed. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `First data file number`

  Value: 0

  Default: 0

  Description: Number of the first velocity file to be loaded when the model time is larger than 'First velocity file model time'.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

  Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction. Another way to use this factor is to convert units of the input files. The unit is assumed to bem/s or m/yr depending on the 'Use years in output instead of seconds' flag. If you provide velocities in cm/yr set this factor to 0.01.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.12 Parameters in section `Boundary temperature model/Box`

- *Parameter name:* `Bottom temperature`

  Value: 0

  Default: 0

  Description: Temperature at the bottom boundary (at minimal z-value). Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Left temperature`

  Value: 1

  Default: 1

  Description: Temperature at the left boundary (at minimal x-value). Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Right temperature`
  Value: 0
  Default: 0
  Description: Temperature at the right boundary (at maximal x-value). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]
- *Parameter name:* `Top temperature`
  Value: 0
  Default: 0
  Description: Temperature at the top boundary (at maximal x-value). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.13 Parameters in section `Boundary temperature model/Box with lithosphere boundary indicators`

- *Parameter name:* `Bottom temperature`
  Value: 0
  Default: 0
  Description: Temperature at the bottom boundary (at minimal z-value). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]
- *Parameter name:* `Left temperature`
  Value: 1
  Default: 1
  Description: Temperature at the left boundary (at minimal x-value). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]
- *Parameter name:* `Left temperature lithosphere`
  Value: 0
  Default: 0
  Description: Temperature at the additional left lithosphere boundary (specified by user in Geometry Model). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]
- *Parameter name:* `Right temperature`
  Value: 0
  Default: 0
  Description: Temperature at the right boundary (at maximal x-value). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]
- *Parameter name:* `Right temperature lithosphere`
  Value: 0
  Default: 0
  Description: Temperature at the additional right lithosphere boundary (specified by user in Geometry Model). Units: K.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Top temperature`

  Value: 0

  Default: 0

  Description: Temperature at the top boundary (at maximal x-value). Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.14 Parameters in section `Boundary temperature model/Constant`

- *Parameter name:* `Boundary indicator to temperature mappings`

  Value:

  Default:

  Description: A comma separated list of mappings between boundary indicators and the temperature associated with the boundary indicators. The format for this list is "indicator1 : value1, indicator2 : value2, ...", where each indicator is a valid boundary indicator (either a number or the symbolic name of a boundary as provided by the geometry model) and each value is the temperature of that boundary.

  Possible values: [Map map of [Anything]:[Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.15 Parameters in section `Boundary temperature model/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Maximal temperature`

  Value: 3773

  Default: 3773

  Description: Maximal temperature. Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimal temperature`

  Value: 273

  Default: 273

  Description: Minimal temperature. Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.16 Parameters in section `Boundary temperature model/Initial temperature`

- *Parameter name:* `Maximal temperature`

  Value: 3773

  Default: 3773

  Description: Maximal temperature. Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimal temperature`

  Value: 0

  Default: 0

  Description: Minimal temperature. Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.17 Parameters in section `Boundary temperature model/Spherical constant`

- *Parameter name:* `Inner temperature`

  Value: 6000

  Default: 6000

  Description: Temperature at the inner boundary (core mantle boundary). Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Outer temperature`

  Value: 0

  Default: 0

  Description: Temperature at the outer boundary (lithosphere water/air). Units: K.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.18 Parameters in section `Boundary traction model`

5.19 Parameters in section `Boundary traction model/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0; 0

  Default: 0; 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.20 Parameters in section `Boundary velocity model`

5.21 Parameters in section `Boundary velocity model/Ascii data model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/velocity-boundary-conditions/ascii-data/test/

  Default: $ASPECT_SOURCE_DIR/data/velocity-boundary-conditions/ascii-data/test/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: box_2d_%s.%d.txt

  Default: box_2d_%s.%d.txt

  Description: The file name of the material data. Provide file in format: (Velocity file name).%s%d where %s is a string specifying the boundary of the model according to the names of the boundary indicators (of a box or a spherical shell).%d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

- *Parameter name:* `Data file time step`

  Value: 1e6

  Default: 1e6

  Description: Time step between following velocity files. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years. The default is one million, i.e., either one million seconds or one million years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Decreasing file order`

  Value: false

  Default: false

  Description: In some cases the boundary files are not numbered in increasing but in decreasing order (e.g. 'Ma BP'). If this flag is set to 'True' the plugin will first load the file with the number 'First velocity file number' and decrease the file number during the model run.

  Possible values: [Bool]

- *Parameter name:* `First data file model time`

  Value: 0

  Default: 0

  Description: Time from which on the velocity file with number 'First velocity file number' is used as boundary condition. Previous to this time, a no-slip boundary condition is assumed. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `First data file number`

  Value: 0

  Default: 0

  Description: Number of the first velocity file to be loaded when the model time is larger than 'First velocity file model time'.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

  Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction. Another way to use this factor is to convert units of the input files. The unit is assumed to bem/s or m/yr depending on the 'Use years in output instead of seconds' flag. If you provide velocities in cm/yr set this factor to 0.01.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.22 Parameters in section `Boundary velocity model/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0; 0

  Default: 0; 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

Possible values: [Anything]

5.23 Parameters in section `Boundary velocity model/GPlates model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/velocity-boundary-conditions/gplates/

  Default: $ASPECT_SOURCE_DIR/data/velocity-boundary-conditions/gplates/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file time step`

  Value: 1e6

  Default: 1e6

  Description: Time step between following velocity files. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years. The default is one million, i.e., either one million seconds or one million years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Decreasing file order`

  Value: false

  Default: false

  Description: In some cases the boundary files are not numbered in increasing but in decreasing order (e.g. 'Ma BP'). If this flag is set to 'True' the plugin will first load the file with the number 'First velocity file number' and decrease the file number during the model run.

  Possible values: [Bool]

- *Parameter name:* `First data file model time`

  Value: 0

  Default: 0

  Description: Time from which on the velocity file with number 'First velocity file number' is used as boundary condition. Previous to this time, a no-slip boundary condition is assumed. Depending on the setting of the global 'Use years in output instead of seconds' flag in the input file, this number is either interpreted as seconds or as years.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `First data file number`

  Value: 0

  Default: 0

  Description: Number of the first velocity file to be loaded when the model time is larger than 'First velocity file model time'.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Lithosphere thickness`

  Value: 100000

  Default: 100000

  Description: Determines the depth of the lithosphere, so that the GPlates velocities can be applied at the sides of the model as well as at the surface.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Point one`

  Value: 1.570796,0.0

  Default: 1.570796,0.0

  Description: Point that determines the plane in which a 2D model lies in. Has to be in the format 'a,b' where a and b are theta (polar angle) and phi in radians.

  Possible values: [Anything]

- *Parameter name:* `Point two`

  Value: 1.570796,1.570796

  Default: 1.570796,1.570796

  Description: Point that determines the plane in which a 2D model lies in. Has to be in the format 'a,b' where a and b are theta (polar angle) and phi in radians.

  Possible values: [Anything]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

  Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Velocity file name`

  Value: phi.%d

  Default: phi.%d

  Description: The file name of the material data. Provide file in format: (Velocity file name).%d.gpml where %d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

5.24 Parameters in section `Checkpointing`

- *Parameter name:* `Steps between checkpoint`

  Value: 0

  Default: 0

  Description: The number of timesteps between performing checkpoints. If 0 and time between checkpoint is not specified, checkpointing will not be performed. Units: None.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Time between checkpoint`

  Value: 0

  Default: 0

  Description: The wall time between performing checkpoints. If 0, will use the checkpoint step frequency instead. Units: Seconds.

  Possible values: [Integer range 0...2147483647 (inclusive)]

5.25 Parameters in section `Compositional fields`

- *Parameter name:* `List of normalized fields`

  Value:

  Default:

  Description: A list of integers smaller than or equal to the number of compositional fields. All compositional fields in this list will be normalized before the first timestep. The normalization is implemented in the following way: First, the sum of the fields to be normalized is calculated at every point and the global maximum is determined. Second, the compositional fields to be normalized are divided by this maximum.

  Possible values: [List list of [Integer range 0...2147483647 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Names of fields`

  Value:

  Default:

  Description: A user-defined name for each of the compositional fields requested.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Number of fields`

  Value: 0

  Default: 0

  Description: The number of fields that will be advected along with the flow field, excluding velocity, pressure and temperature.

  Possible values: [Integer range 0...2147483647 (inclusive)]

5.26 Parameters in section `Compositional initial conditions`

- *Parameter name:* `Model name`

  Value: function

  Default: function

  Description: Select one of the following models:

  'ascii data': Implementation of a model in which the initial composition is derived from files containing data in ascii format. Note the required format of the input data: The first lines may contain any number of comments if they begin with '#', but one of these lines needs to contain the number of grid points in each dimension as for example '# POINTS: 3 3'. The order of the data columns has to be 'x', 'y', 'composition1', 'composition2', etc. in a 2d model and 'x', 'y', 'z', 'composition1', 'composition2', etc. in a 3d model, according to the number of compositional fields, which means that there has to be a single column for every composition in the model.Note that the data in the input files need to be sorted in a specific order: the first coordinate needs to ascend first, followed by the second and the third at last in order to assign the correct data to the prescribed coordinates. If you use a spherical model, then the data will still be handled as Cartesian, however the assumed grid changes. 'x' will be replaced by the radial distance of the point to the bottom of the model, 'y' by the azimuth angle and 'z' by the polar angle measured positive from the north pole. The grid will be assumed to be a latitude-longitude grid. Note that the order of spherical coordinates is 'r', 'phi', 'theta' and not 'r', 'theta', 'phi', since this allows for dimension independent expressions.

  'function': Specify the composition in terms of an explicit formula. The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

  Possible values: [Selection ascii data—function]

5.27 Parameters in section `Compositional initial conditions/Ascii data model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/compositional-initial-conditions/ascii-data/test/

  Default: $ASPECT_SOURCE_DIR/data/compositional-initial-conditions/ascii-data/test/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: box_2d.txt

  Default: box_2d.txt

  Description: The file name of the material data. Provide file in format: (Velocity file name).%s%d where %s is a string specifying the boundary of the model according to the names of the boundary indicators (of a box or a spherical shell).%d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction. Another way to use this factor is to convert units of the input files. The unit is assumed to bem/s or m/yr depending on the 'Use years in output instead of seconds' flag. If you provide velocities in cm/yr set this factor to 0.01.

Possible values: [Double 0...1.79769e+308 (inclusive)]

5.28 Parameters in section `Compositional initial conditions/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.29 Parameters in section Discretization

- *Parameter name:* `Composition polynomial degree`

  Value: 2

  Default: 2

  Description: The polynomial degree to use for the composition variable(s). Units: None.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Stokes velocity polynomial degree`

  Value: 2

  Default: 2

  Description: The polynomial degree to use for the velocity variables in the Stokes system. The polynomial degree for the pressure variable will then be one less in order to make the velocity/pressure pair conform with the usual LBB (Babuska-Brezzi) condition. In other words, we are using a Taylor-Hood element for the Stoeks equations and this parameter indicates the polynomial degree of it. Units: None.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Temperature polynomial degree`

  Value: 2

  Default: 2

  Description: The polynomial degree to use for the temperature variable. Units: None.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Use discontinuous composition discretization`

  Value: false

  Default: false

  Description: Whether to use a composition discretization that is discontinuous as opposed to continuous. This then requires the assembly of face terms between cells, and weak imposition of boundary terms for the composition field via the discontinuous Galerkin method.

  Possible values: [Bool]

- *Parameter name:* `Use discontinuous temperature discretization`

  Value: false

  Default: false

  Description: Whether to use a temperature discretization that is discontinuous as opposed to continuous. This then requires the assembly of face terms between cells, and weak imposition of boundary terms for the temperature field via the interior-penalty discontinuous Galerkin method.

  Possible values: [Bool]

- *Parameter name:* `Use locally conservative discretization`

  Value: false

  Default: false

  Description: Whether to use a Stokes discretization that is locally conservative at the expense of a larger number of degrees of freedom (true), or to go with a cheaper discretization that does not locally conserve mass, although it is globally conservative (false).

When using a locally conservative discretization, the finite element space for the pressure is discontinuous between cells and is the polynomial space P_{-q} of polynomials of degree q in each variable separately. Here, q is one less than the value given in the parameter "Stokes velocity polynomial degree". As a consequence of choosing this element, it can be shown if the medium is considered incompressible that the computed discrete velocity field $\mathbf{u}_h$ satisfies the property $\int_{\partial K} \mathbf{u}_h \cdot \mathbf{n} = 0$ for every cell K, i.e., for each cell inflow and outflow exactly balance each other as one would expect for an incompressible medium. In other words, the velocity field is locally conservative.

On the other hand, if this parameter is set to "false", then the finite element space is chosen as Q_q. This choice does not yield the local conservation property but has the advantage of requiring fewer degrees of freedom. Furthermore, the error is generally smaller with this choice.

For an in-depth discussion of these issues and a quantitative evaluation of the different choices, see [KHB12] .

Possible values: [Bool]

5.30 Parameters in section `Discretization/Stabilization parameters`

- *Parameter name:* `Discontinuous penalty`

  Value: 10

  Default: 10

  Description: The value used to penalize discontinuities in the discontinuous Galerkin method. This is used only for the temperature field, and not for the composition field, as pure advection does not use the interior penalty method. This is largely empirically decided – it must be large enough to ensure the bilinear form is coercive, but not so large as to penalize discontinuity at all costs.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Use artificial viscosity smoothing`

  Value: false

  Default: false

  Description: If set to false, the artificial viscosity of a cell is computed andis computed on every cell separately as discussed in [KHB12]. If set to true, the maximum of the artificial viscosity in the cell as well as the neighbors of the cell is computed and used instead.

  Possible values: [Bool]

- *Parameter name:* `alpha`

  Value: 2

  Default: 2

  Description: The exponent α in the entropy viscosity stabilization. Valid options are 1 or 2. The recommended setting is 2. (This parameter does not correspond to any variable in the 2012 GJI paper by Kronbichler, Heister and Bangerth that describes ASPECT. Rather, the paper always uses 2 as the exponent in the definition of the entropy, following eq. (15).).Units: None.

  Possible values: [Integer range 1...2 (inclusive)]

- *Parameter name:* `beta`

  Value: 0.078

  Default: 0.078

  Description: The β factor in the artificial viscosity stabilization. An appropriate value for 2d is 0.078 and 0.117 for 3d. (For historical reasons, the name used here is different from the one used in the 2012

GJI paper by Kronbichler, Heister and Bangerth that describes ASPECT. This parameter corresponds to the factor $\alpha_{\max}$ in the formulas following equation (15) of the paper. After further experiments, we have also chosen to use a different value than described there: It can be chosen as stated there for uniformly refined meshes, but it needs to be chosen larger if the mesh has cells that are not squares or cubes.) Units: None.

Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `cR`

  Value: 0.33

  Default: 0.33

  Description: The c_R factor in the entropy viscosity stabilization. (For historical reasons, the name used here is different from the one used in the 2012 GJI paper by Kronbichler, Heister and Bangerth that describes ASPECT. This parameter corresponds to the factor α_E in the formulas following equation (15) of the paper. After further experiments, we have also chosen to use a different value than described there.) Units: None.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.31 Parameters in section `Free surface`

- *Parameter name:* `Additional tangential mesh velocity boundary indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries where there the mesh is allowed to move tangential to the boundary. All tangential mesh movements along those boundaries that have tangential material velocity boundary conditions are allowed by default, this parameters allows to generate mesh movements along other boundaries that are open, or have prescribed material velocities or tractions.

  The names of the boundaries listed here can either be numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Free surface stabilization theta`

  Value: 0.5

  Default: 0.5

  Description: Theta parameter described in Kaus et. al. 2010. An unstabilized free surface can overshoot its equilibrium position quite easily and generate unphysical results. One solution is to use a quasi-implicit correction term to the forces near the free surface. This parameter describes how much the free surface is stabilized with this term, where zero is no stabilization, and one is fully implicit.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Surface velocity projection`

  Value: normal

  Default: normal

  Description: After each time step the free surface must be advected in the direction of the velocity field. Mass conservation requires that the mesh velocity is in the normal direction of the surface. However,

for steep topography or large curvature, advection in the normal direction can become ill-conditioned, and instabilities in the mesh can form. Projection of the mesh velocity onto the local vertical direction can preserve the mesh quality better, but at the cost of slightly poorer mass conservation of the domain.

Possible values: [Selection normal—vertical]

5.32 Parameters in section `Geometry model`

- *Parameter name:* `Model name`

  Value: box

  Default: unspecified

  Description: Select one of the following models:

  'box': A box geometry parallel to the coordinate directions. The extent of the box in each coordinate direction is set in the parameter file. The box geometry labels its 2*dim sides as follows: in 2d, boundary indicators 0 through 3 denote the left, right, bottom and top boundaries; in 3d, boundary indicators 0 through 5 indicate left, right, front, back, bottom and top boundaries (see also the documentation of the deal.II class "GeometryInfo"). You can also use symbolic names "left", "right", etc., to refer to these boundaries in input files.

  'box with lithosphere boundary indicators': A box geometry parallel to the coordinate directions. The extent of the box in each coordinate direction is set in the parameter file. This geometry model labels its sides with 2*dim+2*(dim-1) boundary indicators: in 2d, boundary indicators 0 through 3 denote the left, right, bottom and top boundaries, while indicators4 and 5 denote the upper part of the left and right vertical boundary, respectively. In 3d, boundary indicators 0 through 5 indicate left, right, front, back, bottom and top boundaries (see also the documentation of the deal.II class "GeometryInfo"), while indicators 6, 7, 8 and 9 denote the left, rigth, front and back upper parts of the vertical boundaries, respectively. You can also use symbolic names "left", "right", "left lithosphere", etc., to refer to these boundaries in input files.

  Note that for a given "Global refinement level" and no user-specified "Repetitions", the lithosphere part of the mesh will be more refined.

  The additional boundary indicators for the lithosphere allow for selecting boundary conditions for the lithosphere different from those for the underlying mantle. An example application of this geometry is to prescribe a velocity on the lithospheric plates, but use open boundary conditions underneath.

  'chunk': A geometry which can be described as a chunk of a spherical shell, bounded by lines of longitude, latitude and radius. The minimum and maximum longitude, (latitude) and depth of the chunk is set in the parameter file. The chunk geometry labels its 2*dim sides as follows: "west" and "east": minimum and maximum longitude, "south" and "north": minimum and maximum latitude, "inner" and "outer": minimum and maximum radii. Names in the parameter files are as follows: Chunk (minimum —— maximum) (longitude —— latitude): edges of geographical quadrangle (in degrees)Chunk (inner —— outer) radius: Radii at bottom and top of chunk(Longitude —— Latitude —— Radius) repetitions: number of cells in each coordinate direction.

  'ellipsoidal chunk': A 3D chunk geometry that accounts for Earth's ellipticity (default assuming the WGS84 ellipsoid definition) which can be defined in non-coordinate directions. In the description of the ellipsoidal chunk, two of the ellipsoidal axes have the same length so that there is only a semi-major axis and a semi-minor axis. The user has two options for creating an ellipsoidal chunk geometry: 1) by defining two opposing points (SW and NE or NW and SE) a coordinate parallel ellipsoidal chunk geometry will be created. 2) by defining three points a non-coordinate parallel ellipsoidal chunk will be created. The points are defined in the input file by longitude:latitude. It is also possible to define additional subdivisions of the mesh in each direction. Faces of the model are defined as 0, west; 1,east; 2, south; 3, north; 4, inner; 5, outer.

‘sphere’: Geometry model for sphere with a user specified radius. This geometry has only a single boundary, so the only valid boundary indicator to specify in the input file is “0”. It can also be referenced by the symbolic name “surface” in input files.

‘spherical shell’: A geometry representing a spherical shell or a piece of it. Inner and outer radii are read from the parameter file in subsection ’Spherical shell’.

The model assigns boundary indicators as follows: In 2d, inner and outer boundaries get boundary indicators zero and one, and if the opening angle set in the input file is less than 360, then left and right boundaries are assigned indicators two and three. These boundaries can also be referenced using the symbolic names ’inner’, ’outer’ and (if applicable) ’left’, ’right’.

In 3d, inner and outer indicators are treated as in 2d. If the opening angle is chosen as 90 degrees, i.e., the domain is the intersection of a spherical shell and the first octant, then indicator 2 is at the face $x = 0$, 3 at $y = 0$, and 4 at $z = 0$. These last three boundaries can then also be referred to as ’east’, ’west’ and ’south’ symbolically in input files.

Possible values: [Selection box—box with lithosphere boundary indicators—chunk—ellipsoidal chunk—sphere—spherical shell—unspecified]

5.33 Parameters in section `Geometry model/Box`

- *Parameter name:* `Box origin X coordinate`

  Value: 0

  Default: 0

  Description: X coordinate of box origin. Units: m.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Box origin Y coordinate`

  Value: 0

  Default: 0

  Description: Y coordinate of box origin. Units: m.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Box origin Z coordinate`

  Value: 0

  Default: 0

  Description: Z coordinate of box origin. This value is ignored if the simulation is in 2d. Units: m.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `X extent`

  Value: 1

  Default: 1

  Description: Extent of the box in x-direction. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `X periodic`

  Value: false

  Default: false

  Description: Whether the box should be periodic in X direction

  Possible values: [Bool]

- *Parameter name:* `X repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in X direction.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Y extent`

  Value: 1

  Default: 1

  Description: Extent of the box in y-direction. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Y periodic`

  Value: false

  Default: false

  Description: Whether the box should be periodic in Y direction

  Possible values: [Bool]

- *Parameter name:* `Y repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in Y direction.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Z extent`

  Value: 1

  Default: 1

  Description: Extent of the box in z-direction. This value is ignored if the simulation is in 2d. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Z periodic`

  Value: false

  Default: false

  Description: Whether the box should be periodic in Z direction

  Possible values: [Bool]

- *Parameter name:* `Z repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in Z direction.

  Possible values: [Integer range 1...2147483647 (inclusive)]

5.34 Parameters in section `Geometry model/Box with lithosphere boundary indicators`

- *Parameter name:* `Box origin X coordinate`

  Value: 0

  Default: 0

  Description: X coordinate of box origin. Units: m.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Box origin Y coordinate`

  Value: 0

  Default: 0

  Description: Y coordinate of box origin. Units: m.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Box origin Z coordinate`

  Value: 0

  Default: 0

  Description: Z coordinate of box origin. This value is ignored if the simulation is in 2d. Units: m.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Lithospheric thickness`

  Value: 0.2

  Default: 0.2

  Description: The thickness of the lithosphere used to create additional boundary indicators to set specific boundary conditions for the lithosphere.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `X extent`

  Value: 1

  Default: 1

  Description: Extent of the box in x-direction. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `X periodic`

  Value: false

  Default: false

  Description: Whether the box should be periodic in X direction.

  Possible values: [Bool]

- *Parameter name:* `X periodic lithosphere`

  Value: false

  Default: false

  Description: Whether the box should be periodic in X direction in the lithosphere.

  Possible values: [Bool]

- *Parameter name:* `X repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in X direction of the lower box. The same number of repetitions will be used in the upper box.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Y extent`

  Value: 1

  Default: 1

  Description: Extent of the box in y-direction. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Y periodic`

  Value: false

  Default: false

  Description: Whether the box should be periodic in Y direction.

  Possible values: [Bool]

- *Parameter name:* `Y periodic lithosphere`

  Value: false

  Default: false

  Description: Whether the box should be periodic in Y direction in the lithosphere. This value is ignored if the simulation is in 2d.

  Possible values: [Bool]

- *Parameter name:* `Y repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in Y direction of the lower box. If the simulation is in 3d, the same number of repetitions will be used in the upper box.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Y repetitions lithosphere`

  Value: 1

  Default: 1

  Description: Number of cells in Y direction in the lithosphere. This value is ignored if the simulation is in 3d.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Z extent`

  Value: 1

  Default: 1

  Description: Extent of the box in z-direction. This value is ignored if the simulation is in 2d. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Z periodic`

  Value: false

  Default: false

  Description: Whether the box should be periodic in Z direction. This value is ignored if the simulation is in 2d.

  Possible values: [Bool]

- *Parameter name:* `Z repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in Z direction of the lower box. This value is ignored if the simulation is in 2d.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Z repetitions lithosphere`

  Value: 1

  Default: 1

  Description: Number of cells in Z direction in the lithosphere. This value is ignored if the simulation is in 2d.

  Possible values: [Integer range 1...2147483647 (inclusive)]

5.35 Parameters in section `Geometry model/Chunk`

- *Parameter name:* `Chunk inner radius`

  Value: 0

  Default: 0

  Description: Radius at the bottom surface of the chunk. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Chunk maximum latitude`

  Value: 1

  Default: 1

  Description: Maximum latitude of the chunk. This value is ignored if the simulation is in 2d. Units: degrees.

  Possible values: [Double -90...90 (inclusive)]

- *Parameter name:* `Chunk maximum longitude`

  Value: 1

  Default: 1

  Description: Maximum longitude of the chunk. Units: degrees.

  Possible values: [Double -180...360 (inclusive)]

- *Parameter name:* `Chunk minimum latitude`

  Value: 0

  Default: 0

  Description: Minimum latitude of the chunk. This value is ignored if the simulation is in 2d. Units: degrees.

  Possible values: [Double -90...90 (inclusive)]

- *Parameter name:* `Chunk minimum longitude`

  Value: 0

  Default: 0

  Description: Minimum longitude of the chunk. Units: degrees.

  Possible values: [Double -180...360 (inclusive)]

- *Parameter name:* `Chunk outer radius`

  Value: 1

  Default: 1

  Description: Radius at the top surface of the chunk. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Latitude repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in latitude. This value is ignored if the simulation is in 2d

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Longitude repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in longitude.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Radius repetitions`

  Value: 1

  Default: 1

  Description: Number of cells in radius.

  Possible values: [Integer range 1...2147483647 (inclusive)]

5.36 Parameters in section `Geometry model/Ellipsoidal chunk`

- *Parameter name:* `Depth`

  Value: 500000.0

  Default: 500000.0

  Description: Bottom depth of model region.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Depth subdivisions`

  Value: 1

  Default: 1

  Description: The number of subdivisions of the coarse (initial) mesh in depth.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `East-West subdivisions`

  Value: 1

  Default: 1

  Description: The number of subdivisions of the coarse (initial) mesh in the East-West direction.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Eccentricity`

  Value: 8.1819190842622e-2

  Default: 8.1819190842622e-2

  Description: Eccentricity of the ellipsoid. Zero is a perfect sphere, default (8.1819190842622e-2) is WGS84.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `NE corner`

  Value:

  Default:

  Description: Longitude:latitude in degrees of the North-East corner point of model region.The North-East direction is positive. If one of the three corners is not providedthe missing corner value will be calculated so all faces are parallel.

  Possible values: [Anything]

- *Parameter name:* `NW corner`

  Value:

  Default:

  Description: Longitude:latitude in degrees of the North-West corner point of model region. The North-East direction is positive. If one of the three corners is not providedthe missing corner value will be calculated so all faces are parallel.

  Possible values: [Anything]

- *Parameter name:* `North-South subdivisions`

  Value: 1

  Default: 1

  Description: The number of subdivisions of the coarse (initial) mesh in the North-South direction.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `SE corner`

  Value:

  Default:

Description: Longitude:latitude in degrees of the South-East corner point of model region. The North-East direction is positive. If one of the three corners is not providedthe missing corner value will be calculated so all faces are parallel.

Possible values: [Anything]

- *Parameter name:* `SW corner`

  Value:

  Default:

  Description: Longitude:latitude in degrees of the South-West corner point of model region. The North-East direction is positive. If one of the three corners is not providedthe missing corner value will be calculated so all faces are parallel.

  Possible values: [Anything]

- *Parameter name:* `Semi-major axis`

  Value: 6378137.0

  Default: 6378137.0

  Description: The semi-major axis (a) of an ellipsoid. This is the radius for a sphere (eccentricity=0). Default WGS84 semi-major axis.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.37 Parameters in section `Geometry model/Sphere`

- *Parameter name:* `Radius`

  Value: 6371000

  Default: 6371000

  Description: Radius of the sphere. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.38 Parameters in section `Geometry model/Spherical shell`

- *Parameter name:* `Cells along circumference`

  Value: 0

  Default: 0

  Description: The number of cells in circumferential direction that are created in the coarse mesh in 2d. If zero, this number is chosen automatically in a way that produces meshes in which cells have a reasonable aspect ratio for models in which the depth of the mantle is roughly that of the Earth. For planets with much shallower mantles and larger cores, you may want to chose a larger number to avoid cells that are elongated in tangential and compressed in radial direction.

  In 3d, the number of cells is computed differently and does not have an easy interpretation. Valid values for this parameter in 3d are 0 (let this class choose), 6, 12 and 96. Other possible values may be discussed in the documentation of the deal.II function GridGenerator::hyper_shell. The parameter is best left at its default in 3d.

  In either case, this parameter is ignored unless the opening angle of the domain is 360 degrees.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Inner radius`

  Value: 3481000

  Default: 3481000

  Description: Inner radius of the spherical shell. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Opening angle`

  Value: 360

  Default: 360

  Description: Opening angle in degrees of the section of the shell that we want to build. Units: degrees.

  Possible values: [Double 0...360 (inclusive)]

- *Parameter name:* `Outer radius`

  Value: 6336000

  Default: 6336000

  Description: Outer radius of the spherical shell. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.39 Parameters in section `Gravity model`

- *Parameter name:* `Model name`

  Value: vertical

  Default: unspecified

  Description: Select one of the following models:

  'function': Gravity is given in terms of an explicit formula that is elaborated in the parameters in section "Gravity model—Function". The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

  'radial constant': A gravity model in which the gravity direction is radially inward and at constant magnitude. The magnitude is read from the parameter file in subsection 'Radial constant'.

  'radial earth-like': A gravity model in which the gravity direction is radially inward and with a magnitude that matches that of the earth at the core-mantle boundary as well as at the surface and in between is physically correct under the assumption of a constant density.

  'radial linear': A gravity model which is radially inward, where the magnitudedecreases linearly with depth down to zero at the maximal depth the geometry returns, as you would get with a constantdensity spherical domain. (Note that this would be for a full sphere, not a spherical shell.) The magnitude of gravity at the surface is read from the input file in a section "Gravity model/Radial linear".

  'vertical': A gravity model in which the gravity direction is vertically downward and at a constant magnitude by default equal to one.

  Possible values: [Selection function—radial constant—radial earth-like—radial linear—vertical—unspecified]

5.40 Parameters in section `Gravity model/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0; 0

  Default: 0; 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.41 Parameters in section `Gravity model/Radial constant`

- *Parameter name:* `Magnitude`

  Value: 9.81

  Default: 9.81

  Description: Magnitude of the gravity vector in m/s^2. The direction is always radially inward towards the center of the earth.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.42 Parameters in section `Gravity model/Radial linear`

- *Parameter name:* `Magnitude at surface`

  Value: 9.8

  Default: 9.8

  Description: Magnitude of the radial gravity vector at the surface of the domain. Units: m/s^2

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.43 Parameters in section `Gravity model/Vertical`

- *Parameter name:* `Magnitude`

  Value: 1

  Default: 1

  Description: Value of the gravity vector in m/s^2 directed along negative y (2D) or z (3D) axis.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.44 Parameters in section `Heating model`

- *Parameter name:* `List of model names`

  Value:

  Default:

  Description: A comma separated list of heating models that will be used to calculate the heating terms in the energyequation. The results of each of these criteria , i.e., the heating source terms and the latent heat terms for theleft hand side will be added.

  The following heating models are available:

  'adiabatic heating': Implementation of a standard and a simplified model ofadiabatic heating.

  'constant heating': Implementation of a model in which the heating rate is constant.

  'function': Implementation of a model in which the heating rate is given in terms of an explicit formula that is elaborated in the parameters in section "Heating model—Function". The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

  The formula is interpreted as having units W/kg.

  Since the symbol t indicating time may appear in the formulas for the heating rate, it is interpreted as having units seconds unless the global parameter "Use years in output instead of seconds" is set.

  'latent heat': Implementation of a standard model for latent heat.

  'radioactive decay': Implementation of a model in which the internal heating rate is radioactive decaying in the following rule:

  $$(\text{initial concentration}) \cdot 0.5^{\text{time}/(\text{half life})}$$

  The crust and mantle can have different concentrations, and the crust can be defined either by depth or by a certain compositional field. The formula is interpreted as having units W/kg.

  'shear heating': Implementation of a standard model for shear heating.

  Possible values: [MultipleSelection adiabatic heating—constant heating—function—latent heat—radioactive decay—shear heating]

- *Parameter name:* `Model name`

  Value: unspecified

  Default: unspecified

  Description: Select one of the following models:

  Warning: This is the old formulation of specifying heating models and shouldn't be used. Please use 'List ofmodel names' instead.'adiabatic heating': Implementation of a standard and a simplified model ofadiabatic heating.

  'constant heating': Implementation of a model in which the heating rate is constant.

  'function': Implementation of a model in which the heating rate is given in terms of an explicit formula that is elaborated in the parameters in section "Heating model—Function". The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

  The formula is interpreted as having units W/kg.

  Since the symbol t indicating time may appear in the formulas for the heating rate, it is interpreted as having units seconds unless the global parameter "Use years in output instead of seconds" is set.

  'latent heat': Implementation of a standard model for latent heat.

  'radioactive decay': Implementation of a model in which the internal heating rate is radioactive decaying in the following rule:

  $$(\text{initial concentration}) \cdot 0.5^{\text{time}/(\text{half life})}$$

  The crust and mantle can have different concentrations, and the crust can be defined either by depth or by a certain compositional field. The formula is interpreted as having units W/kg.

  'shear heating': Implementation of a standard model for shear heating.

  Possible values: [Selection adiabatic heating—constant heating—function—latent heat—radioactive decay—shear heating—unspecified]

5.45 Parameters in section `Heating model/Adiabatic heating`

- *Parameter name:* `Use simplified adiabatic heating`

  Value: false

  Default: false

  Description: A flag indicating whether the adiabatic heating should be simplified from $\alpha T(\mathbf{u} \cdot \nabla p)$ to $\alpha \rho T(\mathbf{u} \cdot \mathbf{g})$.

  Possible values: [Bool]

5.46 Parameters in section `Heating model/Constant heating`

- *Parameter name:* `Radiogenic heating rate`

  Value: 0e0

  Default: 0e0

  Description: The specific rate of heating due to radioactive decay (or other bulk sources you may want to describe). This parameter corresponds to the variable H in the temperature equation stated in the manual, and the heating term is hoH. Units: W/kg.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.47 Parameters in section `Heating model/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.48 Parameters in section `Heating model/Latent heat`

5.49 Parameters in section `Heating model/Radioactive decay`

- *Parameter name:* `Crust composition number`

  Value: 0

  Default: 0

  Description: Which composition field should be treated as crust

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Crust defined by composition`

  Value: false

  Default: false

  Description: Whether crust defined by composition or depth

  Possible values: [Bool]

- *Parameter name:* `Crust depth`

  Value: 0

  Default: 0

  Description: Depth of the crust when crust if defined by depth. Units: m

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Half decay times`

  Value:

  Default:

  Description: Half decay times. Units: (Seconds), or (Years) if set 'use years instead of seconds'.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Heating rates`

  Value:

  Default:

  Description: Heating rates of different elements (W/kg)

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Initial concentrations crust`

  Value:

  Default:

  Description: Initial concentrations of different elements (ppm)

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Initial concentrations mantle`

  Value:

  Default:

  Description: Initial concentrations of different elements (ppm)

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Number of elements`

  Value: 0

  Default: 0

  Description: Number of radioactive elements

  Possible values: [Integer range 0...2147483647 (inclusive)]

5.50 Parameters in section `Heating model/Shear heating`

5.51 Parameters in section `Initial conditions`

- *Parameter name:* `Model name`

  Value: perturbed box

  Default: unspecified

  Description: Select one of the following models:

  'S40RTS perturbation': An initial temperature field in which the temperature is perturbed following the S20RTS or S40RTS shear wave velocity model by Ritsema and others, which can be downloaded here http://www.earth.lsa.umich.edu/~jritsema/research.html. Information on the vs model can be found in Ritsema, J., Deuss, A., van Heijst, H.J. & Woodhouse, J.H., 2011. S40RTS: a degree-40 shear-velocity model for the mantle from new Rayleigh wave dispersion, teleseismic traveltime and normal-mode splitting function measurements, Geophys. J. Int. 184, 1223-1236. The scaling between the shear wave perturbation and the temperature perturbation can be set by the user with the 'vs to density scaling' parameter and the 'Thermal expansion coefficient in initial temperature scaling' parameter. The scaling is as follows: $\delta ln\rho(r,\theta,\phi) = \xi \cdot \delta ln v_s(r,\theta,\phi)$ and $\delta T(r,\theta,\phi) = -\frac{1}{\alpha}\delta ln\rho(r,\theta,\phi)$. ξ is the 'vs to density scaling' parameter and α is the 'Thermal expansion coefficient in initial temperature scaling' parameter. The temperature perturbation is added to an otherwise constant temperature (incompressible model) or adiabatic reference profile (compressible model). If a depth is specified in 'Remove temperature heterogeneity down to specified depth', there is no temperature perturbation prescribed down to that depth.

  'SAVANI perturbation': An initial temperature field in which the temperature is perturbed following the SAVANI shear wave velocity model by Auer and others, which can be downloaded here http://n.ethz.ch/~auerl/savani.tar.bz2. Information on the vs model can be found in Auer, L., Boschi, L., Becker, T.W., Nissen-Meyer, T. & Giardini, D., 2014. Savani:A variable resolution wholemantle model of anisotropic shear velocityvariations based on multiple data sets. Journal of GeophysicalResearch: Solid Earth 119.4 (2014): 3006-3034. The scaling between the shear wave perturbation and the temperature perturbation can be set by the user with the 'vs to density scaling' parameter and the 'Thermal expansion coefficient in initial temperature scaling' parameter. The scaling is as follows: $\delta ln\rho(r,\theta,\phi) = \xi \cdot \delta ln v_s(r,\theta,\phi)$ and $\delta T(r,\theta,\phi) = -\frac{1}{\alpha}\delta ln\rho(r,\theta,\phi)$. ξ is the 'vs to density scaling' parameter and α is the 'Thermal expansion coefficient in initial temperature scaling' parameter. The temperature perturbation is added to an otherwise constant temperature (incompressible model) or adiabatic reference profile (compressible model).

  'adiabatic': Temperature is prescribed as an adiabatic profile with upper and lower thermal boundary layers, whose ages are given as input parameters.

  'adiabatic boundary': An initial temperature condition that allows for discretizing the model domain into two layers separated by a user-defined isothermal boundary using a table look-up approach. The user includes an input ascii data file that is formatted as 3 columns of 'latitude', 'longitude', and 'depth', where 'depth' is in kilometers and represents the depth of an initial temperature of 1673.15 K (by default). The temperature is defined from the surface (273.15 K) to the isotherm as a linear gradient. Below the isotherm the temperature increases approximately adiabatically (0.0005 K per meter). This initial temperature condition is designed specifically for the ellipsoidal chunk geometry model.

  'ascii data': Implementation of a model in which the initial temperature is derived from files containing data in ascii format. Note the required format of the input data: The first lines may contain any number of comments if they begin with '#', but one of these lines needs to contain the number of grid points in each dimension as for example '# POINTS: 3 3'. The order of the data columns has to be 'x', 'y', 'Temperature [K]' in a 2d model and 'x', 'y', 'z', 'Temperature [K]' in a 3d model, which means that there has to be a single column containing the temperature. Note that the data in the input files need

to be sorted in a specific order: the first coordinate needs to ascend first, followed by the second and the third at last in order to assign the correct data to the prescribed coordinates. If you use a spherical model, then the data will still be handled as Cartesian, however the assumed grid changes. 'x' will be replaced by the radial distance of the point to the bottom of the model, 'y' by the azimuth angle and 'z' by the polar angle measured positive from the north pole. The grid will be assumed to be a latitude-longitude grid. Note that the order of spherical coordinates is 'r', 'phi', 'theta' and not 'r', 'theta', 'phi', since this allows for dimension independent expressions.

'function': Specify the initial temperature in terms of an explicit formula. The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

'harmonic perturbation': An initial temperature field in which the temperature is perturbed following a harmonic function (spherical harmonic or sine depending on geometry and dimension) in lateral and radial direction from an otherwise constant temperature (incompressible model) or adiabatic reference profile (compressible model).

'inclusion shape perturbation': An initial temperature field in which there is an inclusion in a constant-temperature box field. The size, shape, gradient, position, and temperature of the inclusion are defined by parameters.

'mandelbox': Fractal-shaped temperature field.

'perturbed box': An initial temperature field in which the temperature is perturbed slightly from an otherwise constant value equal to one. The perturbation is chosen in such a way that the initial temperature is constant to one along the entire boundary.

'polar box': An initial temperature field in which the temperature is perturbed slightly from an otherwise constant value equal to one. The perturbation is such that there are two poles on opposing corners of the box.

'solidus': This is a temperature initial condition that starts the model close to solidus, it also contains a user defined lithoshpere thickness and with perturbations in both lithosphere thickness and temperature based on spherical harmonic functions. It was used as the initial condition of early Mars after the freezing of the magma ocean, using the solidus from Parmentier et al., Melt-solid segregation, Fractional magma ocean solidification, and implications for longterm planetary evolution. Luna and Planetary Science, 2007.

'spherical gaussian perturbation': An initial temperature field in which the temperature is perturbed by a single Gaussian added to an otherwise spherically symmetric state. Additional parameters are read from the parameter file in subsection 'Spherical gaussian perturbation'.

'spherical hexagonal perturbation': An initial temperature field in which the temperature is perturbed following an N-fold pattern in a specified direction from an otherwise spherically symmetric state. The class's name comes from previous versions when the only option was $N = 6$.

Possible values: [Selection S40RTS perturbation—SAVANI perturbation—adiabatic—adiabatic boundary—ascii data—function—harmonic perturbation—inclusion shape perturbation—mandelbox—perturbed box—polar box—solidus—spherical gaussian perturbation—spherical hexagonal perturbation—unspecified]

5.52 Parameters in section `Initial conditions/Adiabatic`

- *Parameter name:* `Age bottom boundary layer`

  Value: 0e0

  Default: 0e0

  Description: The age of the lower thermal boundary layer, used for the calculation of the half-space cooling model temperature. Units: years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Age top boundary layer`

  Value: 0e0

  Default: 0e0

  Description: The age of the upper thermal boundary layer, used for the calculation of the half-space cooling model temperature. Units: years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Amplitude`

  Value: 0e0

  Default: 0e0

  Description: The amplitude (in K) of the initial spherical temperature perturbation at the bottom of the model domain. This perturbation will be added to the adiabatic temperature profile, but not to the bottom thermal boundary layer. Instead, the maximum of the perturbation and the bottom boundary layer temperature will be used.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Position`

  Value: center

  Default: center

  Description: Where the initial temperature perturbation should be placed. If 'center' is given, then the perturbation will be centered along a 'midpoint' of some sort of the bottom boundary. For example, in the case of a box geometry, this is the center of the bottom face; in the case of a spherical shell geometry, it is along the inner surface halfway between the bounding radial lines.

  Possible values: [Selection center]

- *Parameter name:* `Radius`

  Value: 0e0

  Default: 0e0

  Description: The Radius (in m) of the initial spherical temperature perturbation at the bottom of the model domain.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Subadiabaticity`

  Value: 0e0

  Default: 0e0

  Description: If this value is larger than 0, the initial temperature profile will not be adiabatic, but subadiabatic. This value gives the maximal deviation from adiabaticity. Set to 0 for an adiabatic temperature profile. Units: K.

  The function object in the Function subsection represents the compositional fields that will be used as a reference profile for calculating the thermal diffusivity. This function is one-dimensional and depends only on depth. The format of this functions follows the syntax understood by the muparser library, see Section 5.1.3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.53 Parameters in section `Initial conditions/Adiabatic/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,t

  Default: x,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.54 Parameters in section `Initial conditions/Adiabatic boundary`

- *Parameter name:* `Adiabatic temperature gradient`

  Value: 0.0005

  Default: 0.0005

  Description: The value of the adiabatic temperature gradient. Units: Km^{-1}.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/initial-conditions/adiabatic-boundary/

  Default: $ASPECT_SOURCE_DIR/data/initial-conditions/adiabatic-boundary/

  Description: The path to the isotherm depth data file

  Possible values: [DirectoryName]

- *Parameter name:* `Isotherm depth filename`

  Value: adiabatic_boundary.txt

  Default: adiabatic_boundary.txt

  Description: File from which the isotherm depth data is read.

  Possible values: [FileName (Type: input)]

- *Parameter name:* `Isotherm temperature`

  Value: 1673.15

  Default: 1673.15

  Description: The value of the isothermal boundary temperature. Units: Kelvin.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Surface temperature`

  Value: 273.15

  Default: 273.15

  Description: The value of the suface temperature. Units: Kelvin.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.55 Parameters in section `Initial conditions/Ascii data model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/initial-conditions/ascii-data/test/

  Default: $ASPECT_SOURCE_DIR/data/initial-conditions/ascii-data/test/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: box_2d.txt

  Default: box_2d.txt

  Description: The file name of the material data. Provide file in format: (Velocity file name).%s%d where %s is a string specifying the boundary of the model according to the names of the boundary indicators (of a box or a spherical shell).%d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

  Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction. Another way to use this factor is to convert units of the input files. The unit is assumed to bem/s or m/yr depending on the 'Use years in output instead of seconds' flag. If you provide velocities in cm/yr set this factor to 0.01.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.56 Parameters in section `Initial conditions/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.57 Parameters in section `Initial conditions/Harmonic perturbation`

- *Parameter name:* `Lateral wave number one`

  Value: 3

  Default: 3

  Description: Doubled first lateral wave number of the harmonic perturbation. Equals the spherical harmonic degree in 3D spherical shells. In all other cases one equals half of a sine period over the model domain. This allows for single up-/downswings. Negative numbers reverse the sign of the perturbation but are not allowed for the spherical harmonic case.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Lateral wave number two`

  Value: 2

  Default: 2

  Description: Doubled second lateral wave number of the harmonic perturbation. Equals the spherical harmonic order in 3D spherical shells. In all other cases one equals half of a sine period over the model domain. This allows for single up-/downswings. Negative numbers reverse the sign of the perturbation.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Magnitude`

  Value: 1.0

  Default: 1.0

  Description: The magnitude of the Harmonic perturbation.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 1600.0

  Default: 1600.0

  Description: The reference temperature that is perturbed by theharmonic function. Only used in incompressible models.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Vertical wave number`

  Value: 1

  Default: 1

  Description: Doubled radial wave number of the harmonic perturbation. One equals half of a sine period over the model domain. This allows for single up-/downswings. Negative numbers reverse the sign of the perturbation.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

5.58 Parameters in section `Initial conditions/Inclusion shape perturbation`

- *Parameter name:* `Ambient temperature`

  Value: 1.0

  Default: 1.0

  Description: The background temperature for the temperature field.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Center X`

  Value: 0.5

  Default: 0.5

  Description: The X coordinate for the center of the shape.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Center Y`

  Value: 0.5

  Default: 0.5

  Description: The Y coordinate for the center of the shape.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Center Z`

  Value: 0.5

  Default: 0.5

  Description: The Z coordinate for the center of the shape. This is only necessary for three-dimensional fields.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Inclusion gradient`

  Value: constant

  Default: constant

  Description: The gradient of the inclusion to be generated.

  Possible values: [Selection gaussian—linear—constant]

- *Parameter name:* `Inclusion shape`

  Value: circle

  Default: circle

  Description: The shape of the inclusion to be generated.

  Possible values: [Selection square—circle]

- *Parameter name:* `Inclusion temperature`

  Value: 0.0

  Default: 0.0

  Description: The temperature of the inclusion shape. This is only the true temperature in the case of the constant gradient. In all other cases, it gives one endpoint of the temperature gradient for the shape.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Shape radius`

  Value: 1.0

  Default: 1.0

  Description: The radius of the inclusion to be generated. For shapes with no radius (e.g. square), this will be the width, and for shapes with no width, this gives a general guideline for the size of the shape.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.59 Parameters in section `Initial conditions/S40RTS perturbation`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/initial-conditions/S40RTS/

  Default: $ASPECT_SOURCE_DIR/data/initial-conditions/S40RTS/

  Description: The path to the model data.

  Possible values: [DirectoryName]

- *Parameter name:* `Initial condition file name`

  Value: S40RTS.sph

  Default: S40RTS.sph

  Description: The file name of the spherical harmonics coefficients from Ritsema et al.

  Possible values: [Anything]

- *Parameter name:* `Maximum order`

  Value: 20

  Default: 20

  Description: The maximum order the users specify when reading the data file of spherical harmonic coefficients, which must be smaller than the maximum order the data file stored. This parameter will be used only if 'Specify a lower maximum order' is set to true

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 1600.0

  Default: 1600.0

  Description: The reference temperature that is perturbed by the spherical harmonic functions. Only used in incompressible models.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Remove degree 0 from perturbation`

  Value: true

  Default: true

  Description: Option to remove the degree zero component from the perturbation, which will ensure that the laterally averaged temperature for a fixed depth is equal to the background temperature.

  Possible values: [Bool]

- *Parameter name:* `Remove temperature heterogeneity down to specified depth`

  Value: -1.7976931348623157e+308

  Default: -1.7976931348623157e+308

  Description: This will set the heterogeneity prescribed by S20RTS or S40RTS to zero down to the specified depth (in meters). Note that your resolution has to be adquate to capture this cutoff. For example if you specify a depth of 660km, but your closest spherical depth layers are only at 500km and 750km (due to a coarse resolution) it will only zero out heterogeneities down to 500km. Similar caution has to be taken when using adaptive meshing.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Specify a lower maximum order`

  Value: false

  Default: false

  Description: Option to use a lower maximum order when reading the data file of spherical harmonic coefficients. This is probably used for the faster tests or when the users only want to see the spherical harmonic pattern up to a certain order.

  Possible values: [Bool]

- *Parameter name:* `Spline knots depth file name`

  Value: Spline_knots.txt

  Default: Spline_knots.txt

  Description: The file name of the spline knot locations from Ritsema et al.

  Possible values: [Anything]

- *Parameter name:* `Thermal expansion coefficient in initial temperature scaling`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient β. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Vs to density scaling`

  Value: 0.25

  Default: 0.25

  Description: This parameter specifies how the perturbation in shear wave velocity as prescribed by S20RTS or S40RTS is scaled into a density perturbation. See the general description of this model for more detailed information.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.60 Parameters in section `Initial conditions/SAVANI perturbation`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/initial-conditions/SAVANI/

  Default: $ASPECT_SOURCE_DIR/data/initial-conditions/SAVANI/

  Description: The path to the model data.

  Possible values: [DirectoryName]

- *Parameter name:* `Initial condition file name`

  Value: savani.dlnvs.60.m.ab

  Default: savani.dlnvs.60.m.ab

  Description: The file name of the spherical harmonics coefficients from Auer et al.

  Possible values: [Anything]

- *Parameter name:* `Maximum order`

  Value: 20

  Default: 20

  Description: The maximum order the users specify when reading the data file of spherical harmonic coefficients, which must be smaller than the maximum order the data file stored. This parameter will be used only if 'Specify a lower maximum order' is set to true

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 1600.0

  Default: 1600.0

  Description: The reference temperature that is perturbed by the spherical harmonic functions. Only used in incompressible models.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Remove degree 0 from perturbation`

  Value: true

  Default: true

  Description: Option to remove the degree zero component from the perturbation, which will ensure that the laterally averaged temperature for a fixed depth is equal to the background temperature.

  Possible values: [Bool]

- *Parameter name:* `Remove temperature heterogeneity down to specified depth`

  Value: -1.7976931348623157e+308

  Default: -1.7976931348623157e+308

  Description: This will set the heterogeneity prescribed by SAVANI to zero down to the specified depth (in meters). Note that your resolution has to be adquate to capture this cutoff. For example if you specify a depth of 660km, but your closest spherical depth layers are only at 500km and 750km (due to a coarse resolution) it will only zero out heterogeneities down to 500km. Similar caution has to be taken when using adaptive meshing.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Specify a lower maximum order`

  Value: false

  Default: false

  Description: Option to use a lower maximum order when reading the data file of spherical harmonic coefficients. This is probably used for the faster tests or when the users only want to see the spherical harmonic pattern up to a certain order.

  Possible values: [Bool]

- *Parameter name:* `Spline knots depth file name`

  Value: Spline_knots.txt

  Default: Spline_knots.txt

  Description: The file name of the spline knots taken from the 28 spherical layersof SAVANI tomography model.

  Possible values: [Anything]

- *Parameter name:* `Thermal expansion coefficient in initial temperature scaling`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient β. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Vs to density scaling`

  Value: 0.25

  Default: 0.25

  Description: This parameter specifies how the perturbation in shear wave velocity as prescribed by SAVANI is scaled into a density perturbation. See the general description of this model for more detailed information.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.61 Parameters in section `Initial conditions/Solidus`

- *Parameter name:* `Lithosphere thickness`

  Value: 0

  Default: 0

  Description: The thickness of lithosphere thickness. Units: m

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Supersolidus`

  Value: 0e0

  Default: 0e0

  Description: The difference from solidus, use this number to generate initial conditions that close to solidus instead of exactly at solidus. Use small negative number in this parameter to prevent large melting generation at the beginning. Units: K

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.62 Parameters in section `Initial conditions/Solidus/Data`

- *Parameter name:* `Solidus filename`

  Value:

  Default:

  Description: The solidus data filename. It is a function of radius or pressure in the following format: Line 1: Header Line 2: Unit of temperature (C/K) Unit of pressure (GPa/kbar) or radius (km/m) Line 3 : Column of solidus temperature Column of radius/pressure See data/initial-temperature/solidus.Mars as an example.

  In order to facilitate placing input files in locations relative to the ASPECT source directory, the file name may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [Anything]

5.63 Parameters in section `Initial conditions/Solidus/Perturbation`

- *Parameter name:* `Lateral wave number one`

  Value: 3

  Default: 3

  Description: Doubled first lateral wave number of the harmonic perturbation. Equals the spherical harmonic degree in 3D spherical shells. In all other cases one equals half of a sine period over the model domain. This allows for single up-/downswings. Negative numbers reverse the sign of the perturbation but are not allowed for the spherical harmonic case.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Lateral wave number two`

  Value: 2

  Default: 2

  Description: Doubled second lateral wave number of the harmonic perturbation. Equals the spherical harmonic order in 3D spherical shells. In all other cases one equals half of a sine period over the model domain. This allows for single up-/downswings. Negative numbers reverse the sign of the perturbation.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Lithosphere thickness amplitude`

  Value: 0e0

  Default: 0e0

  Description: The amplitude of the initial lithosphere thickness perturbation in (m)

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Temperature amplitude`

  Value: 0e0

  Default: 0e0

  Description: The amplitude of the initial spherical temperature perturbation in (K)

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.64 Parameters in section `Initial conditions/Spherical gaussian perturbation`

- *Parameter name:* `Amplitude`

  Value: 0.01

  Default: 0.01

  Description: The amplitude of the perturbation.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Angle`

  Value: 0e0

  Default: 0e0

  Description: The angle where the center of the perturbation is placed.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Filename for initial geotherm table`

  Value: initial-geotherm-table

  Default: initial-geotherm-table

  Description: The file from which the initial geotherm table is to be read. The format of the file is defined by what is read in source/initial_conditions/spherical_shell.cc.

  Possible values: [FileName (Type: input)]

- *Parameter name:* `Non-dimensional depth`

  Value: 0.7

  Default: 0.7

  Description: The non-dimensional radial distance where the center of the perturbation is placed.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Sigma`

  Value: 0.2

  Default: 0.2

  Description: The standard deviation of the Gaussian perturbation.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Sign`

  Value: 1

  Default: 1

  Description: The sign of the perturbation.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.65 Parameters in section `Initial conditions/Spherical hexagonal perturbation`

- *Parameter name:* `Angular mode`

  Value: 6

  Default: 6

  Description: The number of convection cells to perturb the system with.

  Possible values: [Integer range -2147483648...2147483647 (inclusive)]

- *Parameter name:* `Rotation offset`

  Value: -45

  Default: -45

  Description: Amount of clockwise rotation in degrees to apply to the perturbations. Default is set to -45 in order to provide backwards compatibility.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.66 Parameters in section `Material model`

- *Parameter name:* `Material averaging`

  Value: none

  Default: none

  Description: Whether or not (and in the first case, how) to do any averaging of material model output data when constructing the linear systems for velocity/pressure, temperature, and compositions in each time step, as well as their corresponding preconditioners.

  Possible choices: none—arithmetic average—harmonic average—geometric average—pick largest—project to Q1—log average

  The process of averaging, and where it may be used, is discussed in more detail in Section 6.2.8.

  More averaging schemes are available in the averaging material model. This material model is a "compositing material model" which can be used in combination with other material models.

  Possible values: [Selection none—arithmetic average—harmonic average—geometric average—pick largest—project to Q1—log average]

- *Parameter name:* `Model name`

  Value: simple

  Default: unspecified

  Description: The name of the material model to be used in this simulation. There are many material models you can choose from, as listed below. They generally fall into two category: (i) models that implement a particular case of material behavior, (ii) models that modify other models in some way. We sometimes call the latter "compositing models". An example of a compositing model is the "depth dependent" model below in that it takes another, freely choosable model as its base and then modifies that model's output in some way.

  You can select one of the following models:

  'Morency and Doin': An implementation of the visco-plastic rheology described by (Morency and Doin, 2004). Compositional fields can each be assigned individual activation energies, reference densities, thermal expansivities, and stress exponents. The effective viscosity is defined as

$$v_{eff} = \left(\frac{1}{v^v_{eff}} + \frac{1}{v^p_{eff}}\right)^{-1}$$

  where

$$v^v_{eff} = B\left(\frac{\dot\epsilon}{\dot\epsilon_{ref}}\right)^{-1+1/n_v} exp\left(\frac{E_a + V_a\rho_m g z}{n_v RT}\right)$$

$$v^p_{eff} = (\tau_0 + \gamma\rho_m g z)\left(\frac{\dot\epsilon^{-1+1/n_p}}{\dot\epsilon^{1/n_p}_{ref}}\right)$$

  where B is a scaling constant; $\dot\epsilon$ is defined as the quadratic sum of the second invariant of the strain rate tensor and a minimum strain rate, $\dot\epsilon_0$; $\dot\epsilon_{ref}$ is a reference strain rate; n_v, and n_p are stress exponents; E_a is the activation energy; V_a is the activation volume; ρ_m is the mantle density; R is the gas constant; T is temperature; τ_0 is the cohestive strength of rocks at the surface; γ is a coefficient of yield stress increase with depth; and z is depth.

  Note: (Morency and Doin, 2004) defines the second invariant of the strain rate in a nonstandard way. The formulation in the paper is given as $\epsilon_{II} = \sqrt{\frac{1}{2}(\epsilon_{11}^2 + \epsilon_{12}^2)}$ where ϵ is the strain rate tensor. For consistency, that is also the formulation implemented in this material model.

Morency, C., and MP. Doin. "Numerical simulations of the mantle lithosphere delamination." Journal of Geophysical Research: Solid Earth (19782012) 109.B3 (2004).

The value for the components of this formula and additional parameters are read from the parameter file in subsection 'Material model/Morency and Doin'.

'Steinberger': This material model looks up the viscosity from the tables that correspond to the paper of Steinberger and Calderwood 2006 ("Models of large-scale viscous flow in the Earth's mantle with constraints from mineral physics and surface observations", Geophys. J. Int., 167, 1461-1481, http://dx.doi.org/10.1111/j.1365-246X.2006.03131.x) and material data from a database generated by the thermodynamics code `Perplex`, see http://www.perplex.ethz.ch/. The default example data builds upon the thermodynamic database by Stixrude 2011 and assumes a pyrolitic composition by Ringwood 1988 but is easily replaceable by other data files.

'averaging': The "averaging" Material model applies an averaging of the quadrature points within a cell. The values to average are supplied by any of the other available material models. In other words, it is a "compositing material model". Parameters related to the average model are read from a subsection "Material model/Averaging".

The user must specify a "Base model" from which material properties are derived. Furthermore an averaging operation must be selected, where the Choice should be from the list none—arithmetic average—harmonic average—geometric average—pick largest—log average—NWD arithmetic average—NWD harmonic average—NWD geometric average.

NWD stands for Normalized Weighed Distance. The models with this in front of their name work with a weighed average, which means each quadrature point requires an individual weight. The weight is determined by the distance, where the exact relation is determined by a bell shaped curve. A bell shaped curve is a continuous function which is one at it's maximum and exactly zero at and beyond it's limit. This bell shaped curve is spanned around each quadrature point to determine the weighting map for each quadrature point. The used bell shape comes from Lucy (1977). The distance is normalized so the largest distance becomes one. This means that if variable "Bell shape limit" is exactly one, the farthest quadrature point is just on the limit and it's weight will be exactly zero. In this plugin it is not implemented as larger and equal than the limit, but larger than, to ensure the the quadrature point at distance zero is always included.

'composition reaction': A material model that behaves in the same way as the simple material model, but includes two compositional fields and a reaction between them. Above a depth given in the input file, the first fields gets converted to the second field.

'depth dependent': The "depth dependent" Material model applies a depth-dependent scaling to any of the other available material models. In other words, it is a "compositing material model".

Parameters related to the depth dependent model are read from a subsection "Material model/Depth dependent model". The user must specify a "Base model" from which material properties are derived. Currently the depth dependent model only allows depth dependence of viscosity - other material properties are taken from the "Base model". Viscosity η at depth z is calculated according to:

$$\eta(z,p,T,X,...) = \eta(z)\eta_b(p,T,X,..)/\eta_{rb} \quad (28)$$

where $\eta(z)$ is the the depth-dependence specified by the depth dependent model, $\eta_b(p,T,X,...)$ is the viscosity calculated from the base model, and η_{rb} is the reference viscosity of the "Base model". In addition to the specification of the "Base model", the user must specify the method to be used to calculate the depth-dependent viscosity $\eta(z)$ as "Material model/Depth dependent model/Depth dependence method", which can be chosen among "None—Function—File—List". Each method and the associated parameters are as follows:

"Function": read a user-specified parsed function from the input file in a subsection "Material model/Depth dependent model/Viscosity depth function". By default, this function is uniformly equal to

1.0e21. Specifying a function that returns a value less than or equal to 0.0 anywhere in the model domain will produce an error.

"File": read a user-specified file containing viscosity values at specified depths. The file containing depth-dependent viscosities is read from a directory specified by the user as "Material model/Depth dependent model/Data directory", from a file with name specified as "Material model/Depth dependent model/Viscosity depth file". The format of this file is ascii text and contains two columns with one header line:

example Viscosity depth file:
Depth (m) Viscosity (Pa-s)
0.0000000e+00 1.0000000e+21
6.7000000e+05 1.0000000e+22

Viscosity is interpolated from this file using linear interpolation. "None": no depth-dependence. Viscosity is taken directly from "Base model"

"List:": read a comma-separated list of depth values corresponding to the maximum depths of layers having constant depth-dependence $\eta(z)$. The layers must be specified in order of increasing depth, and the last layer in the list must have a depth greater than or equal to the maximal depth of the model. The list of layer depths is specified as "Material model/Depth dependent model/Depth list" and the corresponding list of layer viscosities is specified as "Material model/Depth dependent model/Viscosity list"

'diffusion dislocation': An implementation of a viscous rheology including diffusion and dislocation creep. Compositional fields can each be assigned individual activation energies, reference densities, thermal expansivities, and stress exponents. The effective viscosity is defined as

$$v_{\text{eff}} = \left(\frac{1}{v_{\text{eff}}^{\text{diff}}} + \frac{1}{v_{\text{eff}}^{\text{dis}}} \right)^{-1}$$

where

$$v_{\text{i}} = 0.5 * A^{-\frac{1}{n_i}} d^{\frac{m_i}{n_i}} \dot{\varepsilon}_i^{\frac{1-n_i}{n_i}} \exp\left(\frac{E_i^* + PV_i^*}{n_i RT} \right)$$

where d is grain size, i corresponds to diffusion or dislocation creep, $\dot{\varepsilon}$ is the square root of the second invariant of the strain rate tensor, R is the gas constant, T is temperature, and P is pressure. A_i are prefactors, n_i and m_i are stress and grain size exponents E_i are the activation energies and V_i are the activation volumes.

The ratio of diffusion to dislocation strain rate is found by Newton's method, iterating to find the stress which satisfies the above equations. The value for the components of this formula and additional parameters are read from the parameter file in subsection 'Material model/DiffusionDislocation'.

'drucker prager': A material model that has constant values for all coefficients but the density and viscosity. The defaults for all coefficients are chosen to be similar to what is believed to be correct for Earth's mantle. All of the values that define this model are read from a section "Material model/-Drucker Prager" in the input file, see Section **??**.Note that the model does not take into account any dependencies of material properties on compositional fields.

The viscosity is computed according to the Drucker Prager frictional plasticity criterion (non-associative) based on a user-defined internal friction angle ϕ and cohesion C. In 3D: $\sigma_y = \frac{6C\cos(\phi)}{\sqrt{(3)}(3+\sin(\phi))} + \frac{2P\sin(\phi)}{\sqrt{(3)}(3+\sin(\phi))}$, where P is the pressure. See for example Zienkiewicz, O. C., Humpheson, C. and Lewis, R. W. (1975), Géotechnique 25, No. 4, 671-689. With this formulation we circumscribe instead of inscribe the Mohr Coulomb yield surface. In 2D the Drucker Prager yield surface is the same as

the Mohr Coulomb surface: $\sigma_y = P\sin(\phi) + C\cos(\phi)$. Note that in 2D for $\phi = 0$, these criteria revert to the von Mises criterion (no pressure dependence). See for example Thieulot, C. (2011), PEPI 188, 47-68.

Note that we enforce the pressure to be positive to prevent negative yield strengths and viscosities.

We then use the computed yield strength to scale back the viscosity on to the yield surface using the Viscosity Rescaling Method described in Kachanov, L. M. (2004), Fundamentals of the Theory of Plasticity, Dover Publications, Inc. (Not Radial Return.)A similar implementation can be found in GALE (https://geodynamics.org/cig/software/gale/gale-manual.pdf).

To avoid numerically unfavourably large (or even negative) viscosity ranges, we cut off the viscosity with a user-defined minimum and maximum viscosity: $\eta_eff = \frac{1}{\frac{1}{\eta_min + \eta} + \frac{1}{\eta_max}}$.

Note that this model uses the formulation that assumes an incompressible medium despite the fact that the density follows the law $\rho(T) = \rho_0(1 - \beta(T - T_{\text{ref}}))$.

'latent heat': A material model that includes phase transitions and the possibility that latent heat is released or absorbed when material crosses one of the phase transitions of up to two different materials (compositional fields). This model implements a standard approximation of the latent heat terms following Christensen & Yuen, 1985. The change of entropy is calculated as $DeltaS = \gamma\frac{\Delta\rho}{\rho^2}$ with the Clapeyron slope γ and the density change $\Delta\rho$ of the phase transition being input parameters. The model employs an analytic phase function in the form $X = 0.5\left(1 + \tanh\left(\frac{\Delta p}{\Delta p_0}\right)\right)$ with $\Delta p = p - p_{transition} - \gamma\left(T - T_{transition}\right)$ and Δp_0 being the pressure difference over the width of the phase transition (specified as input parameter).

'latent heat melt': A material model that includes the latent heat of melting for two materials: peridotite and pyroxenite. The melting model for peridotite is taken from Katz et al., 2003 (A new parameterization of hydrous mantle melting) and the one for pyroxenite from Sobolev et al., 2011 (Linking mantle plumes, large igneous provinces and environmental catastrophes). The model assumes a constant entropy change for melting 100% of the material, which can be specified in the input file. The partial derivatives of entropy with respect to temperature and pressure required for calculating the latent heat consumption are then calculated as product of this constant entropy change, and the respective derivative of the function the describes the melt fraction. This is linearly averaged with respect to the fractions of the two materials present. If no compositional fields are specified in the input file, the model assumes that the material is peridotite. If compositional fields are specified, the model assumes that the first compositional field is the fraction of pyroxenite and the rest of the material is peridotite.

Otherwise, this material model has a temperature- and pressure-dependent density and viscosity and the density and thermal expansivity depend on the melt fraction present. It is possible to extent this model to include a melt fraction dependence of all the material parameters by calling the function melt_fraction in the calculation of the respective parameter. However, melt and solid move with the same velocity and melt extraction is not taken into account (batch melting).

'multicomponent': This model is for use with an arbitrary number of compositional fields, where each field represents a rock type which can have completely different properties from the others. However, each rock type itself has constant material properties. The value of the compositional field is interpreed as a volume fraction. If the sum of the fields is greater than one, they are renormalized. If it is less than one, material properties for "background mantle" make up the rest. When more than one field is present, the material properties are averaged arithmetically. An exception is the viscosity, where the averaging should make more of a difference. For this, the user selects between arithmetic, harmonic, geometric, or maximum composition averaging.

'simple': A material model that has constant values for all coefficients but the density and viscosity. The defaults for all coefficients are chosen to be similar to what is believed to be correct for Earth's

mantle. All of the values that define this model are read from a section "Material model/Simple model" in the input file, see Section 5.79.

This model uses the following set of equations for the two coefficients that are non-constant:

$$\eta(p,T,\mathfrak{c}) = \tau(T)\zeta(\mathfrak{c})\eta_0, \tag{29}$$

$$\rho(p,T,\mathfrak{c}) = (1-\alpha(T-T_0))\,\rho_0 + \Delta\rho\; c_0, \tag{30}$$

where c_0 is the first component of the compositional vector $\mathfrak{c}$ if the model uses compositional fields, or zero otherwise.

The temperature pre-factor for the viscosity formula above is defined as

$$\tau(T) = H\left(e^{-\beta(T-T_0)/T_0}\right), \qquad H(x) = \begin{cases} 10^{-2} & \text{if } x < 10^{-2}, \\ x & \text{if } 10^{-2} \le x \le 10^{2}, \\ 10^{2} & \text{if } x > 10^{2}, \end{cases} \tag{31}$$

where β corresponds to the input parameter "Thermal viscosity exponent" and T_0 to the parameter "Reference temperature". If you set $T_0 = 0$ in the input file, the thermal pre-factor $\tau(T) = 1$.

The compositional pre-factor for the viscosity is defined as

$$\zeta(\mathfrak{c}) = \xi^{c_0} \tag{32}$$

if the model has compositional fields and equals one otherwise. ξ corresponds to the parameter "Composition viscosity prefactor" in the input file.

Finally, in the formula for the density, α corresponds to the "Thermal expansion coefficient" and $\Delta\rho$ corresponds to the parameter "Density differential for compositional field 1".

Note that this model uses the formulation that assumes an incompressible medium despite the fact that the density follows the law $\rho(T) = \rho_0(1-\alpha(T-T_{\text{ref}}))$.

> **Note:** Despite its name, this material model is not exactly "simple", as indicated by the formulas above. While it was originally intended to be simple, it has over time acquired all sorts of temperature and compositional dependencies that weren't initially intended. Consequently, there is now a "simpler" material model that now fills the role the current model was originally intended to fill.

'simple compressible': A material model that has constant values for all coefficients but the density. The defaults for all coefficients are chosen to be similar to what is believed to be correct for Earth's mantle. All of the values that define this model are read from a section "Material model/Simple compressible model" in the input file, see Section 5.78.

This model uses the following equations for the density:

$$\rho(p,T) = \rho_0\,(1-\alpha(T-T_a))\exp\beta(P-P_0)) \tag{33}$$

'simpler': A material model that has constant values except for density, which depends linearly on temperature:

$$\rho(p,T) = (1-\alpha(T-T_0))\,\rho_0. \tag{34}$$

> **Note:** This material model fills the role the "simple" material model was originally intended to fill, before the latter acquired all sorts of complicated temperature and compositional dependencies.

Possible values: [Selection Morency and Doin—Steinberger—averaging—composition reaction—depth dependent—diffusion dislocation—drucker prager—latent heat—latent heat melt—multicomponent—simple—simple compressible—simpler—unspecified]

5.67 Parameters in section `Material model/Averaging`

- *Parameter name:* `Averaging operation`

  Value: none

  Default: none

  Description: Chose the averaging operation to use.

  Possible values: [Selection none—arithmetic average—harmonic average—geometric average—pick largest—log average—nwd arithmetic average—nwd harmonic average—nwd geometric average]

- *Parameter name:* `Base model`

  Value: simple

  Default: simple

  Description: The name of a material model that will be modified by anaveraging operation. Valid values for this parameter are the names of models that are also valid for the "Material models/Model name" parameter. See the documentation for that for more information.

  Possible values: [Selection Morency and Doin—Steinberger—averaging—composition reaction—depth dependent—diffusion dislocation—drucker prager—latent heat—latent heat melt—multicomponent—simple—simple compressible—simpler]

- *Parameter name:* `Bell shape limit`

  Value: 1

  Default: 1

  Description: The limit normalized distance between 0 and 1 where the bell shape becomes zero. See the manual for a more information.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.68 Parameters in section `Material model/Composition reaction model`

- *Parameter name:* `Composition viscosity prefactor 1`

  Value: 1.0

  Default: 1.0

  Description: A linear dependency of viscosity on the first compositional field. Dimensionless prefactor. With a value of 1.0 (the default) the viscosity does not depend on the composition.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Composition viscosity prefactor 2`

  Value: 1.0

  Default: 1.0

  Description: A linear dependency of viscosity on the second compositional field. Dimensionless prefactor. With a value of 1.0 (the default) the viscosity does not depend on the composition.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Density differential for compositional field 1`

  Value: 0

  Default: 0

  Description: If compositional fields are used, then one would frequently want to make the density depend on these fields. In this simple material model, we make the following assumptions: if no compositional fields are used in the current simulation, then the density is simply the usual one with its linear dependence on the temperature. If there are compositional fields, then the density only depends on the first and the second one in such a way that the density has an additional term of the kind $+\Delta\rho\, c_1(\mathbf{x})$. This parameter describes the value of $\Delta\rho$ for the first field. Units: kg/m^3/unit change in composition.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Density differential for compositional field 2`

  Value: 0

  Default: 0

  Description: If compositional fields are used, then one would frequently want to make the density depend on these fields. In this simple material model, we make the following assumptions: if no compositional fields are used in the current simulation, then the density is simply the usual one with its linear dependence on the temperature. If there are compositional fields, then the density only depends on the first and the second one in such a way that the density has an additional term of the kind $+\Delta\rho\, c_1(\mathbf{x})$. This parameter describes the value of $\Delta\rho$ for the second field. Units: kg/m^3/unit change in composition.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Reaction depth`

  Value: 0

  Default: 0

  Description: Above this depth the compositional fields react: The first field gets converted to the second field. Units: m.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference density`

  Value: 3300

  Default: 3300

  Description: Reference density ρ_0. Units: kg/m^3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference specific heat`

  Value: 1250

  Default: 1250

  Description: The value of the specific heat cp. Units: $J/kg/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: The reference temperature T_0. Units: K.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal conductivity`

  Value: 4.7

  Default: 4.7

  Description: The value of the thermal conductivity k. Units: $W/m/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient β. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal viscosity exponent`

  Value: 0.0

  Default: 0.0

  Description: The temperature dependence of viscosity. Dimensionless exponent.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Viscosity`

  Value: 5e24

  Default: 5e24

  Description: The value of the constant viscosity. Units: $kg/m/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.69 Parameters in section `Material model/Depth dependent model`

- *Parameter name:* `Base model`

  Value: simple

  Default: simple

  Description: The name of a material model that will be modified by a depth dependent viscosity. Valid values for this parameter are the names of models that are also valid for the "Material models/Model name" parameter. See the documentation for that for more information.

  Possible values: [Selection Morency and Doin—Steinberger—averaging—composition reaction—depth dependent—diffusion dislocation—drucker prager—latent heat—latent heat melt—multicomponent—simple—simple compressible—simpler]

- *Parameter name:* `Data directory`

  Value: ./

  Default: ./

  Description: The path to the model data. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Depth dependence method`

  Value: None

  Default: None

  Description: Method that is used to specify how the viscosity should vary with depth.

  Possible values: [Selection Function—File—List—None]

- *Parameter name:* `Depth list`

  Value:

  Default:

  Description: A comma-separated list of depth values for use with the "List" "Depth dependence method". The list must be provided in order ofincreasing depth, and the last value must be greater than or equal to the maximal depth of the model. The depth list is interpreted as a layered viscosity structure and the depth values specify the maximum depths of each layer.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Viscosity depth file`

  Value: visc-depth.txt

  Default: visc-depth.txt

  Description: The name of the file containing depth-dependent viscosity data.

  Possible values: [Anything]

- *Parameter name:* `Viscosity list`

  Value:

  Default:

  Description: A comma-separated list of viscosity values, corresponding to the depth values provided in "Depth list". The number of viscosity values specified here must be the same as the number of depths provided in "Depth list"

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.70 Parameters in section `Material model/Depth dependent model/Viscosity depth function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 1.0e21

  Default: 1.0e21

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,t

  Default: x,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.71 Parameters in section `Material model/Diffusion dislocation`

- *Parameter name:* `Activation energies for diffusion creep`

  Value: 375e3

  Default: 375e3

  Description: List of activation energies, E_a, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: J/mol

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Activation energies for dislocation creep`

  Value: 530e3

  Default: 530e3

  Description: List of activation energies, E_a, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: J/mol

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Activation volumes for diffusion creep`

  Value: 6e-6

  Default: 6e-6

  Description: List of activation volumes, V_a, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one value is given, then all use the same value. Units: m^3/mol

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Activation volumes for dislocation creep`

  Value: 1.4e-5

  Default: 1.4e-5

  Description: List of activation volumes, V_a, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one value is given, then all use the same value. Units: m^3/mol

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Densities`

  Value: 3300.

  Default: 3300.

  Description: List of densities, ρ, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: kg/m^3

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Effective viscosity coefficient`

  Value: 1.0

  Default: 1.0

  Description: Scaling coefficient for effective viscosity.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Grain size`

  Value: 1e-3

  Default: 1e-3

  Description: Units: m

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Grain size exponents for diffusion creep`

  Value: 3

  Default: 3

  Description: List of grain size exponents, $m_diffusion$, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: None

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Heat capacity`

  Value: 1.25e3

  Default: 1.25e3

  Description: Units: $J/(K*kg)$

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum strain rate ratio iterations`

  Value: 40

  Default: 40

  Description: Maximum number of iterations to find the correct diffusion/dislocation strain rate ratio.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Maximum viscosity`

  Value: 1e28

  Default: 1e28

  Description: Upper cutoff for effective viscosity. Units: Pas

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum strain rate`

  Value: 1.4e-20

  Default: 1.4e-20

  Description: Stabilizes strain dependent viscosity. Units: $1/s$

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum viscosity`

  Value: 1e17

  Default: 1e17

  Description: Lower cutoff for effective viscosity. Units: Pas

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Prefactors for diffusion creep`

  Value: 1.5e-15

  Default: 1.5e-15

  Description: List of viscosity prefactors, A, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: $Pa^{-n_{diffusion}} m^{n_{diffusion}/m_{diffusion}} s^{-1}$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Prefactors for dislocation creep`

  Value: 1.1e-16

  Default: 1.1e-16

  Description: List of viscosity prefactors, A, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: $Pa^{-n_{dislocation}} m^{n_{dislocation}/m_{dislocation}} s^{-1}$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: For calculating density by thermal expansivity. Units: K

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference viscosity`

  Value: 1e22

  Default: 1e22

  Description: Reference viscosity for nondimensionalization. Units Pas

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Strain rate residual tolerance`

  Value: 1e-22

  Default: 1e-22

  Description: Tolerance for correct diffusion/dislocation strain rate ratio.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Stress exponents for diffusion creep`

  Value: 1

  Default: 1

  Description: List of stress exponents, $n_diffusion$, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: None

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Stress exponents for dislocation creep`

  Value: 3.5

  Default: 3.5

  Description: List of stress exponents, $n_dislocation$, for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: None

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Thermal diffusivity`

  Value: 0.8e-6

  Default: 0.8e-6

  Description: Units: m^2/s

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansivities`

  Value: 3.5e-5

  Default: 3.5e-5

  Description: List of thermal expansivities for background mantle and compositional fields, for a total of N+1 values, where N is the number of compositional fields. If only one values is given, then all use the same value. Units: $1/K$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Viscosity averaging scheme`

  Value: harmonic

  Default: harmonic

  Description: When more than one compositional field is present at a point with different viscosities, we need to come up with an average viscosity at that point. Select a weighted harmonic, arithmetic, geometric, or maximum composition.

  Possible values: [Selection arithmetic—harmonic—geometric—maximum composition]

5.72 Parameters in section `Material model/Drucker Prager`

- *Parameter name:* `Reference density`

  Value: 3300

  Default: 3300

  Description: The reference density ρ_0. Units: kg/m^3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference specific heat`

  Value: 1250

  Default: 1250

  Description: The value of the specific heat cp. Units: $J/kg/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: The reference temperature T_0. The reference temperature is used in the density calculation. Units: K.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference viscosity`

  Value: 1e22

  Default: 1e22

  Description: The value of the reference viscosity η_0. Units: $kg/m/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal conductivity`

  Value: 4.7

  Default: 4.7

  Description: The value of the thermal conductivity k. Units: $W/m/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient β. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.73 Parameters in section `Material model/Drucker Prager/Viscosity`

- *Parameter name:* `Angle of internal friction`

  Value: 0

  Default: 0

  Description: The value of the angle of internal friction ϕ. For a value of zero, in 2D the von Mises criterion is retrieved. Angles higher than 30 degrees are harder to solve numerically. Units: degrees.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Cohesion`

  Value: 2e7

  Default: 2e7

  Description: The value of the cohesion C. Units: Pa.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum viscosity`

  Value: 1e24

  Default: 1e24

  Description: The value of the maximum viscosity cutoff η_max. Units: $Pa\ s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum viscosity`

  Value: 1e19

  Default: 1e19

  Description: The value of the minimum viscosity cutoff η_min. Units: $Pa\ s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference strain rate`

  Value: 1e-15

  Default: 1e-15

  Description: The value of the initial strain rate prescribed during the first nonlinear iteration $\dot{\epsilon}_ref$. Units: $1/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.74 Parameters in section `Material model/Latent heat`

- *Parameter name:* `Composition viscosity prefactor`

  Value: 1.0

  Default: 1.0

  Description: A linear dependency of viscosity on composition. Dimensionless prefactor.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Compressibility`

  Value: 5.124e-12

  Default: 5.124e-12

  Description: The value of the compressibility κ. Units: $1/Pa$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Corresponding phase for density jump`

  Value:

  Default:

  Description: A list of phases, which correspond to the Phase transition density jumps. The density jumps occur only in the phase that is given by this phase value. 0 stands for the 1st compositional fields, 1 for the second compositional field and -1 for none of them. List must have the same number of entries as Phase transition depths. Units: Pa/K.

  Possible values: [List list of [Integer range 0...2147483647 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Define transition by depth instead of pressure`

  Value: true

  Default: true

  Description: Whether to list phase transitions by depth or pressure. If this parameter is true,then the input file will use Phase transitions depths and Phase transition widthsto define the phase transition. If it is false, the parameter file will read inphase transition data from Phase transition pressures andPhase transition pressure widths.

  Possible values: [Bool]

- *Parameter name:* `Density differential for compositional field 1`

  Value: 0

  Default: 0

  Description: If compositional fields are used, then one would frequently want to make the density depend on these fields. In this simple material model, we make the following assumptions: if no compositional fields are used in the current simulation, then the density is simply the usual one with its linear dependence on the temperature. If there are compositional fields, then the density only depends on the first one in such a way that the density has an additional term of the kind $+\Delta\rho \, c_1(\mathbf{x})$. This parameter describes the value of $\Delta\rho$. Units: kg/m^3/unit change in composition.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Phase transition Clapeyron slopes`

  Value:

  Default:

  Description: A list of Clapeyron slopes for each phase transition. A positive Clapeyron slope indicates that the phase transition will occur in a greater depth, if the temperature is higher than the one given in Phase transition temperatures and in a smaller depth, if the temperature is smaller than the one given in Phase transition temperatures. For negative slopes the other way round. List must have the same number of entries as Phase transition depths. Units: Pa/K.

  Possible values: [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Phase transition density jumps`

  Value:

  Default:

  Description: A list of density jumps at each phase transition. A positive value means that the density increases with depth. The corresponding entry in Corresponding phase for density jump determines

if the density jump occurs in peridotite, eclogite or none of them.List must have the same number of entries as Phase transition depths. Units: kg/m^3.

Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Phase transition depths`

  Value:

  Default:

  Description: A list of depths where phase transitions occur. Values must monotonically increase. Units: m.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Phase transition pressure widths`

  Value:

  Default:

  Description: A list of widths for each phase transition, in terms of pressure. The phase functions are scaled with these values, leading to a jump betwen phases for a value of zero and a gradual transition for larger values. List must have the same number of entries as Phase transition pressures. Define transition by depth instead of pressure must be set to falseto use this parameter.Units: Pa.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Phase transition pressures`

  Value:

  Default:

  Description: A list of pressures where phase transitions occur. Values must monotonically increase. Define transition by depth instead ofpressure must be set to false to use this parameter.Units: Pa.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Phase transition temperatures`

  Value:

  Default:

  Description: A list of temperatures where phase transitions occur. Higher or lower temperatures lead to phase transition ocurring in smaller or greater depths than given in Phase transition depths, depending on the Clapeyron slope given in Phase transition Clapeyron slopes. List must have the same number of entries as Phase transition depths. Units: K.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Phase transition widths`

  Value:

  Default:

  Description: A list of widths for each phase transition, in terms of depth. The phase functions are scaled with these values, leading to a jump betwen phases for a value of zero and a gradual transition for larger values. List must have the same number of entries as Phase transition depths. Units: m.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Reference density`
  Value: 3300
  Default: 3300
  Description: Reference density ρ_0. Units: kg/m^3.
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `Reference specific heat`
  Value: 1250
  Default: 1250
  Description: The value of the specific heat cp. Units: $J/kg/K$.
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `Reference temperature`
  Value: 293
  Default: 293
  Description: The reference temperature T_0. Units: K.
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `Thermal conductivity`
  Value: 2.38
  Default: 2.38
  Description: The value of the thermal conductivity k. Units: $W/m/K$.
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `Thermal expansion coefficient`
  Value: 4e-5
  Default: 4e-5
  Description: The value of the thermal expansion coefficient β. Units: $1/K$.
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `Thermal viscosity exponent`
  Value: 0.0
  Default: 0.0
  Description: The temperature dependence of viscosity. Dimensionless exponent.
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `Viscosity`
  Value: 5e24
  Default: 5e24
  Description: The value of the constant viscosity. Units: $kg/m/s$.
  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Viscosity prefactors`

  Value:

  Default:

  Description: A list of prefactors for the viscosity for each phase. The reference viscosity will be multiplied by this factor to get the corresponding viscosity for each phase. List must have one more entry than Phase transition depths. Units: non-dimensional.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.75 Parameters in section `Material model/Latent heat melt`

- *Parameter name:* `A1`

  Value: 1085.7

  Default: 1085.7

  Description: Constant parameter in the quadratic function that approximates the solidus of peridotite. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `A2`

  Value: 1.329e-7

  Default: 1.329e-7

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the solidus of peridotite. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `A3`

  Value: -5.1e-18

  Default: -5.1e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the solidus of peridotite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `B1`

  Value: 1475.0

  Default: 1475.0

  Description: Constant parameter in the quadratic function that approximates the lherzolite liquidus used for calculating the fraction of peridotite-derived melt. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `B2`

  Value: 8.0e-8

  Default: 8.0e-8

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the lherzolite liquidus used for calculating the fraction of peridotite-derived melt. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `B3`

  Value: -3.2e-18

  Default: -3.2e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the lherzolite liquidus used for calculating the fraction of peridotite-derived melt. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `C1`

  Value: 1780.0

  Default: 1780.0

  Description: Constant parameter in the quadratic function that approximates the liquidus of peridotite. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `C2`

  Value: 4.50e-8

  Default: 4.50e-8

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the liquidus of peridotite. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `C3`

  Value: -2.0e-18

  Default: -2.0e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the liquidus of peridotite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Composition viscosity prefactor`

  Value: 1.0

  Default: 1.0

  Description: A linear dependency of viscosity on composition. Dimensionless prefactor.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Compressibility`

  Value: 5.124e-12

  Default: 5.124e-12

  Description: The value of the compressibility κ. Units: $1/Pa$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `D1`

  Value: 976.0

  Default: 976.0

  Description: Constant parameter in the quadratic function that approximates the solidus of pyroxenite. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `D2`

  Value: 1.329e-7

  Default: 1.329e-7

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the solidus of pyroxenite. Note that this factor is different from the value given in Sobolev, 2011, because they use the potential temperature whereas we use the absolute temperature. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `D3`

  Value: -5.1e-18

  Default: -5.1e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the solidus of pyroxenite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Density differential for compositional field 1`

  Value: 0

  Default: 0

  Description: If compositional fields are used, then one would frequently want to make the density depend on these fields. In this simple material model, we make the following assumptions: if no compositional fields are used in the current simulation, then the density is simply the usual one with its linear dependence on the temperature. If there are compositional fields, then the density only depends on the first one in such a way that the density has an additional term of the kind $+\Delta\rho \, c_1(\mathbf{x})$. This parameter describes the value of $\Delta\rho$. Units: kg/m^3/unit change in composition.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `E1`

  Value: 663.8

  Default: 663.8

  Description: Prefactor of the linear depletion term in the quadratic function that approximates the melt fraction of pyroxenite. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `E2`

  Value: -611.4

  Default: -611.4

  Description: Prefactor of the quadratic depletion term in the quadratic function that approximates the melt fraction of pyroxenite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Mass fraction cpx`

  Value: 0.15

  Default: 0.15

  Description: Mass fraction of clinopyroxene in the peridotite to be molten. Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum pyroxenite melt fraction`

  Value: 0.5429

  Default: 0.5429

  Description: Maximum melt fraction of pyroxenite in this parameterization. At higher temperatures peridotite begins to melt.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Peridotite melting entropy change`

  Value: -300

  Default: -300

  Description: The entropy change for the phase transition from solid to melt of peridotite. Units: $J/(kgK)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Pyroxenite melting entropy change`

  Value: -400

  Default: -400

  Description: The entropy change for the phase transition from solid to melt of pyroxenite. Units: $J/(kgK)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference density`

  Value: 3300

  Default: 3300

  Description: Reference density ρ_0. Units: kg/m^3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference specific heat`

  Value: 1250

  Default: 1250

  Description: The value of the specific heat *cp*. Units: $J/kg/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: The reference temperature T_0. Units: K.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Relative density of melt`

  Value: 0.9

  Default: 0.9

  Description: The relative density of melt compared to the solid material. This means, the density change upon melting is this parameter times the density of solid material.Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal conductivity`

  Value: 2.38

  Default: 2.38

  Description: The value of the thermal conductivity k. Units: $W/m/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient`

  Value: 4e-5

  Default: 4e-5

  Description: The value of the thermal expansion coefficient α_s. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient of melt`

  Value: 6.8e-5

  Default: 6.8e-5

  Description: The value of the thermal expansion coefficient α_f. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal viscosity exponent`

  Value: 0.0

  Default: 0.0

  Description: The temperature dependence of viscosity. Dimensionless exponent.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Viscosity`

  Value: 5e24

  Default: 5e24

  Description: The value of the constant viscosity. Units: $kg/m/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `beta`

  Value: 1.5

  Default: 1.5

  Description: Exponent of the melting temperature in the melt fraction calculation. Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `r1`

  Value: 0.5

  Default: 0.5

  Description: Constant in the linear function that approximates the clinopyroxene reaction coefficient. Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `r2`

  Value: 8e-11

  Default: 8e-11

  Description: Prefactor of the linear pressure term in the linear function that approximates the clinopyroxene reaction coefficient. Units: $1/Pa$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.76 Parameters in section `Material model/Morency and Doin`

- *Parameter name:* `Activation energies`

  Value: 500

  Default: 500

  Description: List of activation energies, E_a, for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one values is given, then all use the same value. Units: kJ/mol

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Activation volume`

  Value: 6.4e-6

  Default: 6.4e-6

  Description: (V_a). Units: m^3/mol

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Coefficient of yield stress increase with depth`

  Value: 0.25

  Default: 0.25

  Description: (γ). Units: None

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Cohesive strength of rocks at the surface`

  Value: 117

  Default: 117

  Description: (τ_0). Units: Pa

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Densities`

  Value: 3300.

  Default: 3300.

  Description: List of densities, ρ, for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one values is given, then all use the same value. Units: kg/m^3

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Heat capacity`

  Value: 1.25e3

  Default: 1.25e3

  Description: Units: $J/(K * kg)$

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum strain rate`

  Value: 1.4e-20

  Default: 1.4e-20

  Description: Stabilizes strain dependent viscosity. Units: $1/s$

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Preexponential constant for viscous rheology law`

  Value: 1.24e14

  Default: 1.24e14

  Description: (B). Units: None

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference strain rate`

  Value: 6.4e-16

  Default: 6.4e-16

  Description: $(\dot{\epsilon}_{ref})$. Units: $1/s$

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: For calculating density by thermal expansivity. Units: K

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference viscosity`

  Value: 1e22

  Default: 1e22

  Description: Reference viscosity for nondimensionalization.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Stress exponents for plastic rheology`

  Value: 30

  Default: 30

  Description: List of stress exponents, n_p, for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one values is given, then all use the same value. Units: None

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Stress exponents for viscous rheology`

  Value: 3

  Default: 3

  Description: List of stress exponents, n_v, for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one values is given, then all use the same value. Units: None

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Thermal diffusivity`

  Value: 0.8e-6

  Default: 0.8e-6

  Description: Units: m^2/s

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansivities`

  Value: 3.5e-5

  Default: 3.5e-5

  Description: List of thermal expansivities for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one values is given, then all use the same value. Units: $1/K$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.77 Parameters in section `Material model/Multicomponent`

- *Parameter name:* `Densities`

  Value: 3300.

  Default: 3300.

  Description: List of densities for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one value is given, then all use the same value. Units: kg/m^3

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: The reference temperature T_0. Units: K.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Specific heats`

  Value: 1250.

  Default: 1250.

  Description: List of specific heats for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one value is given, then all use the same value. Units: $J/kg/K$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Thermal conductivities`

  Value: 4.7

  Default: 4.7

  Description: List of thermal conductivities for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one value is given, then all use the same value. Units: $W/m/K$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Thermal expansivities`

  Value: 4.e-5

  Default: 4.e-5

  Description: List of thermal expansivities for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one value is given, then all use the same value. Units: $1/K$

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Viscosities`

  Value: 1.e21

  Default: 1.e21

  Description: List of viscosities for background mantle and compositional fields,for a total of N+1 values, where N is the number of compositional fields.If only one value is given, then all use the same value. Units: Pas

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Viscosity averaging scheme`

  Value: harmonic

  Default: harmonic

  Description: When more than one compositional field is present at a point with different viscosities, we need to come up with an average viscosity at that point. Select a weighted harmonic, arithmetic, geometric, or maximum composition.

  Possible values: [Selection arithmetic—harmonic—geometric—maximum composition]

5.78 Parameters in section `Material model/Simple compressible model`

- *Parameter name:* `Reference compressibility`

  Value: 4e-12

  Default: 4e-12

  Description: The value of the reference compressibility. Units: $1/Pa$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference density`

  Value: 3300

  Default: 3300

  Description: Reference density ρ_0. Units: kg/m^3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference specific heat`

  Value: 1250

  Default: 1250

  Description: The value of the specific heat cp. Units: $J/kg/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal conductivity`

  Value: 4.7

  Default: 4.7

  Description: The value of the thermal conductivity k. Units: $W/m/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient α. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Viscosity`

  Value: 1e21

  Default: 1e21

  Description: The value of the constant viscosity η_0. Units: $kg/m/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.79 Parameters in section `Material model/Simple model`

- *Parameter name:* `Composition viscosity prefactor`

  Value: 1.0

  Default: 1.0

  Description: A linear dependency of viscosity on the first compositional field. Dimensionless prefactor. With a value of 1.0 (the default) the viscosity does not depend on the composition. See the general documentation of this model for a formula that states the dependence of the viscosity on this factor, which is called ξ there.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Density differential for compositional field 1`

  Value: 0

  Default: 0

  Description: If compositional fields are used, then one would frequently want to make the density depend on these fields. In this simple material model, we make the following assumptions: if no compositional fields are used in the current simulation, then the density is simply the usual one with its linear dependence on the temperature. If there are compositional fields, then the density only depends on the first one in such a way that the density has an additional term of the kind $+\Delta\rho\, c_1(\mathbf{x})$. This parameter describes the value of $\Delta\rho$. Units: kg/m^3/unit change in composition.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference density`

  Value: 3300

  Default: 3300

  Description: Reference density ρ_0. Units: kg/m^3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference specific heat`

  Value: 1250

  Default: 1250

  Description: The value of the specific heat cp. Units: $J/kg/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: The reference temperature T_0. The reference temperature is used in both the density and viscosity formulas. Units: K.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal conductivity`

  Value: 4.7

  Default: 4.7

  Description: The value of the thermal conductivity k. Units: $W/m/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient α. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal viscosity exponent`

  Value: 0.0

  Default: 0.0

  Description: The temperature dependence of viscosity. Dimensionless exponent. See the general documentation of this model for a formula that states the dependence of the viscosity on this factor, which is called β there.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Viscosity`

  Value: 5e24

  Default: 5e24

  Description: The value of the constant viscosity η_0. This viscosity may be modified by both temperature and compositional dependencies. Units: $kg/m/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.80 Parameters in section `Material model/Simpler model`

- *Parameter name:* `Reference density`

  Value: 3300

  Default: 3300

  Description: Reference density ρ_0. Units: kg/m^3.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference specific heat`

  Value: 1250

  Default: 1250

  Description: The value of the specific heat cp. Units: $J/kg/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Reference temperature`

  Value: 293

  Default: 293

  Description: The reference temperature T_0. The reference temperature is used in the density formula. Units: K.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal conductivity`

  Value: 4.7

  Default: 4.7

  Description: The value of the thermal conductivity k. Units: $W/m/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Thermal expansion coefficient`

  Value: 2e-5

  Default: 2e-5

  Description: The value of the thermal expansion coefficient β. Units: $1/K$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Viscosity`

  Value: 5e24

  Default: 5e24

  Description: The value of the viscosity η. Units: $kg/m/s$.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.81 Parameters in section `Material model/Steinberger model`

- *Parameter name:* `Bilinear interpolation`

  Value: true

  Default: true

  Description: Whether to use bilinear interpolation to compute material properties (slower but more accurate).

  Possible values: [Bool]

- *Parameter name:* `Compressible`

  Value: false

  Default: false

  Description: Whether to include a compressible material description.For a description see the manual section.

  Possible values: [Bool]

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/material-model/steinberger/

  Default: $ASPECT_SOURCE_DIR/data/material-model/steinberger/

  Description: The path to the model data. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Latent heat`

  Value: false

  Default: false

  Description: Whether to include latent heat effects in the calculation of thermal expansivity and specific heat. Following the approach of Nakagawa et al. 2009.

  Possible values: [Bool]

- *Parameter name:* `Lateral viscosity file name`

  Value: temp-viscosity-prefactor.txt

  Default: temp-viscosity-prefactor.txt

  Description: The file name of the lateral viscosity data.

  Possible values: [Anything]

- *Parameter name:* `Material file names`

  Value: pyr-ringwood88.txt

  Default: pyr-ringwood88.txt

  Description: The file names of the material data. List with as many components as active compositional fields (material data is assumed to be in order with the ordering of the fields).

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Maximum lateral viscosity variation`

  Value: 1e2

  Default: 1e2

  Description: The relative cutoff value for lateral viscosity variations caused by temperature deviations. The viscosity may vary laterally by this factor squared.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum viscosity`

  Value: 1e23

  Default: 1e23

  Description: The maximum viscosity that is allowed in the viscosity calculation. Larger values will be cut off.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum viscosity`

  Value: 1e19

  Default: 1e19

  Description: The minimum viscosity that is allowed in the viscosity calculation. Smaller values will be cut off.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Radial viscosity file name`

  Value: radial-visc.txt

  Default: radial-visc.txt

  Description: The file name of the radial viscosity data.

  Possible values: [Anything]

- *Parameter name:* `Reference viscosity`

  Value: 1e23

  Default: 1e23

  Description: The reference viscosity that is used for pressure scaling.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Use lateral average temperature for viscosity`

  Value: true

  Default: true

  Description: Whether to use to use the laterally averaged temperature instead of the adiabatic temperature for the viscosity calculation. This ensures that the laterally averaged viscosities remain more or less constant over the model runtime. This behaviour might or might not be desired.

  Possible values: [Bool]

5.82 Parameters in section `Mesh refinement`

- *Parameter name:* `Additional refinement times`

  Value:

  Default:

  Description: A list of times so that if the end time of a time step is beyond this time, an additional round of mesh refinement is triggered. This is mostly useful to make sure we can get through the initial transient phase of a simulation on a relatively coarse mesh, and then refine again when we are in a time range that we are interested in and where we would like to use a finer mesh. Units: Each element of the list has units years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Coarsening fraction`

  Value: 0.05

  Default: 0.05

  Description: The fraction of cells with the smallest error that should be flagged for coarsening.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Initial adaptive refinement`

  Value: 2

  Default: 2

  Description: The number of adaptive refinement steps performed after initial global refinement but while still within the first time step.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Initial global refinement`

  Value: 2

  Default: 2

  Description: The number of global refinement steps performed on the initial coarse mesh, before the problem is first solved there.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Minimum refinement level`

  Value: 0

  Default: 0

  Description: The minimum refinement level each cell should have, and that can not be exceeded by coarsening. Should not be higher than the 'Initial global refinement' parameter.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Normalize individual refinement criteria`

  Value: true

  Default: true

  Description: If multiple refinement criteria are specified in the "Strategy" parameter, then they need to be combined somehow to form the final refinement indicators. This is done using the method described by the "Refinement criteria merge operation" parameter which can either operate on the

raw refinement indicators returned by each strategy (i.e., dimensional quantities) or using normalized values where the indicators of each strategy are first normalized to the interval $[0, 1]$ (which also makes them non-dimensional). This parameter determines whether this normalization will happen.

Possible values: [Bool]

- *Parameter name:* `Refinement criteria merge operation`

  Value: max

  Default: max

  Description: If multiple mesh refinement criteria are computed for each cell (by passing a list of more than element to the `Strategy` parameter in this section of the input file) then one will have to decide which one should win when deciding which cell to refine. The operation that selects from these competing criteria is the one that is selected here. The options are:

  - `plus`: Add the various error indicators together and refine those cells on which the sum of indicators is largest.
  - `max`: Take the maximum of the various error indicators and refine those cells on which the maximal indicators is largest.

  The refinement indicators computed by each strategy are modified by the "Normalize individual refinement criteria" and "Refinement criteria scale factors" parameters.

  Possible values: [Selection plus—max]

- *Parameter name:* `Refinement criteria scaling factors`

  Value:

  Default:

  Description: A list of scaling factors by which every individual refinement criterion will be multiplied by. If only a single refinement criterion is selected (using the "Strategy" parameter, then this parameter has no particular meaning. On the other hand, if multiple criteria are chosen, then these factors are used to weigh the various indicators relative to each other.

  If "Normalize individual refinement criteria" is set to true, then the criteria will first be normalized to the interval $[0, 1]$ and then multiplied by the factors specified here. You will likely want to choose the factors to be not too far from 1 in that case, say between 1 and 10, to avoid essentially disabling those criteria with small weights. On the other hand, if the criteria are not normalized to $[0, 1]$ using the parameter mentioned above, then the factors you specify here need to take into account the relative numerical size of refinement indicators (which in that case carry physical units).

  You can experimentally play with these scaling factors by choosing to output the refinement indicators into the graphical output of a run.

  If the list of indicators given in this parameter is empty, then this indicates that they should all be chosen equal to one. If the list is not empty then it needs to have as many entries as there are indicators chosen in the "Strategy" parameter.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Refinement fraction`

  Value: 0.3

  Default: 0.3

  Description: The fraction of cells with the largest error that should be flagged for refinement.

  Possible values: [Double 0...1 (inclusive)]

- *Parameter name:* `Run postprocessors on initial refinement`

  Value: false

  Default: false

  Description: Whether or not the postproccessors should be run at the end of each of ths initial adaptive refinement cycles at the of the simulation start.

  Possible values: [Bool]

- *Parameter name:* `Strategy`

  Value: thermal energy density

  Default: thermal energy density

  Description: A comma separated list of mesh refinement criteria that will be run whenever mesh refinement is required. The results of each of these criteria, i.e., the refinement indicators they produce for all the cells of the mesh will then be normalized to a range between zero and one and the results of different criteria will then be merged through the operation selected in this section.

  The following criteria are available:

  'boundary': A class that implements a mesh refinement criterion which always flags all cells on specified boundaries for refinement. This is useful to provide high accuracy for processes at or close to the edge of the model domain.

  To use this refinement criterion, you may want to combine it with other refinement criteria, setting the 'Normalize individual refinement criteria' flag and using the 'max' setting for 'Refinement criteria merge operation'.

  'composition': A mesh refinement criterion that computes refinement indicators from the compositional fields. If there is more than one compositional field, then it simply takes the sum of the indicators computed from each of the compositional field.

  'density': A mesh refinement criterion that computes refinement indicators from a field that describes the spatial variability of the density, ρ. Because this quantity may not be a continuous function (ρ and C_p may be discontinuous functions along discontinuities in the medium, for example due to phase changes), we approximate the gradient of this quantity to refine the mesh. The error indicator defined here takes the magnitude of the approximate gradient and scales it by $h_K^{1+d/2}$ where h_K is the diameter of each cell and d is the dimension. This scaling ensures that the error indicators converge to zero as $h_K \to 0$ even if the energy density is discontinuous, since the gradient of a discontinuous function grows like $1/h_K$.

  'maximum refinement function': A mesh refinement criterion that ensures a maximum refinement level described by an explicit formula with the depth or position as argument. Which coordinate representation is used is determined by an input parameter. Whatever the coordinate system chosen, the function you provide in the input file will by default depend on variables 'x', 'y' and 'z' (if in 3d). However, the meaning of these symbols depends on the coordinate system. In the Cartesian coordinate system, they simply refer to their natural meaning. If you have selected 'depth' for the coordinate system, then 'x' refers to the depth variable and 'y' and 'z' will simply always be zero. If you have selected a spherical coordinate system, then 'x' will refer to the radial distance of the point to the origin, 'y' to the azimuth angle and 'z' to the polar angle measured positive from the north pole. Note that the order of spherical coordinates is r,phi,theta and not r,theta,phi, since this allows for dimension independent expressions. Each coordinate system also includes a final 't' variable which represents the model time, evaluated in years if the 'Use years in output instead of seconds' parameter is set, otherwise evaluated in seconds. After evaluating the function, its values are rounded to the nearest integer.

  The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

'minimum refinement function': A mesh refinement criterion that ensures a minimum refinement level described by an explicit formula with the depth or position as argument. Which coordinate representation is used is determined by an input parameter. Whatever the coordinate system chosen, the function you provide in the input file will by default depend on variables 'x', 'y' and 'z' (if in 3d). However, the meaning of these symbols depends on the coordinate system. In the Cartesian coordinate system, they simply refer to their natural meaning. If you have selected 'depth' for the coordinate system, then 'x' refers to the depth variable and 'y' and 'z' will simply always be zero. If you have selected a spherical coordinate system, then 'x' will refer to the radial distance of the point to the origin, 'y' to the azimuth angle and 'z' to the polar angle measured positive from the north pole. Note that the order of spherical coordinates is r,phi,theta and not r,theta,phi, since this allows for dimension independent expressions. Each coordinate system also includes a final 't' variable which represents the model time, evaluated in years if the 'Use years in output instead of seconds' parameter is set, otherwise evaluated in seconds. After evaluating the function, its values are rounded to the nearest integer.

The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

'nonadiabatic temperature': A mesh refinement criterion that computes refinement indicators from the excess temperature(difference between temperature and adiabatic temperature.

'particle density': A mesh refinement criterion that computes refinement indicators based on the density of particles. In practice this plugin equilibrates the number of particles per cell, leading to fine cells in high particle density regions and coarse cells in low particle density regions. This plugin is mostly useful for models with inhomogeneous particle density, e.g. when tracking an initial interface with a high particle density, or when the spatial particle density denotes the region of interest. Additionally, this plugin tends to balance the computational load between processes in parallel computations, because the particle and mesh density is more aligned.

'slope': A class that implements a mesh refinement criterion intended for use with a free surface. It calculates a local slope based on the angle between the surface normal and the local gravity vector. Cells with larger angles are marked for refinement.

To use this refinement criterion, you may want to combine it with other refinement criteria, setting the 'Normalize individual refinement criteria' flag and using the 'max' setting for 'Refinement criteria merge operation'.

'strain rate': A mesh refinement criterion that computes therefinement indicators equal to the strain rate norm computed at the center of the elements.

'temperature': A mesh refinement criterion that computes refinement indicators from the temperature field.

'thermal energy density': A mesh refinement criterion that computes refinement indicators from a field that describes the spatial variability of the thermal energy density, $\rho C_p T$. Because this quantity may not be a continuous function (ρ and C_p may be discontinuous functions along discontinuities in the medium, for example due to phase changes), we approximate the gradient of this quantity to refine the mesh. The error indicator defined here takes the magnitude of the approximate gradient and scales it by $h_K^{1.5}$ where h_K is the diameter of each cell. This scaling ensures that the error indicators converge to zero as $h_K \to 0$ even if the energy density is discontinuous, since the gradient of a discontinuous function grows like $1/h_K$.

'topography': A class that implements a mesh refinement criterion, which always flags all cells in the uppermost layer for refinement. This is useful to provide high accuracy for processes at or close to the surface.

To use this refinement criterion, you may want to combine it with other refinement criteria, setting the 'Normalize individual refinement criteria' flag and using the 'max' setting for 'Refinement criteria merge operation'.

'velocity': A mesh refinement criterion that computes refinement indicators from the velocity field.

'viscosity': A mesh refinement criterion that computes refinement indicators from a field that describes the spatial variability of the logarithm of the viscosity, $\log \eta$. (We choose the logarithm of the viscosity because it can vary by orders of magnitude.)Because this quantity may not be a continuous function (η may be a discontinuous function along discontinuities in the medium, for example due to phase changes), we approximate the gradient of this quantity to refine the mesh. The error indicator defined here takes the magnitude of the approximate gradient and scales it by $h_K^{1+d/2}$ where h_K is the diameter of each cell and d is the dimension. This scaling ensures that the error indicators converge to zero as $h_K \to 0$ even if the energy density is discontinuous, since the gradient of a discontinuous function grows like $1/h_K$.

Possible values: [MultipleSelection boundary—composition—density—maximum refinement function—minimum refinement function—nonadiabatic temperature—particle density—slope—strain rate—temperature—thermal energy density—topography—velocity—viscosity]

- *Parameter name:* `Time steps between mesh refinement`

  Value: 10

  Default: 10

  Description: The number of time steps after which the mesh is to be adapted again based on computed error indicators. If 0 then the mesh will never be changed.

  Possible values: [Integer range 0...2147483647 (inclusive)]

5.83 Parameters in section `Mesh refinement/Boundary`

- *Parameter name:* `Boundary refinement indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries where there should be mesh refinement.

  The names of the boundaries listed here can either be numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

5.84 Parameters in section `Mesh refinement/Composition`

- *Parameter name:* `Compositional field scaling factors`

  Value:

  Default:

  Description: A list of scaling factors by which every individual compositional field will be multiplied by. If only a single compositional field exists, then this parameter has no particular meaning. On the other hand, if multiple criteria are chosen, then these factors are used to weigh the various indicators relative to each other.

  If the list of scaling factors given in this parameter is empty, then this indicates that they should all be chosen equal to one. If the list is not empty then it needs to have as many entries as there are compositional fields.

  Possible values: [List list of [Double 0...1.79769e+308 (inclusive)] of length 0...4294967295 (inclusive)]

5.85 Parameters in section `Mesh refinement/Maximum refinement function`

- *Parameter name:* `Coordinate system`

  Value: depth

  Default: depth

  Description: A selection that determines the assumed coordinate system for the function variables. Allowed values are 'depth', 'cartesian' and 'spherical'. 'depth' will create a function, in which only the first variable is non-zero, which is interpreted to be the depth of the point. 'spherical' coordinates are interpreted as r,phi or r,phi,theta in 2D/3D respectively with theta being the polar angle.

  Possible values: [Selection depth—cartesian—spherical]

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.86 Parameters in section `Mesh refinement/Minimum refinement function`

- *Parameter name:* `Coordinate system`

  Value: depth

  Default: depth

  Description: A selection that determines the assumed coordinate system for the function variables. Allowed values are 'depth', 'cartesian' and 'spherical'. 'depth' will create a function, in which only the first variable is non-zero, which is interpreted to be the depth of the point. 'spherical' coordinates are interpreted as r,phi or r,phi,theta in 2D/3D respectively with theta being the polar angle.

  Possible values: [Selection depth—cartesian—spherical]

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.87 Parameters in section `Model settings`

- *Parameter name:* `Fixed composition boundary indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries on which the composition is fixed and described by the boundary composition object selected in its own section of this input file. All boundary indicators used by the geometry but not explicitly listed here will end up with no-flux (insulating) boundary conditions.

  The names of the boundaries listed here can either by numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  This parameter only describes which boundaries have a fixed composition, but not what composition should hold on these boundaries. The latter piece of information needs to be implemented in a plugin in the BoundaryComposition group, unless an existing implementation in this group already provides what you want.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Fixed temperature boundary indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries on which the temperature is fixed and described by the boundary temperature object selected in its own section of this input file. All boundary indicators used by the geometry but not explicitly listed here will end up with no-flux (insulating) boundary conditions.

  The names of the boundaries listed here can either by numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  This parameter only describes which boundaries have a fixed temperature, but not what temperature should hold on these boundaries. The latter piece of information needs to be implemented in a plugin in the BoundaryTemperature group, unless an existing implementation in this group already provides what you want.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Free surface boundary indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries where there is a free surface. Set to nothing to disable all free surface computations.

  The names of the boundaries listed here can either by numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Include adiabatic heating`

  Value: false

  Default: false

  Description: Whether to include adiabatic heating into the model or not. From a physical viewpoint, adiabatic heating should always be used but may be undesirable when comparing results with known benchmarks that do not include this term in the temperature equation.Warning: deprecated! Add 'adiabatic heating' to the 'List of model names' instead.

  Possible values: [Bool]

- *Parameter name:* `Include latent heat`

  Value: false

  Default: false

  Description: Whether to include the generation of latent heat at phase transitions into the model or not. From a physical viewpoint, latent heat should always be used but may be undesirable when comparing results with known benchmarks that do not include this term in the temperature equation or when dealing with a model without phase transitions.Warning: deprecated! Add 'latent heat' to the 'List of model names' instead.

  Possible values: [Bool]

- *Parameter name:* `Include shear heating`

  Value: false

  Default: false

  Description: Whether to include shear heating into the model or not. From a physical viewpoint, shear heating should always be used but may be undesirable when comparing results with known benchmarks that do not include this term in the temperature equation.Warning: deprecated! Add 'shear heating' to the 'List of model names' instead.

  Possible values: [Bool]

- *Parameter name:* `Prescribed traction boundary indicators`

  Value:

  Default:

  Description: A comma separated list denoting those boundaries on which a traction force is prescribed, i.e., where known external forces act, resulting in an unknown velocity. This is often used to model "open" boundaries where we only know the pressure. This pressure then produces a force that is normal to the boundary and proportional to the pressure.

  The format of valid entries for this parameter is that of a map given as "key1 [selector]: value1, key2 [selector]: value2, key3: value3, ..." where each key must be a valid boundary indicator (which is either an integer or the symbolic name the geometry model in use may have provided for this part of the boundary) and each value must be one of the currently implemented boundary traction models. "selector" is an optional string given as a subset of the letters 'xyz' that allows you to apply the boundary conditions only to the components listed. As an example, '1 y: function' applies the type 'function' to the y component on boundary 1. Without a selector it will affect all components of the traction.

  Possible values: [Map map of [Anything]:[Selection function—zero traction] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Prescribed velocity boundary indicators`

  Value:

  Default:

  Description: A comma separated list denoting those boundaries on which the velocity is prescribed, i.e., where unknown external forces act to prescribe a particular velocity. This is often used to prescribe a velocity that equals that of overlying plates.

  The format of valid entries for this parameter is that of a map given as "key1 [selector]: value1, key2 [selector]: value2, key3: value3, ..." where each key must be a valid boundary indicator (which is either an integer or the symbolic name the geometry model in use may have provided for this part of the boundary) and each value must be one of the currently implemented boundary velocity models. "selector" is an optional string given as a subset of the letters 'xyz' that allows you to apply the boundary conditions only to the components listed. As an example, '1 y: function' applies the type 'function' to the y component on boundary 1. Without a selector it will affect all components of the velocity.

  Note that the no-slip boundary condition is a special case of the current one where the prescribed velocity happens to be zero. It can thus be implemented by indicating that a particular boundary is part of the ones selected using the current parameter and using "zero velocity" as the boundary values. Alternatively, you can simply list the part of the boundary on which the velocity is to be zero with the parameter "Zero velocity boundary indicator" in the current parameter section.

  Note that when "Use years in output instead of seconds" is set to true, velocity should be given in m/yr.

  Possible values: [Map map of [Anything]:[Selection ascii data—function—gplates—zero velocity] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Remove nullspace`

  Value:

  Default:

  Description: Choose none, one or several from

  - net rotation
  - angular momentum
  - net translation
  - linear momentum
  - net x translation
  - net y translation
  - net z translation
  - linear x momentum
  - linear y momentum
  - linear z momentum

  These are a selection of operations to remove certain parts of the nullspace from the velocity after solving. For some geometries and certain boundary conditions the velocity field is not uniquely determined but contains free translations and/or rotations. Depending on what you specify here, these non-determined modes will be removed from the velocity field at the end of the Stokes solve step.

  The "angular momentum" option removes a rotation such that the net angular momentum is zero. The "linear * momentum" options remove translations such that the net momentum in the relevant direction

is zero. The "net rotation" option removes the net rotation of the domain, and the "net * translation" options remove the net translations in the relevant directions. For most problems there should not be a significant difference between the momentum and rotation/translation versions of nullspace removal, although the momentum versions are more physically motivated. They are equivalent for constant density simulations, and approximately equivalent when the density variations are small.

Note that while more than one operation can be selected it only makes sense to pick one rotational and one translational operation.

Possible values: [MultipleSelection net rotation—angular momentum—net translation—linear momentum—net x translation—net y translation—net z translation—linear x momentum—linear y momentum—linear z momentum]

- *Parameter name:* `Tangential velocity boundary indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries on which the velocity is tangential and unrestrained, i.e., free-slip where no external forces act to prescribe a particular tangential velocity (although there is a force that requires the flow to be tangential).

  The names of the boundaries listed here can either by numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

- *Parameter name:* `Zero velocity boundary indicators`

  Value:

  Default:

  Description: A comma separated list of names denoting those boundaries on which the velocity is zero.

  The names of the boundaries listed here can either by numbers (in which case they correspond to the numerical boundary indicators assigned by the geometry object), or they can correspond to any of the symbolic names the geometry object may have provided for each part of the boundary. You may want to compare this with the documentation of the geometry model you use in your model.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

5.88 Parameters in section `Postprocess`

- *Parameter name:* `List of postprocessors`

  Value:

  Default:

  Description: A comma separated list of postprocessor objects that should be run at the end of each time step. Some of these postprocessors will declare their own parameters which may, for example, include that they will actually do something only every so many time steps or years. Alternatively, the text 'all' indicates that all available postprocessors should be run after each time step.

  The following postprocessors are available:

  'Stokes residual': A postprocessor that outputs the Stokes residuals during the iterative solver algorithm into a file stokes_residuals.txt in the output directory.

  'basic statistics': A postprocessor that computes some simplified statistics like the Rayleigh number and other quantities that only make sense in certain model setups.

'boundary densities': A postprocessor that computes the laterally averaged density at the top and bottom of the domain.

'boundary pressures': A postprocessor that computes the laterally averaged pressure at the top and bottom of the domain.

'command': A postprocessor that executes a command line process.

'composition statistics': A postprocessor that computes some statistics about the compositional fields, if present in this simulation. In particular, it computes maximal and minimal values of each field, as well as the total mass contained in this field as defined by the integral $m_i(t) = \int_\Omega c_i(\mathbf{x}, t)\, dx$.

'depth average': A postprocessor that computes depth averaged quantities and writes them into a file <depth_average.ext> in the output directory, where the extension of the file is determined by the output format you select. In addition to the output format, a number of other parameters also influence this postprocessor, and they can be set in the section `Postprocess/Depth average` in the input file.

In the output files, the x-value of each data point corresponds to the depth, whereas the y-value corresponds to the simulation time. The time is provided in seconds or, if the global "Use years in output instead of seconds" parameter is set, in years.

'dynamic topography': A postprocessor that computes a measure of dynamic topography based on the stress at the surface. The data is written into text files named 'dynamic_topography.NNNNN' in the output directory, where NNNNN is the number of the time step.

The exact approach works as follows: At the centers of all cells that sit along the top surface, we evaluate the stress and evaluate the component of it in the direction in which gravity acts. In other words, we compute $\sigma_{rr} = \hat{g}^T(2\eta\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{div }\mathbf{u})I)\hat{g} - p_d$ where $\hat{g} = \mathbf{g}/\|\mathbf{g}\|$ is the direction of the gravity vector $\mathbf{g}$ and $p_d = p - p_a$ is the dynamic pressure computed by subtracting the adiabatic pressure p_a from the total pressure p computed as part of the Stokes solve. From this, the dynamic topography is computed using the formula $h = \frac{\sigma_{rr}}{\|\mathbf{g}\|\rho}$ where ρ is the density at the cell center. The file format then consists of lines with Euclidean coordinates followed by the corresponding topography value.

(As a side note, the postprocessor chooses the cell center instead of the center of the cell face at the surface, where we really are interested in the quantity, since this often gives better accuracy. The results should in essence be the same, though.)

'heat flux statistics': A postprocessor that computes some statistics about the (conductive) heat flux across boundaries. For each boundary indicator (see your geometry description for which boundary indicators are used), the heat flux is computed in outward direction, i.e., from the domain to the outside, using the formula $\int_{\Gamma_i} k\nabla T \cdot \mathbf{n}$ where Γ_i is the part of the boundary with indicator i, k is the thermal conductivity as reported by the material model, T is the temperature, and $\mathbf{n}$ is the outward normal. Note that the quantity so computed does not include any energy transported across the boundary by material transport in cases where $\mathbf{u} \cdot \mathbf{n} \neq 0$.

As stated, this postprocessor computes the *outbound* heat flux. If you are interested in the opposite direction, for example from the core into the mantle when the domain describes the mantle, then you need to multiply the result by -1.

'heating statistics': A postprocessor that computes some statistics about heating, averaged by volume.

'mass flux statistics': A postprocessor that computes some statistics about the mass flux across boundaries. For each boundary indicator (see your geometry description for which boundary indicators are used), the mass flux is computed in outward direction, i.e., from the domain to the outside, using the formula $\int_{\Gamma_i} \rho\mathbf{v} \cdot \mathbf{n}$ where Γ_i is the part of the boundary with indicator i, ρ is the density as reported by the material model, $\mathbf{v}$ is the velocity, and $\mathbf{n}$ is the outward normal.

As stated, this postprocessor computes the *outbound* mass flux. If you are interested in the opposite direction, for example from the core into the mantle when the domain describes the mantle, then you need to multiply the result by -1.

‘point values’: A postprocessor that evaluates the solution (i.e., velocity, pressure, temperature, and compositional fields along with other fields that are treated as primary variables) at the end of every time step at a given set of points and then writes this data into the file ¡point_values.txt¿ in the output directory. The points at which the solution should be evaluated are specified in the section `Postprocess/Point values` in the input file.

In the output file, data is organized as (i) time, (ii) the 2 or 3 coordinates of the evaluation points, and (iii) followed by the values of the solution vector at this point. The time is provided in seconds or, if the global “Use years in output instead of seconds” parameter is set, in years. In the latter case, the velocity is also converted to meters/year, instead of meters/second.

> **Note:** Evaluating the solution of a finite element field at arbitrarily chosen points is an expensive process. Using this postprocessor will only be efficient if the number of evaluation points is relatively small. If you need a very large number of evaluation points, you should consider extracting this information from the visualization program you use to display the output of the ’visualization’ postprocessor.

‘pressure statistics’: A postprocessor that computes some statistics about the pressure field.

‘spherical velocity statistics’: A postprocessor that computes radial, tangential and total RMS velocity.

‘temperature statistics’: A postprocessor that computes some statistics about the temperature field.

‘topography’: A postprocessor intended for use with a free surface. After every step, it loops over all the vertices on the top surface and determines the maximum and minimum topography relative to a reference datum (initial box height for a box geometry model or initial radius for a sphere/spherical shell geometry model). Outputs topography in meters

‘tracers’: A Postprocessor that creates tracer particles that follow the velocity field of the simulation. The particles can be generated and propagated in various ways and they can carry a number of constant or time-varying properties. The postprocessor can write output positions and properties of all tracers at chosen intervals, although this is not mandatory. It also allows other parts of the code to query the tracers for information.

‘velocity boundary statistics’: A postprocessor that computes some statistics about the velocity along the boundaries. For each boundary indicator (see your geometry description for which boundary indicators are used), the min and max velocity magnitude is computed.

‘velocity statistics’: A postprocessor that computes some statistics about the velocity field.

‘viscous dissipation statistics’: A postprocessor that computes the viscous dissipationfor the whole domain as: $\frac{1}{2}\int_V \sigma : \dot{\epsilon} dV = \int_V (-p\nabla \cdot u + 2\mu\dot{\epsilon} : \dot{\epsilon} - \frac{2\mu}{3}(\nabla \cdot u)^2) dV$.

The information produced by this postprocessor is a subset of what is generated by the ’heating statistics’ postprocessor.

‘visualization’: A postprocessor that takes the solution and writes it into files that can be read by a graphical visualization program. Additional run time parameters are read from the parameter subsection ’Visualization’.

Possible values: [MultipleSelection Stokes residual—basic statistics—boundary densities—boundary pressures—command—composition statistics—depth average—dynamic topography—heat flux statistics—heating statistics—mass flux statistics—point values—pressure statistics—spherical velocity statistics—temperature statistics—topography—tracers—velocity boundary statistics—velocity statistics—viscous dissipation statistics—visualization]

5.89 Parameters in section `Postprocess/Command`

- *Parameter name:* `Command`

  Value:

  Default:

  Description: Command to execute.

  Possible values: [Anything]

- *Parameter name:* `Run on all processes`

  Value: false

  Default: false

  Description: Whether to run command from all processes (true), or only on process 0 (false).

  Possible values: [Bool]

- *Parameter name:* `Terminate on failure`

  Value: false

  Default: false

  Description: Select whether ASPECT should terminate if the command returns a non-zero exit status.

  Possible values: [Bool]

5.90 Parameters in section `Postprocess/Depth average`

- *Parameter name:* `List of output variables`

  Value: all

  Default: all

  Description: A comma separated list which specifies which quantites to average in each depth slice. It defaults to averaging all availabe quantities, but this can be an expensive operation, so you may want to select only a few.

  Possible values: [MultipleSelection all—temperature—composition—adiabatic temperature—velocity magnitude—sinking velocity—Vs—Vp—viscosity—vertical heat flux]

- *Parameter name:* `Number of zones`

  Value: 10

  Default: 10

  Description: The number of zones in depth direction within which we are to compute averages. By default, we subdivide the entire domain into 10 depth zones and compute temperature and other averages within each of these zones. However, if you have a very coarse mesh, it may not make much sense to subdivide the domain into so many zones and you may wish to choose less than this default. It may also make computations slightly faster. On the other hand, if you have an extremely highly resolved mesh, choosing more zones might also make sense.

  Possible values: [Integer range 1...2147483647 (inclusive)]

- *Parameter name:* `Output format`

  Value: gnuplot

  Default: gnuplot

Description: The format in which the output shall be produced. The format in which the output is generated also determines the extension of the file into which data is written.

Possible values: [Selection none—dx—ucd—gnuplot—povray—eps—gmv—tecplot—tecplot_binary—vtk—vtu—hdf5—
intermediate—txt]

- *Parameter name:* `Time between graphical output`

  Value: 1e8

  Default: 1e8

  Description: The time interval between each generation of graphical output files. A value of zero indicates that output should be generated in each time step. Units: years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.91 Parameters in section `Postprocess/Dynamic Topography`

- *Parameter name:* `Subtract mean of dynamic topography`

  Value: false

  Default: false

  Description: Option to remove the mean dynamic topography in the outputted data file (not visualization). 'true' subtracts the mean, 'false' leaves the calculated dynamic topography as is.

  Possible values: [Bool]

5.92 Parameters in section `Postprocess/Point values`

- *Parameter name:* `Evaluation points`

  Value:

  Default:

  Description: The list of points at which the solution should be evaluated. Points need to be separated by semicolons, and coordinates of each point need to be separated by commas.

  Possible values: [List list of [List list of [Double -1.79769e+308...1.79769e+308 (inclusive)] of length 2...2 (inclusive)] of length 0...4294967295 (inclusive) separated by ¡;¿]

5.93 Parameters in section `Postprocess/Tracers`

- *Parameter name:* `Data output format`

  Value: vtu

  Default: vtu

  Description: File format to output raw particle data in. If you select 'none' no output will be written.Select one of the following models:

  'ascii': This particle output plugin writes particle positions and properties into space separated ascii files.

  'hdf5': This particle output plugin writes particle positions and properties into hdf5 files.

  'vtu': This particle output plugin writes particle positions and properties into vtu files.

  Possible values: [Selection ascii—hdf5—vtu—none]

- *Parameter name:* `Integration scheme`

  Value: rk2

  Default: rk2

  Description: This parameter is used to decide which method to use to solve the equation that describes the position of particles, i.e., $\frac{d}{dt}\mathbf{x}_k(t) = \mathbf{u}(\mathbf{x}_k(t), t)$, where k is an index that runs over all particles, and $\mathbf{u}(\mathbf{x}, t)$ is the velocity field that results from the Stokes equations.

  In practice, the exact velocity $\mathbf{u}(\mathbf{x}, t)$ is of course not available, but only a numerical approximation $\mathbf{u}_h(\mathbf{x}, t)$. Furthermore, this approximation is only available at discrete time steps, $\mathbf{u}^n(\mathbf{x}) = \mathbf{u}(\mathbf{x}, t^n)$, and these need to be interpolated between time steps if the integrator for the equation above requires an evaluation at time points between the discrete time steps. If we denote this interpolation in time by $\tilde{\mathbf{u}}_h(\mathbf{x}, t)$ where $\tilde{\mathbf{u}}_h(\mathbf{x}, t^n) = \mathbf{u}^n(\mathbf{x})$, then the equation the differential equation solver really tries to solve is $\frac{d}{dt}\tilde{\mathbf{x}}_k(t) = \tilde{\mathbf{u}}_h(\mathbf{x}_k(t), t)$.

  As a consequence of these considerations, if you try to assess convergence properties of an ODE integrator – for example to verify that the RK4 integrator converges with forth order –, it is important to recall that the integrator may not solve the equation you think it solves. If, for example, we call the numerical solution of the ODE $\tilde{\mathbf{x}}_{k,h}(t)$, then the error will typically satisfy a relationship like

  $$\|\tilde{\mathbf{x}}_k(T) - \tilde{\mathbf{x}}_{k,h}(T)\| \leq C(T)\Delta t^p$$

  where Δt is the time step and p the convergence order of the method, and $C(T)$ is a (generally unknown) constant that depends on the end time T at which one compares the solutions. On the other hand, an analytically computed trajectory would likely use the *exact* velocity, and one may be tempted to compute $\|\mathbf{x}_k(T) - \tilde{\mathbf{x}}_{k,h}(T)\|$, but this quantity will, in the best case, only satisfy an estimate of the form

  $$\|\mathbf{x}_k(T) - \tilde{\mathbf{x}}_{k,h}(T)\| \leq C_1(T)\Delta t^p + C_2(T)\|\mathbf{u} - \mathbf{u}_h\| + C_3(T)\|\mathbf{u}_h - \tilde{\mathbf{u}}_h\|$$

  with appropriately chosen norms for the second and third term. These second and third terms typically converge to zero at relatively low rates (compared to the order p of the integrator, which can often be chosen relatively high) in the mesh size h and the time step size
  Deltat, limiting the overall accuracy of the ODE integrator.

  Select one of the following models:

  'euler': Explicit Euler scheme integrator, where $y_{n+1} = y_n + dt * v(y_n)$. This requires only one integration substep per timestep.

  'rk2': Second Order Runge Kutta integrator $y_{n+1} = y_n + dt * v(t_{n+1/2}, y_n + 0.5 * k_1)$ where $k_1 = y_n + 0.5 * dt * v(t_n, y_n)$

  'rk4': Runge Kutta fourth order integrator, where $y_{n+1} = y_n + (1/6) * k_1 + (1/3) * k_2 + (1/3) * k_3 + (1/6) * k_4$ and k_1, k_2, k_3, k_4 are defined as usual.

  Possible values: [Selection euler—rk2—rk4]

- *Parameter name:* `Interpolation scheme`

  Value: first particle

  Default: first particle

  Description: Select one of the following models:

  'first particle': Return the properties of the first tracer in the given cell.

  Possible values: [Selection first particle]

- *Parameter name:* `List of tracer properties`

  Value:

Default:

Description: A comma separated list of tracer properties that should be tracked. By default none is selected, which means only position, velocity and id of the tracers are output.

The following properties are available:

'function': Implementation of a model in which the tracer property is set by evaluating an explicit function at the initial position of each particle. The function is defined in the parameters in section "Tracers—Function". The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

'initial composition': Implementation of a plugin in which the tracer property is given as the initial composition at the particle's initial position. The tracer gets as many properties as there are compositional fields.

'initial position': Implementation of a plugin in which the tracer property is given as the initial position of the tracer.

'pT path': Implementation of a plugin in which the tracer property is defined as the current pressure and temperature at this position. This can be used to generate pressure-temperature paths of material points over time.

'position': Implementation of a plugin in which the tracer property is defined as the current position.

'velocity': Implementation of a plugin in which the tracer property is defined as the recent velocity at this position.

Possible values: [MultipleSelection function—initial composition—initial position—pT path—position—velocity]

- *Parameter name:* `Load balancing strategy`

  Value: none

  Default: none

  Description: Strategy that is used to balance the computationalload across processors for adaptive meshes.

  Possible values: [Selection none—remove particles—remove and add particles—repartition]

- *Parameter name:* `Maximum tracers per cell`

  Value: 100

  Default: 100

  Description: Upper limit for particle number per cell. This limit is useful for adaptive meshes to prevent coarse cells from slowing down the whole model. It will be checked and enforced after mesh refinement, after MPI transfer of particles and after particle movement. If there are `n_number_of_particles` > `max_particles_per_cell` particles in one cell then `n_number_of_particles - max_particles_per_cell` particles in this cell are randomly chosen and destroyed.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Minimum tracers per cell`

  Value: 0

  Default: 0

  Description: Lower limit for particle number per cell. This limit is useful for adaptive meshes to prevent fine cells from being empty of particles. It will be checked and enforced after mesh refinement and after particle movement. If there are `n_number_of_particles` < `min_particles_per_cell` particles in one cell then `min_particles_per_cell - n_number_of_particles` particles are generated and randomly

placed in this cell. If the particles carry properties the individual property plugins control how the properties of the new particles are initialized.

Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Number of tracers`

  Value: 1000

  Default: 1000

  Description: Total number of tracers to create (not per processor or per element). The number is parsed as a floating point number (so that one can specify, for example, '1e4' particles) but it is interpreted as an integer, of course.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Particle generator name`

  Value: random uniform

  Default: random uniform

  Description: Select one of the following models:

  'ascii file': Generates a distribution of tracers from coordinates specified in an Ascii data file. The file format is a simple text file, with as many columns as spatial dimensions and as many lines as tracers to be generated. Initial comment lines starting with '#' will be discarded.All of the values that define this generator are read from a section "Particle generator/Ascii file" in the input file, see Section **??**.

  'probability density function': Generate a random distribution of particles over the entire simulation domain. The probability density is prescribed in the form of a user-prescribed function. The format of this function follows the syntax understood by the muparser library, see Section 5.1.3. The return value of the function is always checked to be a non-negative probability density but it can be zero inparts of the domain.

  'random uniform': Generates a random uniform distribution of particles over the entire simulation domain.

  'uniform box': Generate a uniform distribution of particles over a rectangular domain in 2D or 3D. Uniform here means the particles will be generated with an equal spacing in each spatial dimension. Note that in order to produce a regular distribution the number of generated tracers might not exactly match the one specified in the input file.

  'uniform radial': Generate a uniform distribution of particlesover a spherical domain in 2D or 3D. Uniform here means the particles will be generated with an equal spacing in each spherical spatial dimension, i.e., the particles are created at positions that increase linearly with equal spacing in radius, colatitude and longitude around a certain center point. Note that in order to produce a regular distribution the number of generated tracers might not exactly match the one specified in the input file.

  Possible values: [Selection ascii file—probability density function—random uniform—uniform box—uniform radial]

- *Parameter name:* `Time between data output`

  Value: 1e8

  Default: 1e8

  Description: The time interval between each generation of output files. A value of zero indicates that output should be generated every time step.

  Units: years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Tracer weight`

  Value: 10

  Default: 10

  Description: Weight that is associated with the computational load of a single particle. The sum of tracer weights will be added to the sum of cell weights to determine the partitioning of the mesh. Every cell without tracers is associated with a weight of 1000.

  Possible values: [Integer range 0...2147483647 (inclusive)]

5.94 Parameters in section `Postprocess/Tracers/Function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.95 Parameters in section `Postprocess/Tracers/Generator`

5.96 Parameters in section `Postprocess/Tracers/Generator/Ascii file`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/particle/generator/ascii/

  Default: $ASPECT_SOURCE_DIR/data/particle/generator/ascii/

  Description: The name of a directory that contains the tracer data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: tracer.dat

  Default: tracer.dat

  Description: The name of the tracer file.

  Possible values: [Anything]

5.97 Parameters in section `Postprocess/Tracers/Generator/Probability density function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.98 Parameters in section `Postprocess/Tracers/Generator/Uniform box`

- *Parameter name:* `Maximum x`

  Value: 1

  Default: 1

  Description: Maximum x coordinate for the region of tracers.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum y`

  Value: 1

  Default: 1

  Description: Maximum y coordinate for the region of tracers.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum z`

  Value: 1

  Default: 1

  Description: Maximum z coordinate for the region of tracers.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum x`

  Value: 0

  Default: 0

  Description: Minimum x coordinate for the region of tracers.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum y`

  Value: 0

  Default: 0

  Description: Minimum y coordinate for the region of tracers.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum z`
  Value: 0
  Default: 0
  Description: Minimum z coordinate for the region of tracers.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.99 Parameters in section `Postprocess/Tracers/Generator/Uniform radial`

- *Parameter name:* `Center x`
  Value: 0
  Default: 0
  Description: x coordinate for the center of the spherical region, where tracers are generated.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Center y`
  Value: 0
  Default: 0
  Description: y coordinate for the center of the spherical region, where tracers are generated.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Center z`
  Value: 0
  Default: 0
  Description: z coordinate for the center of the spherical region, where tracers are generated.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Maximum latitude`
  Value: 3.1415
  Default: 3.1415
  Description: Maximum latitude coordinate for the region of tracers in degrees. Measured from the center position.
  Possible values: [Double 0...180 (inclusive)]

- *Parameter name:* `Maximum longitude`
  Value: 3.1415
  Default: 3.1415
  Description: Maximum longitude coordinate for the region of tracers in degrees. Measured from the center position.
  Possible values: [Double 0...360 (inclusive)]

- *Parameter name:* `Maximum radius`
  Value: 1
  Default: 1
  Description: Maximum radial coordinate for the region of tracers. Measured from the center position.
  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Minimum latitude`

  Value: 0

  Default: 0

  Description: Minimum latitude coordinate for the region of tracers in degrees. Measured from the center position.

  Possible values: [Double 0...180 (inclusive)]

- *Parameter name:* `Minimum longitude`

  Value: 0

  Default: 0

  Description: Minimum longitude coordinate for the region of tracers in degrees. Measured from the center position.

  Possible values: [Double 0...360 (inclusive)]

- *Parameter name:* `Minimum radius`

  Value: 0

  Default: 0

  Description: Minimum radial coordinate for the region of tracers. Measured from the center position.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Radial layers`

  Value: 1

  Default: 1

  Description: The number of radial shells of particles that will be generatedaround the central point.

  Possible values: [Integer range 1...2147483647 (inclusive)]

5.100 Parameters in section `Postprocess/Visualization`

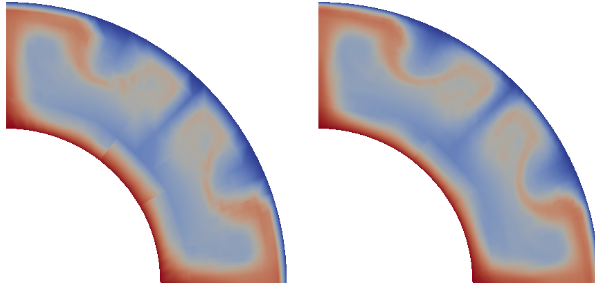
- *Parameter name:* `Interpolate output`

  Value: false

  Default: false

  Description: deal.II offers the possibility to linearly interpolate output fields of higher order elements to a finer resolution. This somewhat compensates the fact that most visualization software only offers linear interpolation between grid points and therefore the output file is a very coarse representation of the actual solution field. Activating this option increases the spatial resolution in each dimension by a factor equal to the polynomial degree used for the velocity finite element (usually 2). In other words, instead of showing one quadrilateral or hexahedron in the visualization per cell on which ASPECT computes, it shows multiple (for quadratic elements, it will describe each cell of the mesh on which we compute as 2×2 or $2 \times 2 \times 2$ cells in 2d and 3d, respectively; correspondingly more subdivisions are used if you use cubic, quartic, or even higher order elements for the velocity).

  The effect of using this option can be seen in the following picture showing a variation of the output produced with the input files from Section 6.3.1:



Here, the left picture shows one visualization cell per computational cell (i.e., the option is switch off, as is the default), and the right picture shows the same simulation with the option switched on. The images show the same data, demonstrating that interpolating the solution onto bilinear shape functions as is commonly done in visualizing data loses information.

Of course, activating this option also greatly increases the amount of data ASPECT will write to disk: approximately by a factor of 4 in 2d, and a factor of 8 in 3d, when using quadratic elements for the velocity, and correspondingly more for even higher order elements.

Possible values: [Bool]

- *Parameter name:* `List of output variables`

  Value:

  Default:

  Description: A comma separated list of visualization objects that should be run whenever writing graphical output. By default, the graphical output files will always contain the primary variables velocity, pressure, and temperature. However, one frequently wants to also visualize derived quantities, such as the thermodynamic phase that corresponds to a given temperature-pressure value, or the corresponding seismic wave speeds. The visualization objects do exactly this: they compute such derived quantities and place them into the output file. The current parameter is the place where you decide which of these additional output variables you want to have in your output file.

  The following postprocessors are available:

  'Vp anomaly': A visualization output object that generates output showing the anomaly in the seismic compression wave speed V_p as a spatially variable function with one value per cell. This anomaly is shown as a percentage change relative to the average value of V_p at the depth of this cell.

  'Vs anomaly': A visualization output object that generates output showing the anomaly in the seismic shear wave speed V_s as a spatially variable function with one value per cell. This anomaly is shown as a percentage change relative to the average value of V_s at the depth of this cell.

  'artificial viscosity': A visualization output object that generates output showing the value of the artificial viscosity on each cell.

  'boundary indicators': A visualization output object that generates output about the used boundary indicators. In a loop over the active cells, if a cell lies at a domain boundary, the boundary indicator of the face along the boundary is requested. In case the cell does not lie along any domain boundary, the cell is assigned the value of the largest used boundary indicator plus one. When a cell is situated in one of the corners of the domain, multiple faces will have a boundary indicator. This postprocessor returns the value of the first face along a boundary that is encountered in a loop over all the faces.

'compositional vector': A visualization output object that outputs vectors whose components are derived from compositional fields. Input parameters for this postprocessor are defined in section Postprocess/Visualization/Compositional fields as vectors

'density': A visualization output object that generates output for the density.

'depth': A visualization output postprocessor that outputs the depth for all points inside the domain, as determined by the geometry model.

'dynamic topography': A visualization output object that generates output for the dynamic topography. The approach to determine the dynamic topography requires us to compute the stress tensor and evaluate the component of it in the direction in which gravity acts. In other words, we compute $\sigma_{rr} = \hat{g}^T(2\eta\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{div }\mathbf{u})I)\hat{g} - p_d$ where $\hat{g} = \mathbf{g}/\|\mathbf{g}\|$ is the direction of the gravity vector $\mathbf{g}$ and $p_d = p - p_a$ is the dynamic pressure computed by subtracting the adiabatic pressure p_a from the total pressure p computed as part of the Stokes solve. From this, the dynamic topography is computed using the formula $h = \frac{\sigma_{rr}}{\|\mathbf{g}\|\rho}$ where ρ is the density at the cell center.

Strictly speaking, the dynamic topography is of course a quantity that is only of interest at the surface. However, we compute it everywhere to make things fit into the framework within which we produce data for visualization. You probably only want to visualize whatever data this postprocessor generates at the surface of your domain and simply ignore the rest of the data generated.

'error indicator': A visualization output object that generates output showing the estimated error or other mesh refinement indicator as a spatially variable function with one value per cell.

'friction heating': A visualization output object that generates output for the amount of friction heating often referred to as $\tau : \epsilon$. More concisely, in the incompressible case, the quantity that is output is defined as $\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u})$ where η is itself a function of temperature, pressure and strain rate. In the compressible case, the quantity that's computed is $\eta[\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{tr }\varepsilon(\mathbf{u}))\mathbf{I}] : [\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{tr }\varepsilon(\mathbf{u}))\mathbf{I}]$.

'gravity': A visualization output object that outputs the gravity vector.

'heating': A visualization output object that generates output for all the heating terms used in the energy equation.

'material properties': A visualization output object that generates output for the material properties given by the material model.There are a number of other visualization postprocessors that offer to write individual material properties. However, they all individually have to evaluate the material model. This is inefficient if one wants to output more than just one or two of the fields provided by the material model. The current postprocessor allows to output a (potentially large) subset of all of the information provided by material models at once, with just a single material model evaluation per output point.

'melt fraction': A visualization output object that generates output for the melt fraction at the temperature and pressure of the current point (batch melting). Does not take into account latent heat. If there are no compositional fields, this postprocessor will visualize the melt fraction of peridotite (calculated using the anhydrous model of Katz, 2003). If there is at least one compositional field, the postprocessor assumes that the first compositional field is the content of pyroxenite, and will visualize the melt fraction for a mixture of peridotite and pyroxenite (using the melting model of Sobolev, 2011 for pyroxenite). All the parameters that were used in these calculations can be changed in the input file, the most relevant maybe being the mass fraction of Cpx in peridotite in the Katz melting model (Mass fraction cpx), which right now has a default of 15%. The corresponding p-T-diagrams can be generated by running the tests melt_postprocessor_peridotite and melt_postprocessor_pyroxenite.

'nonadiabatic pressure': A visualization output object that generates output for the non-adiabatic component of the pressure.

'nonadiabatic temperature': A visualization output object that generates output for the non-adiabatic component of the pressure.

'particle count': A visualization output object that generates output about the number of particles per cell.

'partition': A visualization output object that generates output for the parallel partition that every cell of the mesh is associated with.

'seismic vp': A visualization output object that generates output for the seismic P-wave speed.

'seismic vs': A visualization output object that generates output for the seismic S-wave speed.

'shear stress': A visualization output object that generates output for the 3 (in 2d) or 6 (in 3d) components of the shear stress tensor, i.e., for the components of the tensor $2\eta\varepsilon(\mathbf{u})$ in the incompressible case and $2\eta\left[\varepsilon(\mathbf{u})-\frac{1}{3}(\textrm{tr}\,\varepsilon(\mathbf{u}))\mathbf{I}\right]$ in the compressible case. The shear stress differs from the full stress tensor by the absence of the pressure.

'specific heat': A visualization output object that generates output for the specific heat C_p.

'strain rate': A visualization output object that generates output for the norm of the strain rate, i.e., for the quantity $\sqrt{\varepsilon(\mathbf{u}):\varepsilon(\mathbf{u})}$ in the incompressible case and $\sqrt{[\varepsilon(\mathbf{u})-\frac{1}{3}(\textrm{tr}\,\varepsilon(\mathbf{u}))\mathbf{I}]:[\varepsilon(\mathbf{u})-\frac{1}{3}(\textrm{tr}\,\varepsilon(\mathbf{u}))\mathbf{I}]}$ in the compressible case.

'stress': A visualization output object that generates output for the 3 (in 2d) or 6 (in 3d) components of the stress tensor, i.e., for the components of the tensor $2\eta\varepsilon(\mathbf{u})+pI$ in the incompressible case and $2\eta\left[\varepsilon(\mathbf{u})-\frac{1}{3}(\textrm{tr}\,\varepsilon(\mathbf{u}))\mathbf{I}\right]+pI$ in the compressible case.

'thermal conductivity': A visualization output object that generates output for the thermal conductivity k.

'thermal diffusivity': A visualization output object that generates output for the thermal diffusivity $\kappa=\frac{k}{\rho c_p}$, with k the thermal conductivity.

'thermal expansivity': A visualization output object that generates output for the thermal expansivity.

'thermodynamic phase': A visualization output object that generates output for the integer number of the phase that is thermodynamically stable at the temperature and pressure of the current point.

'vertical heat flux': A visualization output object that generates output for the heat flux in the vertical direction, which is the sum of the advective and the conductive heat flux, with the sign convention of positive flux upwards.

'viscosity': A visualization output object that generates output for the viscosity.

'viscosity ratio': A visualization output object that generates output for the ratio between dislocation viscosity and diffusion viscosity.

Possible values: [MultipleSelection Vp anomaly—Vs anomaly—artificial viscosity—boundary indicators—compositional vector—density—depth—dynamic topography—error indicator—friction heating—gravity—heating—material properties—melt fraction—nonadiabatic pressure—nonadiabatic temperature—particle count—partition—seismic vp—seismic vs—shear stress—specific heat—strain rate—stress—thermal conductivity—thermal diffusivity—thermal expansivity—thermodynamic phase—vertical heat flux—viscosity—viscosity ratio]

- *Parameter name:* `Number of grouped files`

  Value: 0

  Default: 0

  Description: VTU file output supports grouping files from several CPUs into a given number of files using MPI I/O when writing on a parallel filesystem. Select 0 for no grouping. This will disable parallel file output and instead write one file per processor. A value of 1 will generate one big file containing the whole solution, while a larger value will create that many files (at most as many as there are mpi ranks).

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Output format`

  Value: vtu

  Default: vtu

  Description: The file format to be used for graphical output.

  Possible values: [Selection none—dx—ucd—gnuplot—povray—eps—gmv—tecplot—tecplot_binary—vtk—vtu—hdf5—
  intermediate]

- *Parameter name:* `Output mesh velocity`

  Value: false

  Default: false

  Description: For free surface computations Aspect uses an Arbitrary-Lagrangian-Eulerian formulation to handle deforming the domain, so the mesh has its own velocity field. This may be written as an output field by setting this parameter to true.

  Possible values: [Bool]

- *Parameter name:* `Temporary output location`

  Value:

  Default:

  Description: On large clusters it can be advantageous to first write the output to a temporary file on a local file system and later move this file to a network file system. If this variable is set to a non-empty string it will be interpreted as a temporary storage location.

  Possible values: [Anything]

- *Parameter name:* `Time between graphical output`

  Value: 1e8

  Default: 1e8

  Description: The time interval between each generation of graphical output files. A value of zero indicates that output should be generated in each time step. Units: years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Write in background thread`

  Value: false

  Default: false

  Description: File operations can potentially take a long time, blocking the progress of the rest of the model run. Setting this variable to 'true' moves this process into a background thread, while the rest of the model continues.

  Possible values: [Bool]

5.101 Parameters in section `Postprocess/Visualization/Compositional fields as vectors`

- *Parameter name:* `Names of fields`

  Value:

  Default:

Description: A list of sets of compositional fields which should be output as vectors. Sets are separated from each other by semicolons and vector components within each set are separated by commas (e.g. $vec1_x$, $vec1_y$; $vec2_x$, $vec2_y$) where each name must be a defined named compositional field. If only one name is given in a set, it is interpreted as the first in a sequence of dim consecutive compositional fields.

Possible values: [Anything]

- *Parameter name:* `Names of vectors`

  Value:

  Default:

  Description: Names of vectors as they will appear in the output.

  Possible values: [List list of [Anything] of length 0...4294967295 (inclusive)]

5.102 Parameters in section `Postprocess/Visualization/Dynamic Topography`

- *Parameter name:* `Subtract mean of dynamic topography`

  Value: false

  Default: false

  Description: Option to remove the mean dynamic topography in the outputted data file (not visualization). 'true' subtracts the mean, 'false' leaves the calculated dynamic topography as is.

  Possible values: [Bool]

5.103 Parameters in section `Postprocess/Visualization/Material properties`

- *Parameter name:* `List of material properties`

  Value: density,thermal expansivity,specific heat,viscosity

  Default: density,thermal expansivity,specific heat,viscosity

  Description: A comma separated list of material properties that should be written whenever writing graphical output. By default, the material properties will always contain the density, thermal expansivity, specific heat and viscosity. The following material properties are available:

  viscosity—density—thermal expansivity—specific heat—thermal conductivity—thermal diffusivity—compressibility—e
  derivative temperature—entropy derivative pressure—reaction terms

  Possible values: [MultipleSelection viscosity—density—thermal expansivity—specific heat—thermal conductivity—thermal diffusivity—compressibility—entropy derivative temperature—entropy derivative pressure—reaction terms]

5.104 Parameters in section `Postprocess/Visualization/Melt fraction`

- *Parameter name:* `A1`

  Value: 1085.7

  Default: 1085.7

  Description: Constant parameter in the quadratic function that approximates the solidus of peridotite. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `A2`

  Value: 1.329e-7

  Default: 1.329e-7

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the solidus of peridotite. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `A3`

  Value: -5.1e-18

  Default: -5.1e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the solidus of peridotite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `B1`

  Value: 1475.0

  Default: 1475.0

  Description: Constant parameter in the quadratic function that approximates the lherzolite liquidus used for calculating the fraction of peridotite-derived melt. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `B2`

  Value: 8.0e-8

  Default: 8.0e-8

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the lherzolite liquidus used for calculating the fraction of peridotite-derived melt. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `B3`

  Value: -3.2e-18

  Default: -3.2e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the lherzolite liquidus used for calculating the fraction of peridotite-derived melt. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `C1`

  Value: 1780.0

  Default: 1780.0

  Description: Constant parameter in the quadratic function that approximates the liquidus of peridotite. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `C2`

  Value: 4.50e-8

  Default: 4.50e-8

Description: Prefactor of the linear pressure term in the quadratic function that approximates the liquidus of peridotite. Units: C/Pa.

Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `C3`

  Value: -2.0e-18

  Default: -2.0e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the liquidus of peridotite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `D1`

  Value: 976.0

  Default: 976.0

  Description: Constant parameter in the quadratic function that approximates the solidus of pyroxenite. Units: C.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `D2`

  Value: 1.329e-7

  Default: 1.329e-7

  Description: Prefactor of the linear pressure term in the quadratic function that approximates the solidus of pyroxenite. Note that this factor is different from the value given in Sobolev, 2011, because they use the potential temperature whereas we use the absolute temperature. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `D3`

  Value: -5.1e-18

  Default: -5.1e-18

  Description: Prefactor of the quadratic pressure term in the quadratic function that approximates the solidus of pyroxenite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `E1`

  Value: 663.8

  Default: 663.8

  Description: Prefactor of the linear depletion term in the quadratic function that approximates the melt fraction of pyroxenite. Units: C/Pa.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `E2`

  Value: -611.4

  Default: -611.4

  Description: Prefactor of the quadratic depletion term in the quadratic function that approximates the melt fraction of pyroxenite. Units: $C/(Pa^2)$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `Mass fraction cpx`

  Value: 0.15

  Default: 0.15

  Description: Mass fraction of clinopyroxene in the peridotite to be molten. Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `beta`

  Value: 1.5

  Default: 1.5

  Description: Exponent of the melting temperature in the melt fraction calculation. Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `r1`

  Value: 0.5

  Default: 0.5

  Description: Constant in the linear function that approximates the clinopyroxene reaction coefficient. Units: non-dimensional.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

- *Parameter name:* `r2`

  Value: 8e-11

  Default: 8e-11

  Description: Prefactor of the linear pressure term in the linear function that approximates the clinopyroxene reaction coefficient. Units: $1/Pa$.

  Possible values: [Double -1.79769e+308...1.79769e+308 (inclusive)]

5.105 Parameters in section `Prescribed Stokes solution`

- *Parameter name:* `Model name`

  Value: unspecified

  Default: unspecified

  Description: Select one of the following models:

  'ascii data': Implementation of a model in which the velocityis derived from files containing data in ascii format. Note the required format of the input data: The first lines may contain any number of comments if they begin with '#', but one of these lines needs to contain the number of grid points in each dimension as for example '# POINTS: 3 3'. The order of the data columns has to be 'x', 'y', 'v_x' , 'v_y' in a 2d model and 'x', 'y', 'z', 'v_x' , 'v_y' , 'v_z' in a 3d model. Note that the data in the input files need to be sorted in a specific order: the first coordinate needs to ascend first, followed by the second and the third at last in order to assign the correct data to the prescribed coordinates. If you use a spherical model, then the data will still be handled as Cartesian, however the assumed grid changes. 'x' will be replaced by the radial distance of the point to the bottom of the model, 'y' by the azimuth angle and 'z' by the polar angle measured positive from the north pole. The grid will be assumed to be a latitude-longitude grid. Note that the order of spherical coordinates is 'r', 'phi', 'theta' and not 'r', 'theta', 'phi', since this allows for dimension independent expressions.

'circle': This value describes a vector field that rotates around the z-axis with constant angular velocity (i.e., with a velocity that increases with distance from the axis). The pressure is set to zero.

'function': This plugin allows to prescribe the Stokes solution for the velocity and pressure field in terms of an explicit formula. The format of these functions follows the syntax understood by the muparser library, see Section 5.1.3.

Possible values: [Selection ascii data—circle—function—unspecified]

5.106 Parameters in section `Prescribed Stokes solution/Ascii data model`

- *Parameter name:* `Data directory`

  Value: $ASPECT_SOURCE_DIR/data/prescribed-stokes-solution/

  Default: $ASPECT_SOURCE_DIR/data/prescribed-stokes-solution/

  Description: The name of a directory that contains the model data. This path may either be absolute (if starting with a '/') or relative to the current directory. The path may also include the special text '$ASPECT_SOURCE_DIR' which will be interpreted as the path in which the ASPECT source files were located when ASPECT was compiled. This interpretation allows, for example, to reference files located in the 'data/' subdirectory of ASPECT.

  Possible values: [DirectoryName]

- *Parameter name:* `Data file name`

  Value: box_2d.txt

  Default: box_2d.txt

  Description: The file name of the material data. Provide file in format: (Velocity file name).%s%d where %s is a string specifying the boundary of the model according to the names of the boundary indicators (of a box or a spherical shell).%d is any sprintf integer qualifier, specifying the format of the current file number.

  Possible values: [Anything]

- *Parameter name:* `Scale factor`

  Value: 1

  Default: 1

  Description: Scalar factor, which is applied to the boundary velocity. You might want to use this to scale the velocities to a reference model (e.g. with free-slip boundary) or another plate reconstruction. Another way to use this factor is to convert units of the input files. The unit is assumed to bem/s or m/yr depending on the 'Use years in output instead of seconds' flag. If you provide velocities in cm/yr set this factor to 0.01.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.107 Parameters in section `Prescribed Stokes solution/Pressure function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0

  Default: 0

  Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as 'sin' or 'cos'. In addition, it may contain expressions like 'if(x¿0, 1, -1)' where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

  If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

  Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is 'x' (in 1d), 'x,y' (in 2d) or 'x,y,z' (in 3d) for spatial coordinates and 't' for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to 'r,phi,theta,t' and then use these variable names in your function expression.

  Possible values: [Anything]

5.108 Parameters in section `Prescribed Stokes solution/Velocity function`

- *Parameter name:* `Function constants`

  Value:

  Default:

  Description: Sometimes it is convenient to use symbolic constants in the expression that describes the function, rather than having to use its numeric value everywhere the constant appears. These values can be defined using this parameter, in the form 'var1=value1, var2=value2, ...'.

  A typical example would be to set this runtime parameter to 'pi=3.1415926536' and then use 'pi' in the expression of the actual formula. (That said, for convenience this class actually defines both 'pi' and 'Pi' by default, but you get the idea.)

  Possible values: [Anything]

- *Parameter name:* `Function expression`

  Value: 0; 0

  Default: 0; 0

Description: The formula that denotes the function you want to evaluate for particular values of the independent variables. This expression may contain any of the usual operations such as addition or multiplication, as well as all of the common functions such as ‘sin’ or ‘cos’. In addition, it may contain expressions like ‘if(x¿0, 1, -1)’ where the expression evaluates to the second argument if the first argument is true, and to the third argument otherwise. For a full overview of possible expressions accepted see the documentation of the muparser library at http://muparser.beltoforion.de/.

If the function you are describing represents a vector-valued function with multiple components, then separate the expressions for individual components by a semicolon.

Possible values: [Anything]

- *Parameter name:* `Variable names`

  Value: x,y,t

  Default: x,y,t

  Description: The name of the variables as they will be used in the function, separated by commas. By default, the names of variables at which the function will be evaluated is ‘x’ (in 1d), ‘x,y’ (in 2d) or ‘x,y,z’ (in 3d) for spatial coordinates and ‘t’ for time. You can then use these variable names in your function expression and they will be replaced by the values of these variables at which the function is currently evaluated. However, you can also choose a different set of names for the independent variables at which to evaluate your function expression. For example, if you work in spherical coordinates, you may wish to set this input parameter to ‘r,phi,theta,t’ and then use these variable names in your function expression.

  Possible values: [Anything]

5.109 Parameters in section `Termination criteria`

- *Parameter name:* `Checkpoint on termination`

  Value: false

  Default: false

  Description: Whether to checkpoint the simulation right before termination.

  Possible values: [Bool]

- *Parameter name:* `End step`

  Value: 100

  Default: 100

  Description: Terminate the simulation once the specified timestep has been reached.

  Possible values: [Integer range 0...2147483647 (inclusive)]

- *Parameter name:* `Termination criteria`

  Value: end time

  Default: end time

  Description: A comma separated list of termination criteria that will determine when the simulation should end. Whether explicitly stated or not, the “end time” termination criterion will always be used.The following termination criteria are available:

  ‘end step’: Terminate the simulation once the specified timestep has been reached.

  ‘end time’: Terminate the simulation once the end time specified in the input file has been reached. Unlike all other termination criteria, this criterion is *always* active, whether it has been explicitly

selected or not in the input file (this is done to preserve historical behavior of ASPECT, but it also likely does not inconvenience anyone since it is what would be selected in most cases anyway).

'steady state velocity': A criterion that terminates the simulation when the RMS of the velocity field stays within a certain range for a specified period of time.

'user request': Terminate the simulation gracefully when a file with a specified name appears in the output directory. This allows the user to gracefully exit the simulation at any time by simply creating such a file using, for example, `touch output/terminate`. The file's location is chosen to be in the output directory, rather than in a generic location such as the Aspect directory, so that one can run multiple simulations at the same time (which presumably write to different output directories) and can selectively terminate a particular one.

Possible values: [MultipleSelection end step—end time—steady state velocity—user request]

5.110 Parameters in section `Termination criteria/Steady state velocity`

- *Parameter name:* `Maximum relative deviation`

  Value: 0.05

  Default: 0.05

  Description: The maximum relative deviation of the RMS in recent simulation time for the system to be considered in steady state. If the actual deviation is smaller than this number, then the simulation will be terminated.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Time in steady state`

  Value: 1e7

  Default: 1e7

  Description: The minimum length of simulation time that the system should be in steady state before termination.Units: years if the 'Use years in output instead of seconds' parameter is set; seconds otherwise.

  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.111 Parameters in section `Termination criteria/User request`

- *Parameter name:* `File name`

  Value: terminate-aspect

  Default: terminate-aspect

  Description: The name of a file that, if it exists in the output directory (whose name is also specified in the input file) will lead to termination of the simulation. The file's location is chosen to be in the output directory, rather than in a generic location such as the Aspect directory, so that one can run multiple simulations at the same time (which presumably write to different output directories) and can selectively terminate a particular one.

  Possible values: [FileName (Type: input)]

6 Cookbooks

In this section, let us present a number of "cookbooks" – examples of how to use ASPECT in typical or less typical ways. As discussed in Sections 4 and 5, ASPECT is driven by run-time parameter files, and so setting up a particular situation primarily comes down to creating a parameter file that has the right entries.

Provide ap proximate run-times fo each of these cookbooks.

Thus, the subsections below will discuss in detail what parameters to set and to what values. Note that parameter files need not specify *all* parameters – of which there is a bewildering number – but only those that are relevant to the particular situation we would like to model. All parameters not listed explicitly in the input file are simply left at their default value (the default values are also documented in Section 5).

Of course, there are situations where what you want to do is not covered by the models already implemented. Specifically, you may want to try a different geometry, a different material or gravity model, or different boundary conditions. In such cases, you will need to implement these extensions in the actual source code. Section 7 provides information on how to do that.

The remainder of this section shows a number of applications of ASPECT. They are grouped into three categories: Simple setups of examples that show thermal convection (Section 6.2), setups that try to model geophysical situations (Section 6.3) and setups that are used to benchmark ASPECT to ensure correctness or to test accuracy of our solvers (Section 6.4). Before we get there, however, we will review how one usually approaches setting up computations in Section 6.1.

> **Note:** The input files discussed in the following sections can generally be found in the `cookbooks/` directory of your ASPECT installation.

6.1 How to set up computations

ASPECT's computations are controlled by input parameter files such as those we will discuss in the following sections.[15] Basically, these are just regular text files you can edit with programs like `gedit`, `kwrite` or `kate` when working on Linux, or something as simple as `NotePad` on Windows. When setting up these input files, you basically have to describe everything that characterizes the computation you want to do. In particular, this includes the following:

- What internal forces act on the medium (the equation)?
- What external forces do we have (the right hand side)
- What is the domain (geometry)?
- What happens at the boundary for each variable involved (boundary conditions)?
- How did it look at the beginning (initial conditions)?

For each of these questions, there are one or more input parameters (sometimes grouped into sections) that allow you to specify what you want. For example, to choose a geometry, you will typically have a block like this in your input file:

```
set Dimension = 2
subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 1
    set Y extent = 1
  end
end
```

This indicates that you want to do a computation in 2d, using a rectangular geometry (a "box") with edge length equal to one in both the x- and y-directions. Of course, there are other geometries you can choose from for the `Model name` parameter, and consequently other subsections that specify the details of these geometries.

Similarly, you describe boundary conditions using parameters such as this:

[15] You can also extend ASPECT using plugins – i.e., pieces of code you compile separately and either link into the ASPECT executable itself, or reference from the input file. This is discussed in Section 7.

```
subsection Model settings
  set Fixed temperature boundary indicators   = bottom, top
  set Zero velocity boundary indicators       =
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = left, right, bottom, top
end
```

This snippet describes which of the four boundaries of the two-dimensional box we have selected above should have a prescribed temperature or an insulating boundary, and at which parts of the boundary we want zero, tangential or prescribed velocities.[16]

If you go down the list of questions about the setup above, you have already done the majority of the work describing your computation. The remaining parameters you will typically want to specify have to do with the computation itself. For example, what variables do you want to output and how often? What statistics do you want to compute. The following sections will give ample examples for all of this, but using the questions above as a guideline is already a good first step.

Note: It is of course possible to set up input files for computations completely from scratch. However, in practice, it is often simpler to go through the list of cookbooks already provided and find one that comes close to what you want to do. You would then modify this cookbook until it does what you want to do. The advantage is that you can start with something you already know works, and you can inspect how each change you make – changing the details of the geometry, changing the material model, or changing what is being computed at the end of each time step – affects what you get.

6.2 Simple setups

6.2.1 Convection in a 2d box

In this first example, let us consider a simple situation: a 2d box of dimensions $[0, 1] \times [0, 1]$ that is heated from below, insulated at the left and right, and cooled from the top. We will also consider the simplest model, the incompressible Boussinesq approximation with constant coefficients $\eta, \rho_0, \mathbf{g}, C_p k$, for this testcase. Furthermore, we assume that the medium expands linearly with temperature. This leads to the following set of equations:

$$-\nabla \cdot [2\eta\varepsilon(\mathbf{u})] + \nabla p = \rho_0(1 - \alpha(T - T_0))\mathbf{g} \qquad \text{in } \Omega, \qquad (35)$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega, \qquad (36)$$

$$\rho_0(1 - \alpha(T - T_0))C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T\right) - \nabla \cdot k\nabla T = 0 \qquad \text{in } \Omega. \qquad (37)$$

It is well known that we can non-dimensionalize this set of equations by introducing the Raleigh number $Ra = \frac{g\alpha}{\eta k}$. Formally, we can obtain the non-dimensionalized equations by using the above form and setting coefficients in the following way:

$$\rho_0 = C_p = k = \alpha = \eta = 1, \qquad T_0 = 0, \qquad g = Ra,$$

[16]Internally, the geometry models ASPECT uses label every part of the boundary with what is called a *boundary indicator* – a number that identifies pieces of the boundary. If you know which number each piece has, you can list these numbers on the right hand sides of the assignments of boundary types above. For example, the left boundary of the box has boundary indicator zero (see Section 5.32), and using this number instead of the `left` would have been equally valid. However, numbers are far more difficult to remember than names, and consequently every geometry model provides string aliases such as "`left`" for each boundary indicator describing parts of the boundary. These symbolic aliases are specific to the geometry – for the box, they are "`left`", "`right`", "`bottom`", etc., whereas for a spherical shell they are "`inner`" and "`outer`" – but are described in the documentation of every geometry model, see Section 5.32.

where $\mathbf{g} = -g\mathbf{e}_z$ is the gravity vector in negative z-direction. While this would be a valid description of the problem, it is not what one typically finds in the literature because there the density in the temperature equation is chosen as ρ_0 rather than $\rho(1 - \alpha(T - T_0))$ as used by ASPECT. However, we can mimic this by choosing a very small value for α – small enough to ensure that for all reasonable temperatures, the density used here is equal to ρ_0 for all practical purposes –, and instead making g correspondingly larger. Consequently, in this cookbook we will use the following set of parameters:

$$\rho_0 = C_p = T_0 = k = \eta = 1, \qquad T_0 = 0, \qquad \alpha = 10^{-10}, \qquad g = 10^{10} Ra.$$

We will see all of these values again in the input file discussed below. The problem is completed by stating the velocity boundary conditions: tangential flow along all four of the boundaries of the box.

This situation describes a well-known benchmark problems for which a lot is known and against which we can compare our results. For example, the following is well understood:

- For values of the Rayleigh number less than a critical number $Ra_c \approx 780$, thermal diffusion dominates convective heat transport and any movement in the fluid is damped exponentially. If the Rayleigh number is moderately larger than this threshold then a stable convection pattern forms that transports heat from the bottom to the top boundaries. The simulations we will set up operates in this regime. Specifically, we will choose $Ra = 10^4$.

  On the other hand, if the Rayleigh number becomes even larger, a serious of period doublings starts that makes the system become more and more unstable. We will investigate some of this behavior at the end of this section.

- For certain values of the Rayleigh number, very accurate values for the heat flux through the bottom and top boundaries are available in the literate. For example, Blankenbach *et al.* report a non-dimensional heat flux of 4.884409 ± 0.00001, see [BBC+89]. We will compare our results against this value below.

With this said, let us consider how to represent this situation in practice.

The input file. The verbal description of this problem can be translated into an ASPECT input file in the following way (see Section 5 for a description of all of the parameters that appear in the following input file, and the indices at the end of this manual if you want to find a particular parameter; you can find the input file to run this cookbook example in `cookbooks/convection-box.prm`):

```
# At the top, we define the number of space dimensions we would like to
# work in:
set Dimension                              = 2

# There are several global variables that have to do with what
# time system we want to work in and what the end time is. We
# also designate an output directory.
set Use years in output instead of seconds = false
set End time                               = 0.5
set Output directory                       = output

# Then there are variables that describe the tolerance of
# the linear solver as well as how the pressure should
# be normalized. Here, we choose a zero average pressure
# at the surface of the domain (for the current geometry, the
# surface is defined as the top boundary).
set Linear solver tolerance                = 1e-7
set Temperature solver tolerance           = 1e-10

set Pressure normalization                 = surface
```

```
set Surface pressure                        = 0


# Then come a number of sections that deal with the setup
# of the problem to solve. The first one deals with the
# geometry of the domain within which we want to solve.
# The sections that follow all have the same basic setup
# where we select the name of a particular model (here,
# the box geometry) and then, in a further subsection,
# set the parameters that are specific to this particular
# model.
subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 1
    set Y extent = 1
  end
end


# The next section deals with the initial conditions for the
# temperature (there are no initial conditions for the
# velocity variable since the velocity is assumed to always
# be in a static equilibrium with the temperature field).
# There are a number of models with the 'function' model
# a generic one that allows us to enter the actual initial
# conditions in the form of a formula that can contain
# constants. We choose a linear temperature profile that
# matches the boundary conditions defined below plus
# a small perturbation:
subsection Initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,z
    set Function constants  = p=0.01, L=1, pi=3.1415926536, k=1
    set Function expression = (1.0-z) - p*cos(k*pi*x/L)*sin(pi*z)
  end
end


# Then follows a section that describes the boundary conditions
# for the temperature. The model we choose is called 'box' and
# allows to set a constant temperature on each of the four sides
# of the box geometry. In our case, we choose something that is
# heated from below and cooled from above. (As will be seen
# in the next section, the actual temperature prescribed here
# at the left and right does not matter.)
subsection Boundary temperature model
  set Model name = box

  subsection Box
    set Bottom temperature = 1
    set Left temperature   = 0
    set Right temperature  = 0
```

```
    set Top temperature    = 0
  end
end


# We then also have to prescribe several other parts of the model
# such as which boundaries actually carry a prescribed boundary
# temperature (as described in the documentation of the `box'
# geometry, boundaries 2 and 3 are the bottom and top boundaries)
# whereas all other parts of the boundary are insulated (i.e.,
# no heat flux through these boundaries; this is also often used
# to specify symmetry boundaries).
subsection Model settings
  set Fixed temperature boundary indicators   = bottom, top

  # The next parameters then describe on which parts of the
  # boundary we prescribe a zero or nonzero velocity and
  # on which parts the flow is allowed to be tangential.
  # Here, all four sides of the box allow tangential
  # unrestricted flow but with a zero normal component:
  set Zero velocity boundary indicators       =
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = left, right, bottom, top

  # The final part of this section describes whether we
  # want to include adiabatic heating (from a small
  # compressibility of the medium) or from shear friction,
  # as well as the rate of internal heating. We do not
  # want to use any of these options here:
  set Include adiabatic heating               = false
  set Include shear heating                   = false
end


# The following two sections describe first the
# direction (vertical) and magnitude of gravity and the
# material model (i.e., density, viscosity, etc). We have
# discussed the settings used here in the introduction to
# this cookbook in the manual already.
subsection Gravity model
  set Model name = vertical

  subsection Vertical
    set Magnitude = 1e14   # = Ra / Thermal expansion coefficient
  end
end


subsection Material model
  set Model name = simple # default:

  subsection Simple model
    set Reference density             = 1
    set Reference specific heat       = 1
    set Reference temperature         = 0
    set Thermal conductivity          = 1
```

```
    set Thermal expansion coefficient = 1e-10
    set Viscosity                     = 1
  end
end


# The settings above all pertain to the description of the
# continuous partial differential equations we want to solve.
# The following section deals with the discretization of
# this problem, namely the kind of mesh we want to compute
# on. We here use a globally refined mesh without
# adaptive mesh refinement.
subsection Mesh refinement
  set Initial global refinement                = 4
  set Initial adaptive refinement              = 0
  set Time steps between mesh refinement       = 0
end


# The final part is to specify what ASPECT should do with the
# solution once computed at the end of every time step. The
# process of evaluating the solution is called 'postprocessing'
# and we choose to compute velocity and temperature statistics,
# statistics about the heat flux through the boundaries of the
# domain, and to generate graphical output files for later
# visualization. These output files are created every time
# a time step crosses time points separated by 0.01. Given
# our start time (zero) and final time (0.5) this means that
# we will obtain 50 output files.
subsection Postprocess
  set List of postprocessors = velocity statistics, temperature statistics, ...
                                          ... heat flux statistics, visualization

  subsection Visualization
    set Time between graphical output = 0.01
  end
end
```

Running the program. When you run this program for the first time, you are probably still running ASPECT in debug mode (see Section 4.3) and you will get output like the following:

```
Number of active cells: 256 (on 5 levels)
Number of degrees of freedom: 3,556 (2,178+289+1,089)

*** Timestep 0:  t=0 seconds
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 30+5 iterations.

[... ...]

*** Timestep 1077:  t=0.499901 seconds
   Solving temperature system... 9 iterations.
   Solving Stokes system... 5 iterations.
```

```
  Postprocessing:
    RMS, max velocity:                  43.1 m/s, 69.8 m/s
    Temperature min/avg/max:            0 K, 0.5 K, 1 K
    Heat fluxes through boundary parts: 0.02056 W, -0.02061 W, -4.931 W, 4.931 W



+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    |       454s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Assemble Stokes system          |      1078 |      19.2s |       4.2% |
| Assemble temperature system     |      1078 |       329s |        72% |
| Build Stokes preconditioner     |         1 |    0.0995s |     0.022% |
| Build temperature preconditioner|      1078 |      5.84s |       1.3% |
| Solve Stokes system             |      1078 |      15.6s |       3.4% |
| Solve temperature system        |      1078 |      3.72s |      0.82% |
| Initialization                  |         2 |    0.0474s |      0.01% |
| Postprocessing                  |      1078 |      61.9s |        14% |
| Setup dof systems               |         1 |     0.221s |     0.049% |
+---------------------------------+-----------+------------+------------+
```

If you've read up on the difference between debug and optimized mode (and you should before you switch!) then consider disabling debug mode. If you run the program again, every number should look exactly the same (and it does, in fact, as I am writing this) except for the timing information printed every hundred time steps and at the end of the program:

```
+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    |      48.3s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Assemble Stokes system          |      1078 |      1.68s |       3.5% |
| Assemble temperature system     |      1078 |      26.3s |        54% |
| Build Stokes preconditioner     |         1 |    0.0401s |     0.083% |
| Build temperature preconditioner|      1078 |      4.87s |        10% |
| Solve Stokes system             |      1078 |      6.76s |        14% |
| Solve temperature system        |      1078 |      1.76s |       3.7% |
| Initialization                  |         2 |    0.0241s |      0.05% |
| Postprocessing                  |      1078 |      4.99s |        10% |
| Setup dof systems               |         1 |    0.0394s |     0.082% |
+---------------------------------+-----------+------------+------------+
```

In other words, the program ran about 10 times faster than before. Not all operations became faster to the same degree: assembly, for example, is an area that traverses a lot of code both in ASPECT and in DEAL.II and so encounters a lot of verification code in debug mode. On the other hand, solving linear systems primarily requires lots of matrix vector operations. Overall, the fact that in this example, assembling linear systems and preconditioners takes so much time compared to actually solving them is primarily a reflection of how simple the problem is that we solve in this example. This can also be seen in the fact that the number of iterations necessary to solve the Stokes and temperature equations is so low. For more complex problems with non-constant coefficients such as the viscosity, as well as in 3d, we have to spend much more work solving linear systems whereas the effort to assemble linear systems remains the same.

Visualizing results. Having run the program, we now want to visualize the numerical results we got. ASPECT can generate graphical output in formats understood by pretty much any visualization program

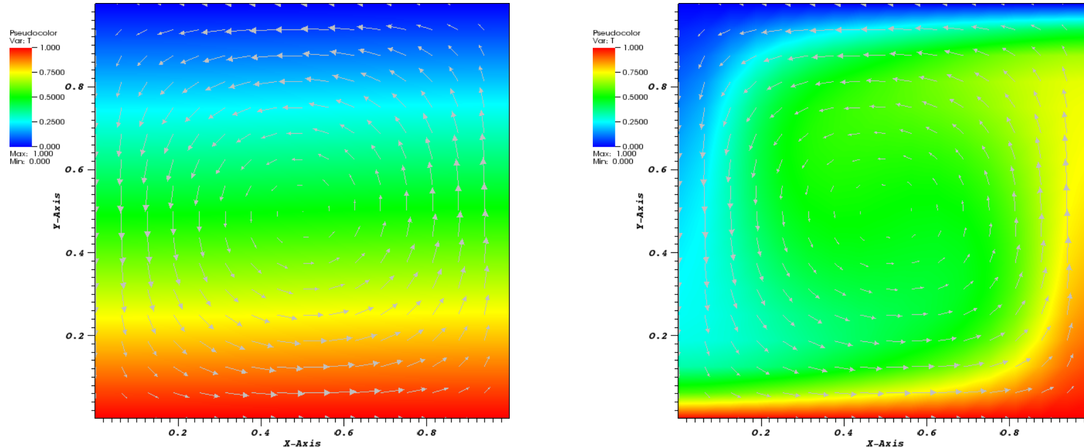


Figure 6: *Convection in a box: Initial temperature and velocity field (left) and final state (right).*

(see the parameters described in Section 5.100) but we will here follow the discussion in Section 4.4 and use the default VTU output format to visualize using the Visit program.

In the parameter file we have specified that graphical output should be generated every 0.01 time units. Looking through these output files, we find that the flow and temperature fields quickly converge to a stationary state. Fig. 6 shows the initial and final states of this simulation.

There are many other things we can learn from the output files generated by ASPECT, specifically from the statistics file that contains information collected at every time step and that has been discussed in Section 4.4.2. In particular, in our input file, we have selected that we would like to compute velocity, temperature, and heat flux statistics. These statistics, among others, are listed in the statistics file whose head looks like this for the current input file:

```
# 1: Time step number
# 2: Time (seconds)
# 3: Number of mesh cells
# 4: Number of Stokes degrees of freedom
# 5: Number of temperature degrees of freedom
# 6: Iterations for temperature solver
# 7: Iterations for Stokes solver
# 8: Time step size (seconds)
# 9: RMS velocity (m/s)
# 10: Max. velocity (m/s)
# 11: Minimal temperature (K)
# 12: Average temperature (K)
# 13: Maximal temperature (K)
# 14: Average nondimensional temperature (K)
# 15: Outward heat flux through boundary with indicator 0 (W)
# 16: Outward heat flux through boundary with indicator 1 (W)
# 17: Outward heat flux through boundary with indicator 2 (W)
# 18: Outward heat flux through boundary with indicator 3 (W)
# 19: Visualization file name
... lots of numbers arranged in columns ...
```

Fig. 7 shows the results of visualizing the data that can be found in columns 2 (the time) plotted against columns 9 and 10 (root mean square and maximal velocities). Plots of this kind can be generated with `Gnuplot` by typing (see Section 4.4.2 for a more thorough discussion):

```
plot "output/statistics" using 2:9 with lines
```

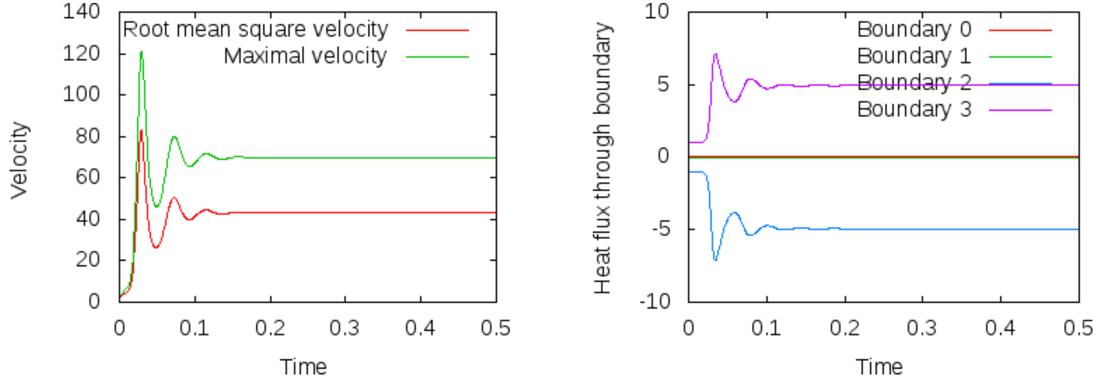


Figure 7: *Convection in a box: Root mean square and maximal velocity as a function of simulation time (left). Heat flux through the four boundaries of the box (right).*

Fig. 7 shows clearly that the simulation enters a steady state after about $t \approx 0.1$ and then changes very little. This can also be observed using the graphical output files from which we have generated Fig. 6. One can look further into this data to find that the flux through the top and bottom boundaries is not exactly the same (up to the obvious difference in sign, given that at the bottom boundary heat flows into the domain and at the top boundary out of it) at the beginning of the simulation until the fluid has attained its equilibrium. However, after $t \approx 0.2$, the fluxes differ by only $5 \cdot 10^{-5}$, i.e., by less than 0.001% of their magnitude.[17] The flux we get at the last time step, 4.931, is less than 1% away from the value reported in [BBC+89] although we compute on a 16×16 mesh and the values reported by Blankenbach are extrapolated from meshes of size up to 72×72. This shows the accuracy that can be obtained using a higher order finite element. Secondly, the fluxes through the left and right boundary are not exactly zero but small. Of course, we have prescribed boundary conditions of the form $\frac{\partial T}{\partial \mathbf{n}} = 0$ along these boundaries, but this is subject to discretization errors. It is easy to verify that the heat flux through these two boundaries disappears as we refine the mesh further.

Furthermore, ASPECT automatically also collects statistics about many of its internal workings. Fig. 8 shows the number of iterations required to solve the Stokes and temperature linear systems in each time step. It is easy to see that these are more difficult to solve in the beginning when the solution still changes significant from time step to time step. However, after some time, the solution remains mostly the same and solvers then only need 9 or 10 iterations for the temperature equation and 4 or 5 iterations for the Stokes equations because the starting guess for the linear solver – the previous time step's solution – is already pretty good. If you look at any of the more complex cookbooks, you will find that one needs many more iterations to solve these equations.

Play time 1: Different Rayleigh numbers. After showing you results for the input file as it can be found in `cookbooks/convection-box.prm`, let us end this section with a few ideas on how to play with it and what to explore. The first direction one could take this example is certainly to consider different Rayleigh numbers. As mentioned above, for the value $Ra = 10^4$ for which the results above have been produced, one gets a stable convection pattern. On the other hand, for values $Ra < Ra_c \approx 780$, any movement of the fluid dies down exponentially and we end up with a situation where the fluid doesn't move and heat is transported from the bottom to the top only through heat conduction. This can be explained by considering that the Rayleigh number in a box of unit extent is defined as $Ra = \frac{g\alpha}{\eta k}$. A small Rayleigh number means that the viscosity is too large (i.e., the buoyancy given by the product of the magnitude of gravity times the thermal expansion coefficient is not strong enough to overcome friction forces within the fluid).

On the other hand, if the Rayleigh number is large (i.e., the viscosity is small or the buoyancy large) then the fluid develops an unsteady convection period. As we consider fluids with larger and larger Ra, this

[17] This difference is far smaller than the numerical error in the heat flux on the mesh this data is computed on.

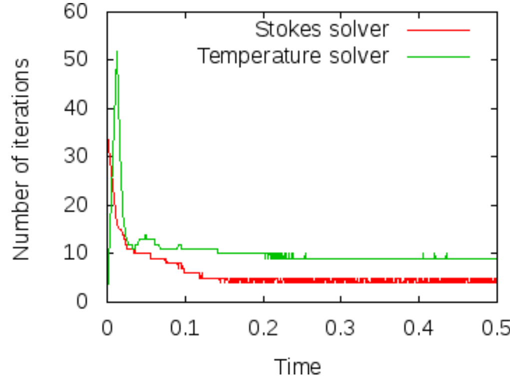


Figure 8: *Convection in a box: Number of linear iterations required to solve the Stokes and temperature equations in each time step.*

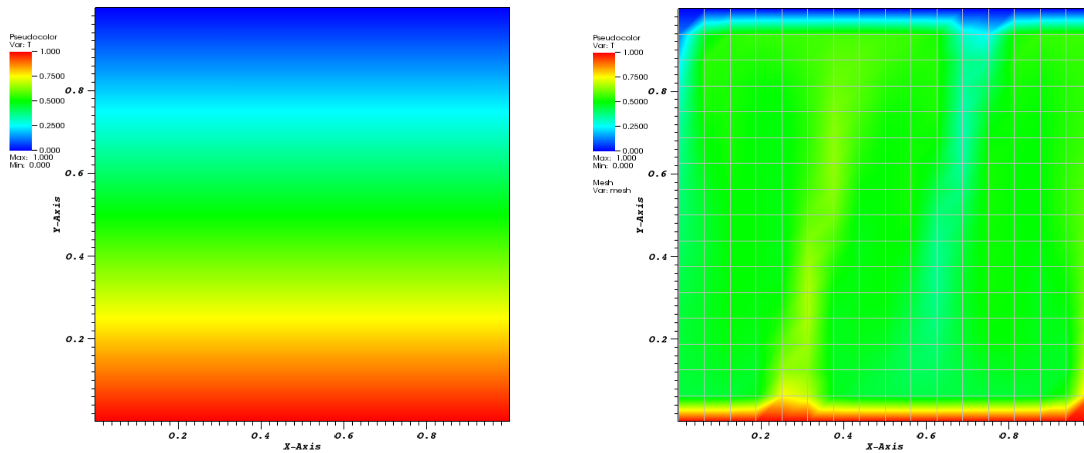


Figure 9: *Convection in a box: Temperature fields at the end of a simulation for $Ra = 10^2$ where thermal diffusion dominates (left) and $Ra = 10^6$ where convective heat transport dominates (right). The mesh on the right is clearly too coarse to resolve the structure of the solution.*

pattern goes through a sequence of period-doubling events until flow finally becomes chaotic. The structures of the flow pattern also become smaller and smaller.

We illustrate these situations in Fig.s 9 and 10. The first shows the temperature field at the end of a simulation for $Ra = 10^2$ (below Ra_c) and at $Ra = 10^6$. Obviously, for the right picture, the mesh is not fine enough to accurately resolve the features of the flow field and we would have to refine it more. The second of the figures shows the velocity and heatflux statistics for the computation with $Ra = 10^6$; it is obvious here that the flow no longer settles into a steady state but has a periodic behavior. This can also be seen by looking at movies of the solution.

To generate these results, remember that we have chosen $\alpha = 10^{-10}$ and $g = 10^{10} Ra$ in our input file. In other words, changing the input file to contain the parameter setting

```
subsection Gravity model
  subsection Vertical
    set Magnitude = 1e16    # = Ra / Thermal expansion coefficient
  end
end
```

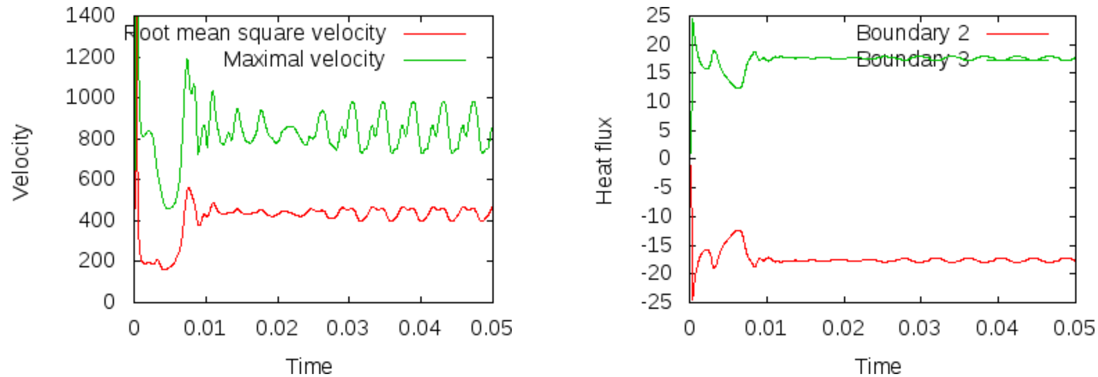


Figure 10: *Convection in a box: Velocities (left) and heat flux across the top and bottom boundaries (right) as a function of time at* $Ra = 10^6$.

will achieve the desired effect of computing with $Ra = 10^6$.

Play time 2: Thinking about finer meshes. In our computations for $Ra = 10^4$ we used a 16×16 mesh and obtained a value for the heat flux that differed from the generally accepted value from Blankenbach *et al.* [BBC+89] by less than 1%. However, it may be interesting to think about computing even more accurately. This is easily done by using a finer mesh, for example. In the parameter file above, we have chosen the mesh setting as follows:

```
subsection Mesh refinement
  set Initial global refinement                = 4
  set Initial adaptive refinement              = 0
  set Time steps between mesh refinement       = 0
end
```

We start out with a box geometry consisting of a single cell that is refined four times. Each time we split each cell into its 4 children, obtaining the 16×16 mesh already mentioned. The other settings indicate that we do not want to refine the mesh adaptively at all in the first time step, and a setting of zero for the last parameter means that we also never want to adapt the mesh again at a later time. Let us stick with the never-changing, globally refined mesh for now (we will come back to adaptive mesh refinement again at a later time) and only vary the initial global refinement. In particular, we could choose the parameter `Initial global refinement` to be 5, 6, or even larger. This will get us closer to the exact solution albeit at the expense of a significantly increased computational time.

A better strategy is to realize that for $Ra = 10^4$, the flow enters a steady state after settling in during the first part of the simulation (see, for example, the graphs in Fig. 7). Since we are not particularly interested in this initial transient process, there is really no reason to spend CPU time using a fine mesh and correspondingly small time steps during this part of the simulation (remember that each refinement results in four times as many cells in 2d and a time step half as long, making reaching a particular time at least 8 times as expensive, assuming that all solvers in ASPECT scale perfectly with the number of cells). Rather, we can use a parameter in the ASPECT input file that let's us increase the mesh resolution at later times. To this end, let us use the following snippet for the input file:

```
subsection Mesh refinement
  set Initial global refinement                = 3
  set Initial adaptive refinement              = 0
  set Time steps between mesh refinement       = 0
  set Additional refinement times              = 0.2, 0.3, 0.4
  set Refinement fraction                      = 1.0
```

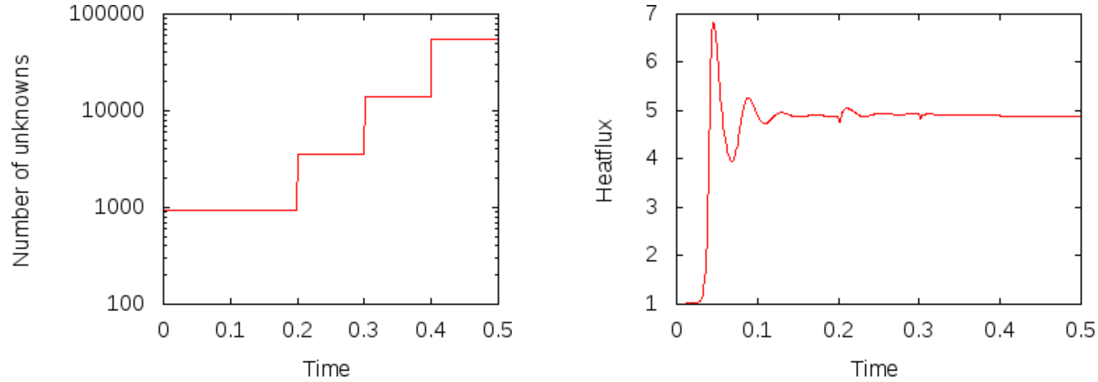


Figure 11: *Convection in a box: Refinement in stages. Total number of unknowns in each time step, including all velocity, pressure and temperature unknowns (left) and heat flux across the top boundary (right).*

```
  set Coarsening fraction                      = 0.0
end
```

What this does is the following: We start with an 8×8 mesh (3 times globally refined) but then at times $t = 0.2, 0.3$ and 0.4 we refine the mesh using the default refinement indicator (which one this is is not important because of the next statement). Each time, we refine, we refine a fraction 1.0 of the cells, i.e., *all* cells and we coarsen a fraction of 0.0 of the cells, i.e. no cells at all. In effect, at these additional refinement times, we do another global refinement, bringing us to refinement levels 4, 5 and finally 6.

Fig. 11 shows the results. In the left panel, we see how the number of unknowns grows over time (note the logscale for the y-axis). The right panel displays the heat flux. The jumps in the number of cells is clearly visible in this picture as well. This may be surprising at first but remember that the mesh is clearly too coarse in the beginning to really resolve the flow and so we should expect that the solution changes significantly if the mesh is refined. This effect becomes smaller with every additional refinement and is barely visible at the last time this happens, indicating that the mesh before this refinement step may already have been fine enough to resolve the majority of the dynamics.

In any case, we can compare the heat fluxes we obtain at the end of these computations: With a globally four times refined mesh, we get a value of 4.931 (an error of approximately 1% against the accepted value from Blankenbach, 4.884409 ± 0.00001). With a globally five times refined mesh we get 4.914 (an error of 0.6%) and with the mesh generated using the procedure above we get 4.895 with the four digits printed on the screen[18] (corresponding to an error of 0.2%). In other words, our simple procedure of refining the mesh during the simulation run yields an accuracy of three times smaller than using the globally refined approach even though the compute time is not much larger than that necessary for the 5 times globally refined mesh.

Play time 3: Changing the finite element in use. Another way to increase the accuracy of a finite element computation is to use a higher polynomial degree for the finite element shape functions. By default, ASPECT uses quadratic shape functions for the velocity and the temperature and linear ones for the pressure. However, this can be changed with a single number in the input file.

Before doing so, let us consider some aspects of such a change. First, looking at the pictures of the solution in Fig. 6, one could surmise that the quadratic elements should be able to resolve the velocity field reasonably well given that it is rather smooth. On the other hand, the temperature field has a boundary layer at the top and bottom. One could conjecture that the temperature polynomial degree is therefore the limiting factor and not the polynomial degree for the flow variables. We will test this conjecture below. Secondly, given the nature of the equations, increasing the polynomial degree of the flow variables increases

[18] The statistics file gives this value to more digits: 4.89488768. However, these are clearly more digits than the result is accurate.

the cost to solve these equations by a factor of $\frac{22}{9}$ in 2d (you can get this factor by counting the number of degrees of freedom uniquely associated with each cell) but leaves the time step size and the cost of solving the temperature system unchanged. On the other hand, increasing the polynomial degree of the temperature variable from 2 to 3 requires $\frac{9}{4}$ times as many degrees of freedom for the temperature and also requires us to reduce the size of the time step by a factor of $\frac{2}{3}$. Because solving the temperature system is not a dominant factor in each time step (see the timing results shown at the end of the screen output above), the reduction in time step is the only important factor. Overall, increasing the polynomial degree of the temperature variable turns out to be the cheaper of the two options.

Following these considerations, let us add the following section to the parameter file:

```
subsection Discretization
  set Stokes velocity polynomial degree       = 2
  set Temperature polynomial degree           = 3
end
```

This leaves the velocity and pressure shape functions at quadratic and linear polynomial degree but increases the polynomial degree of the temperature from quadratic to cubic. Using the original, four times globally refined mesh, we then get the following output:

```
Number of active cells: 256 (on 5 levels)
Number of degrees of freedom: 4,868 (2,178+289+2,401)

*** Timestep 0:  t=0 seconds
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 30+5 iterations.

[... ...]

*** Timestep 1619:  t=0.499807 seconds
   Solving temperature system... 8 iterations.
   Solving Stokes system... 5 iterations.

   Postprocessing:
     RMS, max velocity:                  42.9 m/s, 69.5 m/s
     Temperature min/avg/max:            0 K, 0.5 K, 1 K
     Heat fluxes through boundary parts: -0.004622 W, 0.004624 W, -4.878 W, 4.878 W


+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    |       127s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Assemble Stokes system          |      1620 |      3.03s |       2.4% |
| Assemble temperature system     |      1620 |      75.7s |        60% |
| Build Stokes preconditioner     |         1 |    0.0422s |     0.033% |
| Build temperature preconditioner|      1620 |      21.7s |        17% |
| Solve Stokes system             |      1620 |      10.3s |       8.1% |
| Solve temperature system        |      1620 |       4.9s |       3.8% |
| Initialization                  |         2 |    0.0246s |     0.019% |
| Postprocessing                  |      1620 |      8.05s |       6.3% |
| Setup dof systems               |         1 |    0.0438s |     0.034% |
+---------------------------------+-----------+------------+------------+
```

Note here that the heat flux through the top and bottom boundaries is now computed as 4.878, an error of 0.13%. This is 4 times more accurate than the once more globally refined mesh with the original quadratic

elements, at a cost significantly smaller. Furthermore, we can of course combine this with the mesh that is gradually refined as simulation time progresses, and we then get a heat flux that is equal to 4.8843, only 0.002% away from the accepted value!

As a final remark, to test our hypothesis that it was indeed the temperature polynomial degree that was the limiting factor, we can increase the Stokes polynomial degree to 3 while leaving the temperature polynomial degree at 2. A quick computation shows that in that case we get a heat flux of 4.931 – exactly the same value as we got initially with the lower order Stokes element. In other words, at least for this testcase, it really was the temperature variable that limits the accuracy.

6.2.2 Convection in a 3d box

The world is not two-dimensional. While the previous section introduced a number of the knobs one can play with with ASPECT, things only really become interesting once one goes to 3d. The setup from the previous section is easily adjusted to this and in the following, let us walk through some of the changes we have to consider when going from 2d to 3d. The full input file that contains these modifications and that was used for the simulations we will show subsequently can be found at `cookbooks/convection-box-3d.prm`.

The first set of changes has to do with the geometry: it is three-dimensional, and we will have to address the fact that a box in 3d has 6 sides, not the 4 we had previously. The documentation of the "box" geometry (see Section 5.32) states that these sides are numbered as follows: "*in 3d, boundary indicators 0 through 5 indicate left, right, front, back, bottom and top boundaries.*" Recalling that we want tangential flow all around and want to fix the temperature to known values at the bottom and top, the following will make sense:

```
set Dimension                              = 3

subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 1
    set Y extent = 1
    set Z extent = 1
  end
end

subsection Boundary temperature model
  set Model name = box

  subsection Box
    set Bottom temperature = 1
    set Top temperature    = 0
  end
end

subsection Model settings
  set Fixed temperature boundary indicators   = bottom, top

  set Zero velocity boundary indicators       =
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = left, right, front, back, bottom, top
end
```

The next step is to describe the initial conditions. As before, we will use an instably layered medium but the perturbation now needs to be both in x- and y-direction

```
subsection Initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,y,z
    set Function constants  = p=0.01, L=1, pi=3.1415926536, k=1
    set Function expression = (1.0-z) - p*cos(k*pi*x/L)*sin(pi*z)*y^3
  end
end
```

The third issue we need to address is that we can likely not afford a mesh as fine as in 2d. We choose a mesh that is refined 3 times globally at the beginning, then 3 times adaptively, and is then adapted every 15 time steps. We also allow one additional mesh refinement in the first time step following $t = 0.003$ once the initial instability has given way to a more stable pattern:

```
subsection Mesh refinement
  set Initial global refinement                = 3
  set Initial adaptive refinement              = 3
  set Time steps between mesh refinement       = 15

  set Additional refinement times              = 0.003
end
```

Finally, as we have seen in the previous section, a computation with $Ra = 10^4$ does not lead to a simulation that is exactly exciting. Let us choose $Ra = 10^6$ instead (the mesh chosen above with up to 7 refinement levels after some time is fine enough to resolve this). We can achieve this in the same way as in the previous section by choosing $\alpha = 10^{-10}$ and setting

```
subsection Gravity model
  set Model name = vertical

  subsection Vertical
    set Magnitude = 1e16   # = Ra / Thermal expansion coefficient
  end
end
```

This has some interesting implications. First, a higher Rayleigh number makes time scales correspondingly smaller; where we generated graphical output only once every 0.01 time units before, we now need to choose the corresponding increment smaller by a factor of 100:

```
subsection Postprocess
  set List of postprocessors = velocity statistics, temperature statistics, ...
                       ...heat flux statistics, visualization

  subsection Visualization
    set Time between graphical output = 0.0001
  end
end
```

Secondly, a simulation like this – in 3d, with a significant number of cells, and for a significant number of time steps – will likely take a good amount of time. The computations for which we show results below was run using 64 processors by running it using the command `mpirun -n 64 ./aspect convection-box-3d.prm`. If the machine should crash during such a run, a significant amount of compute time would be lost if we had to run everything from the start. However, we can avoid this by periodically checkpointing the state of the computation:

```
subsection Checkpointing
```

```
  set Steps between checkpoint = 50
end
```

If the computation does crash (or if a computation runs out of the time limit imposed by a scheduling system), then it can be restarted from such checkpointing files (see the parameter `Resume computation` in Section 5.2).

Running with this input file requires a bit of patience[19] since the number of degrees of freedom is just so large: it starts with a bit over 330,000...

```
Running with 64 MPI tasks.
Number of active cells: 512 (on 4 levels)
Number of degrees of freedom: 20,381 (14,739+729+4,913)

*** Timestep 0:  t=0 seconds
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 18 iterations.

Number of active cells: 1,576 (on 5 levels)
Number of degrees of freedom: 63,391 (45,909+2,179+15,303)

*** Timestep 0:  t=0 seconds
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 19 iterations.

Number of active cells: 3,249 (on 5 levels)
Number of degrees of freedom: 122,066 (88,500+4,066+29,500)

*** Timestep 0:  t=0 seconds
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 20 iterations.

Number of active cells: 8,968 (on 5 levels)
Number of degrees of freedom: 331,696 (240,624+10,864+80,208)

*** Timestep 0:  t=0 seconds
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 21 iterations.
[...]
```

...but then increases quickly to around 2 million as the solution develops some structure and, after time $t = 0.003$ where we allow for an additional refinement, increases to over 10 million where it then hovers between 8 and 14 million with a maximum of 15,147,534. Clearly, even on a reasonably quick machine, this will take some time: running this on a machine bought in 2011, doing the 10,000 time steps to get to $t = 0.0219$ takes approximately 484,000 seconds (about five and a half days).

The structure or the solution is easiest to grasp by looking at isosurfaces of the temperature. This is shown in Fig. 12 and you can find a movie of the motion that ensues from the heating at the bottom at `http://www.youtube.com/watch?v=_bKqU_P4j48`. The simulation uses adaptively changing meshes that are fine in rising plumes and sinking blobs and are coarse where nothing much happens. This is most easily seen in the movie at `http://www.youtube.com/watch?v=CzCKYyR-cmg`. Fig. 13 shows some of these meshes,

[19] For computations of this size, one should test a few time steps in debug mode but then, of course, switch to running the actual computation in optimized mode – see Section 4.3.

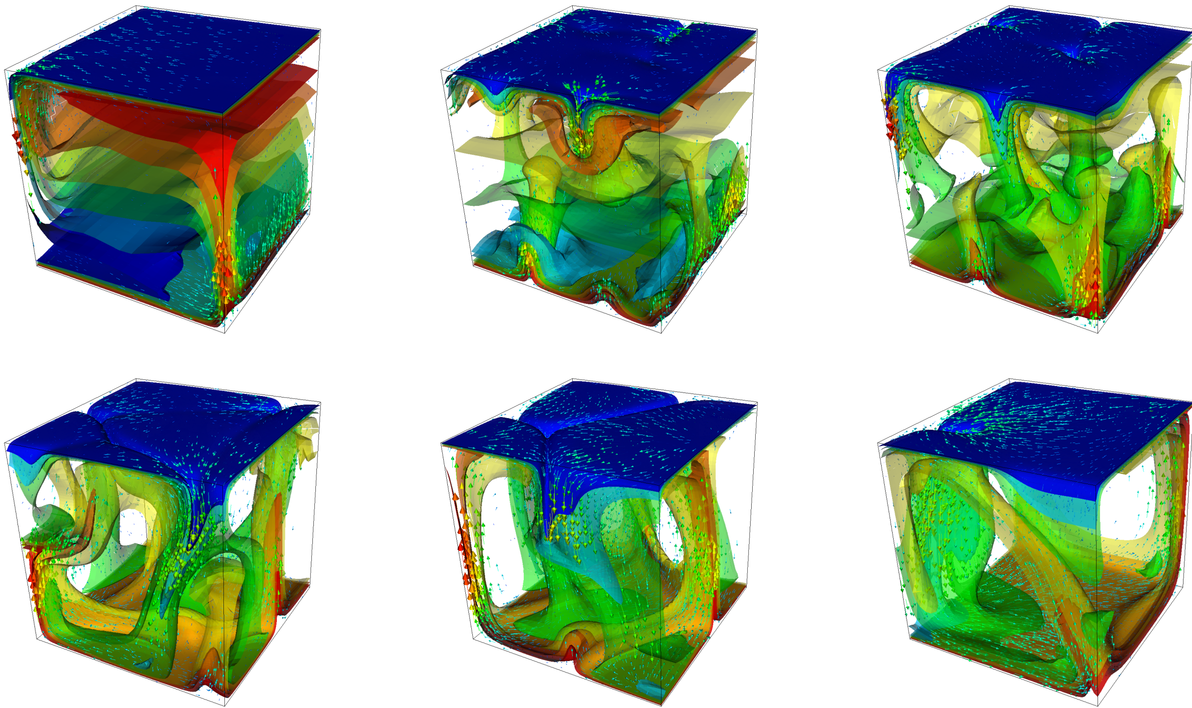


Figure 12: *Convection in a 3d box: Temperature isocontours and some velocity vectors at the first time step after times* $t = 0.001, 0.004, 0.006$ *(top row, left to right) an* $t = 0.01, 0.013, 0.018$ *(bottom row).*

though still pictures do not do the evolving nature of the mesh much justice. The effect of increasing the Rayleigh number is apparent when comparing the size of features with, for example, the picture at the right of Fig. 6. In contrast to that picture, the simulation is also clearly non-stationary.

As before, we could analyze all sorts of data from the statistics file but we will leave this to those interested in specific data. Rather, Fig. 14 only shows the upward heat flux through the bottom and top boundaries of the domain as a function of time.[20] The figure reinforces a pattern that can also be seen by watching the movie of the temperature field referenced above, namely that the simulation can be subdivided into three distinct phases. The first phase corresponds to the initial overturning of the instable layering of the temperature field and is associated with a large spike in heat flux as well as large velocities (not shown here). The second phase, until approximately $t = 0.01$ corresponds to a relative lull: some plumes rise up, but not very fast because the medium is now stably layered but not fully mixed. This can be seen in the relatively low heat fluxes, but also in the fact that there are almost horizontal temperature isosurfaces in the second of the pictures in Fig. 12. After that, the general structure of the temperature field is that the interior of the domain is well mixed with a mostly constant average temperature and thin thermal boundary layers at the top and bottom from which plumes rise and sink. In this regime, the average heat flux is larger but also more variable depending on the number of plumes currently active. Many other analyses would be possible by using what is in the statistics file or by enabling additional postprocessors.

[20]Note that the statistics file actually contains the *outward* heat flux for each of the six boundaries, which corresponds to the *negative* of upward flux for the bottom boundary. The figure therefore shows the negative of the values available in the statistics file.

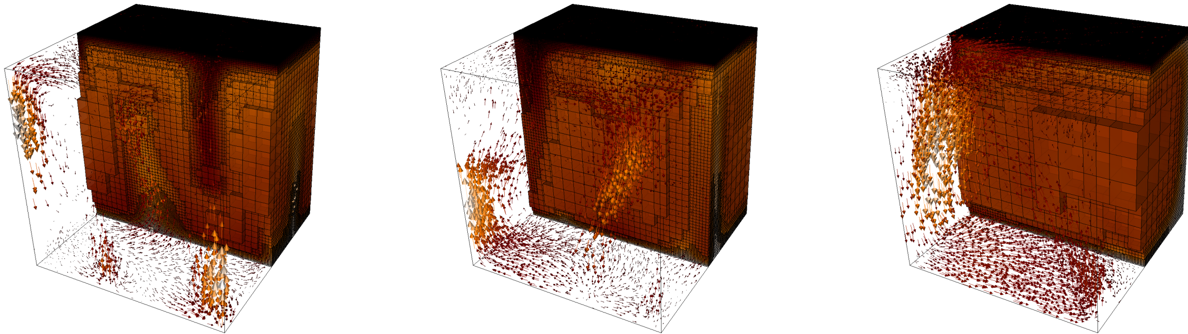


Figure 13: *Convection in a 3d box: Meshes and large-scale velocity field for the third, fourth and sixth of the snapshots shown in Fig. 12.*

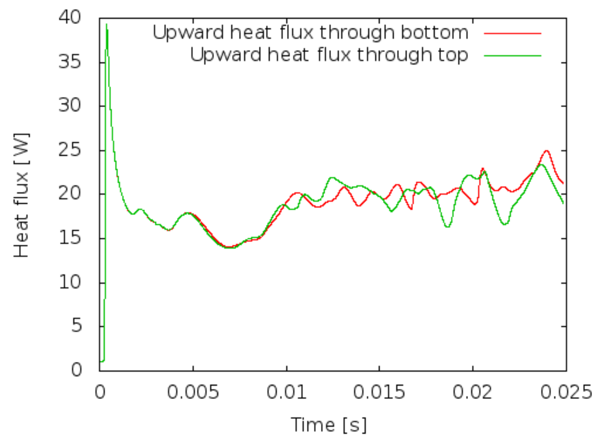


Figure 14: *Convection in a 3d box: Upward heat flux through the bottom and top boundaries as a function of time.*

6.2.3 Convection in a box with prescribed, variable velocity boundary conditions

A similarly simple setup to the ones considered in the previous subsections is to equip the model we had with a different set of boundary conditions. There, we used slip boundary conditions, i.e., the fluid can flow tangentially along the four sides of our box but this tangential velocity is unspecified. On the other hand, in many situations, one would like to actually prescribe the tangential flow velocity as well. A typical application would be to use boundary conditions at the top that describe experimentally determined velocities of plates. This cookbook shows a simple version of something like this. To make it slightly more interesting, we choose a 2×1 domain in 2d.

Like for many other things, ASPECT has a set of plugins for prescribed velocity boundary values (see Sections 5.20 and 7.3.6). These plugins allow one to write sophisticated models for the boundary velocity on parts or all of the boundary, but there is also one simple implementation that just takes a formula for the components of the velocity.

To illustrate this, let us consider the `cookbooks/platelike-boundary.prm` input file. It essentially extends the input file considered in the previous example. The part of this file that we are particularly interested in in the current context is the selection of the kind of boundary conditions on the four sides of the box geometry, which we do using a section like this:

```
subsection Model settings
  set Fixed temperature boundary indicators   = bottom, top
  set Zero velocity boundary indicators       =
  set Tangential velocity boundary indicators = left, right, bottom
  set Prescribed velocity boundary indicators = top: function
end
```

Following the convention for numbering boundaries described in the previous section, this means that we prescribe a fixed temperature at the bottom and top sides of the box (boundary numbers two and three). We use tangential flow at boundaries zero, one and two (left, right and bottom). Finally, the last entry above is a comma separated list (here with only a single element) of pairs consisting of the number of a boundary and the name of the prescribed velocity boundary model to be used on this boundary. Here, we use the `function` boundary model, which allows us to provide a function-like notation for the components of the velocity vector at the boundary.

The second part we need is that we actually describe the function that sets the velocity. We do this as follows:

```
subsection Boundary velocity model
  subsection Function
    set Variable names      = x,z,t
    set Function constants  = pi=3.1415926
    set Function expression = if(x>1+sin(0.5*pi*t), 1, -1); 0
  end
end
```

The first of these gives names to the components of the position vector (here, we are in 2d and we use x and z as spatial variable names) and the time. We could have left this entry at its default, `x,y,t`, but since we often think in terms of "depth" as the vertical direction, let us use `z` for the second coordinate. In the second parameter we define symbolic constants that can be used in the formula for the velocity that is specified in the last parameter. This formula needs to have as many components as there are space dimensions, separated by semicolons. As stated, this means that we prescribe the (horizontal) x-velocity and set the vertical velocity to zero. The horizontal component is here either 1 or -1, depending on whether we are to the right or the left of the point $1 + \sin(\pi t/2)$ that is moving back and forth with time once every four time units. The `if` statement understood by the parser we use for these formulas has the syntax `if(condition, value-if-true, value-if-false)`.

Note: While you can enter most any expression into the parser for these velocity boundary conditions, not all make sense. In particular, if you use an incompressible medium like we do here, then you need to make sure that either the flow you prescribe is indeed tangential, or that at least the flow into and out of the boundary this function applies to is balanced so that in sum the amount of material in the domain stays constant.
It is in general not possible for ASPECT to verify that a given input is sensible. However, you will quickly find out if it isn't: The linear solver for the Stokes equations will simply not converge. For example, if your function expression in the input file above read

```
if(x>1+sin(0.5*pi*t), 1, -1); 1
```

then at the time of writing this you would get the following error message:

```
*** Timestep 0:  t=0 seconds
 Solving temperature system...  0 iterations.
 Rebuilding Stokes preconditioner...
 Solving Stokes system...

...some timing output ...



----------------------------------------------------
Exception on processing:
Iterative method reported convergence failure in step 9539 with residual 6.0552
Aborting!
----------------------------------------------------
```

The reason is, of course, that there is no incompressible (divergence free) flow field that allows for a constant vertical outflow component along the top boundary without corresponding inflow anywhere else.

The remainder of the setup is described in the following, complete input file:

Is there a rea son that this i not a listing environment?

```
############### Global parameters

set Dimension                              = 2
set Start time                             = 0
set End time                               = 20
set Use years in output instead of seconds = false
set Output directory                       = output


############### Parameters describing the model
# Let us here choose again a box domain of size 2x1
# where we fix the temperature at the bottom and top,
# allow free slip along the bottom, left and right,
# and prescribe the velocity along the top using the
# 'function' description.

subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 2
    set Y extent = 1
  end
end
```

```
subsection Model settings
  set Fixed temperature boundary indicators   = bottom, top
  set Zero velocity boundary indicators       =
  set Tangential velocity boundary indicators = left, right, bottom
  set Prescribed velocity boundary indicators = top: function
end


# We then set the temperature to one at the bottom and zero
# at the top:
subsection Boundary temperature model
  set Model name = box

  subsection Box
    set Bottom temperature = 1
    set Top temperature    = 0
  end
end


# The velocity along the top boundary models a spreading
# center that is moving left and right:
subsection Boundary velocity model
  subsection Function
    set Variable names      = x,z,t
    set Function constants  = pi=3.1415926
    set Function expression = if(x>1+sin(0.5*pi*t), 1, -1); 0
  end
end


# We then choose a vertical gravity model and describe the
# initial temperature with a vertical gradient. The default
# strength for gravity is one. The material model is the
# same as before.
subsection Gravity model
  set Model name = vertical
end


subsection Initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,z
    set Function expression = (1-z)
  end
end


subsection Material model
  set Model name = simple

  subsection Simple model
    set Thermal conductivity          = 1e-6
```

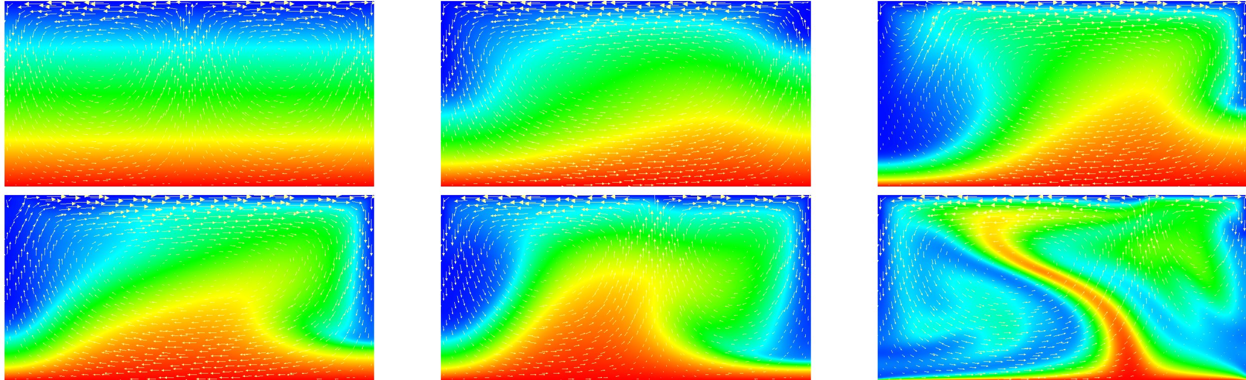


Figure 15: *Variable velocity boundary conditions: Temperature and velocity fields at the initial time (top left) and at various other points in time during the simulation.*

```
    set Thermal expansion coefficient = 1e-4
    set Viscosity                     = 1
  end
end


# The final part of this input file describes how many times the
# mesh is refined and what to do with the solution once computed
subsection Mesh refinement
  set Initial adaptive refinement        = 0
  set Initial global refinement          = 5
  set Time steps between mesh refinement = 0
end


subsection Postprocess
  set List of postprocessors = visualization, temperature statistics, heat flux statistics

  subsection Visualization
    set Time between graphical output = 0.1
  end
end
```

This model description yields a setup with a Rayleigh number of 200 (taking into account that the domain has size 2). It would, thus, be dominated by heat conduction rather than convection if the prescribed velocity boundary conditions did not provide a stirring action. Visualizing the results of this simulation[21] yields images like the ones shown in Fig. 15.

6.2.4 Using passive and active compositional fields

One frequently wants to track where material goes, either because one simply wants to see where stuff ends up (e.g., to determine if a particular model yields mixing between the lower and upper mantle) or because the material model in fact depends not only pressure, temperature and location but also on the mass fractions of certain chemical or other species. We will refer to the first case as *passive* and the latter as *active* to indicate

[21] In fact, the pictures are generated using a twice more refined mesh to provide adequate resolution. We keep the default setting of five global refinements in the parameter file as documented above to keep compute time reasonable when using the default settings.

the role of the additional quantities whose distribution we want to track. We refer to the whole process as *compositional* since we consider quantities that have the flavor of something that denotes the composition of the material at any given point.

There are basically two ways to achieve this: one can advect a set of particles ("tracers") along with the velocity field, or one can advect along a field. In the first case, where the closest particle came from indicates the value of the concentration at any given position. In the latter case, the concentration(s) at any given position is simply given by the value of the field(s) at this location.

ASPECT implements both strategies, at least to a certain degree. In this cookbook, we will follow the route of advected fields.

The passive case. We will consider the exact same situation as in the previous section but we will ask where the material that started in the bottom 20% of the domain ends up, as well as the material that started in the top 20%. For the moment, let us assume that there is no material between the materials at the bottom, the top, and the middle. The way to describe this situation is to simply add the following block of definitions to the parameter file (you can find the full parameter file in cookbooks/compositional-passive.prm:

```
# This is the new part: We declare that there will
# be two compositional fields that will be
# advected along. Their initial conditions are given by
# a function that is one for the lowermost 0.2 height
# units of the domain and zero otherwise in the first case,
# and one in the top most 0.2 height units in the latter.
subsection Compositional fields
  set Number of fields = 2
end

subsection Compositional initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,y
    set Function expression = if(y<0.2, 1, 0) ; if(y>0.8, 1, 0)
  end
end
```

Running this simulation yields results such as the ones shown in Fig. 16 where we show the values of the functions $c_1(\mathbf{x}, t)$ and $c_2(\mathbf{x}, t)$ at various times in the simulation. Because these fields were one only inside the lowermost and uppermost parts of the domain, zero everywhere else, and because they have simply been advected along with the flow field, the places where they are larger than one half indicate where material has been transported to so far.[22]

Fig. 16 shows one aspect of compositional fields that occasionally makes them difficult to use for very long time computations. The simulation shown here runs for 20 time units, where every cycle of the spreading center at the top moving left and right takes 4 time units, for a total of 5 such cycles. While this is certainly no short-term simulation, it is obviously visible in the figure that the interface between the materials has diffused over time. Fig. 17 shows a zoom into the center of the domain at the final time of the simulation. The figure only shows values that are larger than 0.5, and it looks like the transition from red or blue to the edge of the shown region is no wider than 3 cells. This means that the computation is not overly diffusive but it is nevertheless true that this method has difficulty following long and thin filaments.[23] This is an area in which ASPECT may see improvements in the future.

[22]Of course, this interpretation suggests that we could have achieved the same goal by encoding everything into a single function – that would, for example, have had initial values one, zero and minus one in the three parts of the domain we are interested in.

[23]We note that this is no different for tracers where the position of tracers has to be integrated over time and is subject to numerical error. In simulations, their location is therefore not the exact one but also subject to a diffusive process resulting

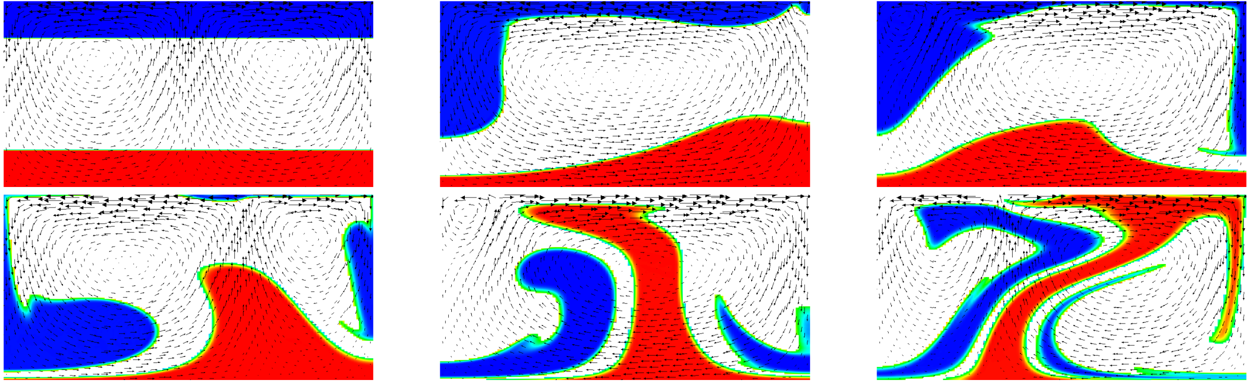


Figure 16: *Passive compositional fields: The figures show, at different times in the simulation, the velocity field along with those locations where the first compositional field is larger than 0.5 (in red, indicating the locations where material from the bottom of the domain has gone) as well as where the second compositional field is larger than 0.5 (in blue, indicating material from the top of the domain. The results were obtained with two more global refinement steps compared to the `cookbooks/compositional-passive.prm` input file.*

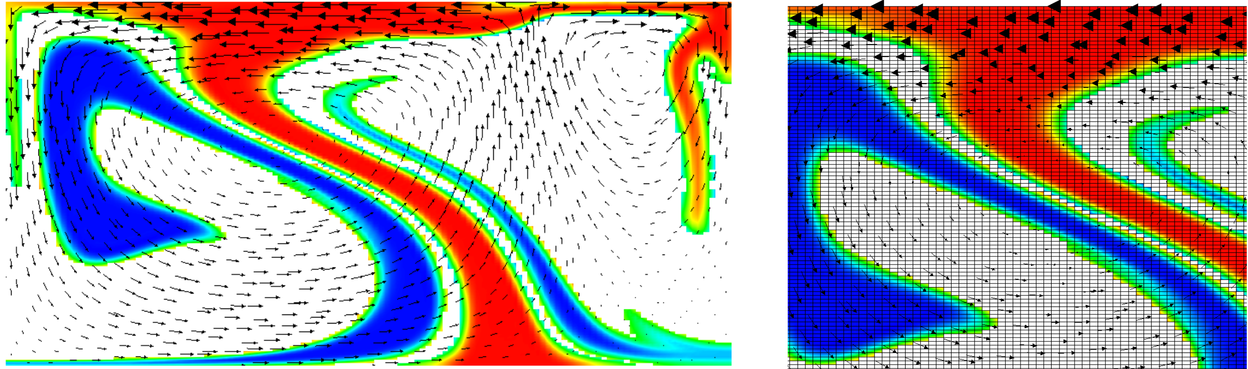


Figure 17: *Passive compositional fields: A later image of the simulation corresponding to the sequence shown in Fig. 16 (left) and zoom-in on the center, also showing the mesh (right).*

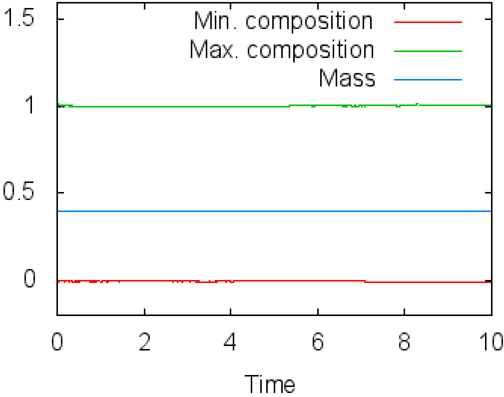


Figure 18: *Passive compositional fields: Minimum and maximum of the first compositional variable over time, as well as the mass $m_1(t) = \int_\Omega c_1(\mathbf{x}, t)$ stored in this variable.*

A different way of looking at the quality of compositional fields as opposed to tracers is to ask whether they conserve mass. In the current context, the mass contained in the ith compositional field is $m_i(t) = \int_\Omega c_i(\mathbf{x}, t)$. This can easily be achieve in the following way, by adding the `composition statistics` postprocessor:

```
subsection Postprocess
  set List of postprocessors = visualization, temperature statistics, composition statistics
end
```

While the scheme we use to advect the compositional fields is not strictly conservative, it is almost perfectly so in practice. For example, in the computations shown in this section (using two additional global mesh refinements over the settings in the parameter file `cookbooks/compositional-passive.prm`), Fig. 18 shows the maximal and minimal values of the first compositional fields over time, along with the mass $m_1(t)$ (these are all tabulated in columns of the statistics file, see Sections 4.1 and 4.4.2). While the maximum and minimum fluctuate slightly due to the instability of the finite element method in resolving discontinuous functions, the mass appears stable at a value of 0.403646 (the exact value, namely the area that was initially filled by each material, is 0.4; the difference is a result of the fact that we can't exactly represent the step function on our mesh with the finite element space). In fact, the maximal difference in this value between time steps 1 and 500 is only $1.1 \cdot 10^{-6}$. In other words, these numbers show that the compositional field approach is almost exactly mass conservative.

The active case. The next step, of course, is to make the flow actually depend on the composition. After all, compositional fields are not only intended to indicate where material come from, but also to indicate the properties of this material. In general, the way to achieve this is to write material models where the density, viscosity, and other parameters depend on the composition, taking into account what the compositional fields actually denote (e.g., if they simply indicate the origin of material, or the concentration of things like olivine, perovskite, ...). The construction of material models is discussed in much greater detail in Section 7.3.1; we do not want to revisit this issue here and instead choose – once again – the simplest material model that is implemented in ASPECT: the `simple` model.

The place where we are going to hook in a compositional dependence is the density. In the `simple` model, the density is fundamentally described by a material that expands linearly with the temperature; for small density variations, this corresponds to a density model of the form $\rho(T) = \rho_0(1 - \alpha(T - T_0))$. This is, by virtue of its simplicity, the most often considered density model. But the `simple` model also has a hook to make the density depend on the first compositional field $c_1(\mathbf{x}, t)$, yielding a dependence of the form $\rho(T) = \rho_0(1 - \alpha(T - T_0)) + \gamma c_1$. Here, let us choose $\rho_0 = 1, \alpha = 0.01, T_0 = 0, \gamma = 100$. The rest of our model setup will be as in the passive case above. Because the temperature will be between zero and one, the temperature induced density variations will be restricted to 0.01, whereas the density variation by origin of the material is 100. This should make sure that dense material remains at the bottom despite the fact that it is hotter than the surrounding material.[24]

This setup of the problem can be described using an input file that is almost completely unchanged from the passive case. The only difference is the use of the following section (the complete input file can be found in `cookbooks/compositional-active.prm`:

```
subsection Material model
  set Model name = simple

  subsection Simple model
    set Thermal conductivity                            = 1e-6
```

from numerical inaccuracies. Furthermore, in long thin filaments, the number of tracers per cell often becomes too small and new tracers have to be inserted; their properties are then interpolated from the surrounding tracers, a process that also incurs a smoothing penalty.

[24]The actual values do not matter as much here. They are chosen in such a way that the system – previously driven primarily by the velocity boundary conditions at the top – now also feels the impact of the density variations. To have an effect, the buoyancy induced by the density difference between materials must be strong enough to balance or at least approach the forces exerted by whatever is driving the velocity at the top.

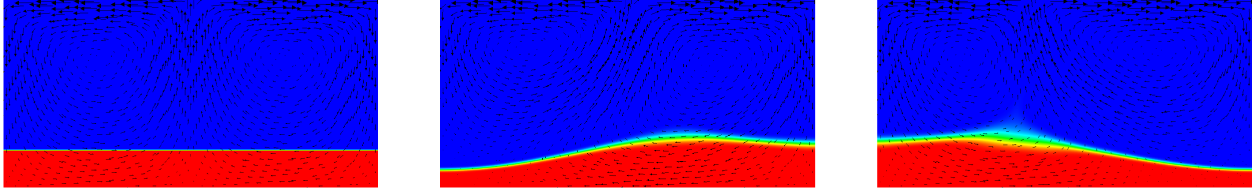


Figure 19: *Active compositional fields: Compositional field 1 at the time $t = 0, 10, 20$. Compared to the results shown in Fig. 16 it is clear that the heavy material stays at the bottom of the domain now. The effect of the density on the velocity field is also clearly visible by noting that at all three times the spreading center at the top boundary is in exactly the same position; this would result in exactly the same velocity field if the density and temperature were constant.*

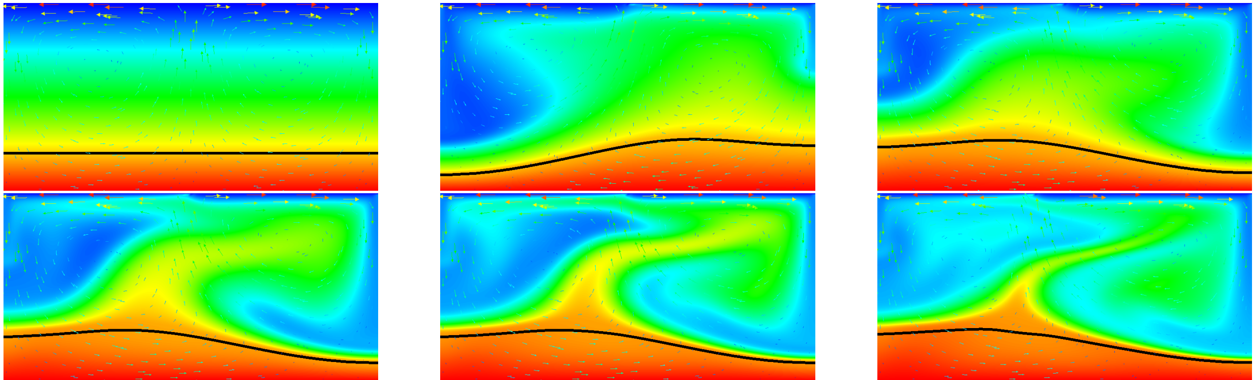


Figure 20: *Active compositional fields: Temperature fields at $t = 0, 2, 4, 8, 12, 20$. The black line is the isocontour line $c_1(\mathbf{x}, t) = 0.5$ delineating the position of the dense material at the bottom.*

```
    set Thermal expansion coefficient                     = 0.01
    set Viscosity                                         = 1
    set Reference density                                 = 1
    set Reference temperature                             = 0
    set Density differential for compositional field 1 = 0.1
  end
end
```

To debug the model, we will also want to visualize the density in our graphical output files. This is done using the following addition to the postprocessing section, using the `density` visualization plugin:

```
subsection Postprocess
  set List of postprocessors = visualization, temperature statistics, composition statistics

  subsection Visualization
    set List of output variables = density
    set Time between graphical output = 0.1
  end
end
```

Results of this model are visualized in Fig.s 19 and 20. What is visible is that over the course of the simulation, the material that starts at the bottom of the domain remains there. This can only happen if the circulation is significantly affected by the high density material once the interface starts to become non-horizontal, and this is indeed visible in the velocity vectors. As a second consequence, if the material at

the bottom does not move away, then there needs to be a different way for the heat provided at the bottom to get through the bottom layer: either there must be a secondary convection system in the bottom layer, or heat is simply conducted. The pictures in the figure seem to suggest that the latter is the case.

It is easy, using the outline above, to play with the various factors that drive this system, namely:

- The magnitude of the velocity prescribed at the top.
- The magnitude of the velocities induced by thermal buoyancy, as resulting from the magnitude of gravity and the thermal expansion coefficient.
- The magnitude of the velocities induced by compositional variability, as described by the coefficient γ and the magnitude of gravity.

Using the coefficients involved in these considerations, it is trivially possible to map out the parameter space to find which of these effects is dominant. As mentioned in discussing the values in the input file, what is important is the *relative* size of these parameters, not the fact that currently the density in the material at the bottom is 100 times larger than in the rest of the domain, an effect that from a physical perspective clearly makes no sense at all.

The active case with reactions. *This section was contributed by Juliane Dannberg and René Gaßmöller.*

In addition, there are setups where one wants the compositional fields to interact with each other. One example would be material upwelling at a mid-ocean ridge and changing the composition to that of oceanic crust when it reaches a certain depth. In this cookbook, we will describe how this kind of behaviour can be achieved by using the `composition reaction` function of the material model.

We will consider the exact same setup as in the previous paragraphs, except for the initial conditions and properties of the two compositional fields. There is one material that initially fills the bottom half of the domain and is less dense than the material above. In addition, there is another material that only gets created when the first material reaches the uppermost 20% of the domain, and that has a higher density. This should cause the first material to move upwards, get partially converted to the second material, which then sinks down again. This means we want to change the initial conditions for the compositional fields:

```
subsection Compositional initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,z
    set Function expression = if(z<0.5, 1, 0); 0
  end
end
```

Moreover, instead of the `simple` material model, we will use the `composition reaction` material model, which basically behaves in the same way, but can handle two active compositional field and a reaction between those two fields. In the input file, the user defines a depth and above this `reaction depth` the first compositional fields is converted to the second field. This can be done by changing the following section (the complete input file can be found in `cookbooks/composition-reaction.prm`).

```
subsection Material model
  set Model name = composition reaction

  subsection Composition reaction model
    set Thermal conductivity                          = 1e-6
    set Thermal expansion coefficient                 = 0.01
    set Viscosity                                     = 1
    set Density differential for compositional field 1 = -5
    set Density differential for compositional field 2 = 5
```

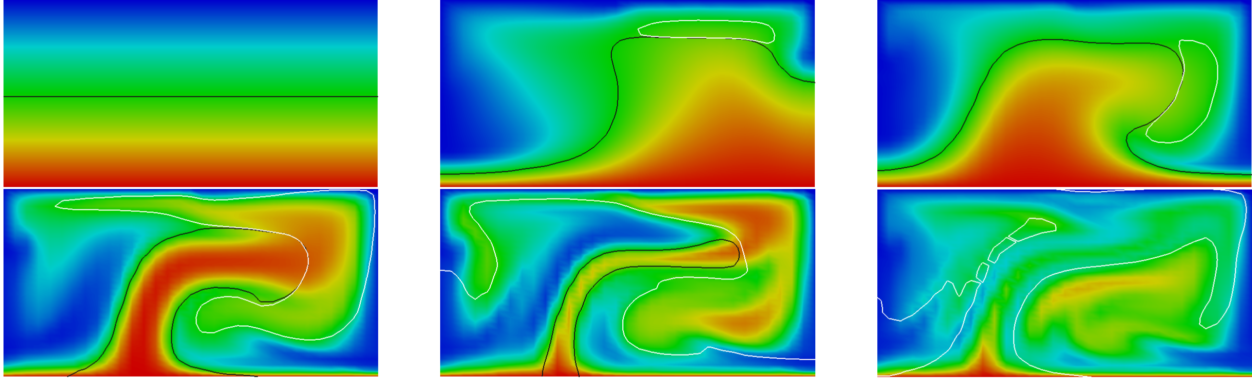


Figure 21: *Reaction between compositional fields: Temperature fields at* $t = 0, 2, 4, 8, 12, 20$. *The black line is the isocontour line* $c_1(\mathbf{x}, t) = 0.5$ *delineating the position of the material starting at the bottom and the white line is the isocontour line* $c_2(\mathbf{x}, t) = 0.5$ *delineating the position of the material that is created by the reaction.*

```
    set Reaction depth                                 = 0.2
  end
end
```

Results of this model are visualized in Fig 21. What is visible is that over the course of the simulation, the material that starts at the bottom of the domain ascends, reaches the reaction depth and gets converted to the second material, which starts to sink down.

6.2.5 Using tracer particles

Using compositional fields to trace where material has come from or is going to has many advantages from a computational point of view. For example, the numerical methods to advect along fields are well developed and we can do so at a cost that is equivalent to one temperature solve for each of the compositional fields. Unless you have many compositional fields, this cost is therefore relatively small compared to the overall cost of a time step. Another advantage is that the value of a compositional field is well defined at every point within the domain. On the other hand, compositional fields over time diffuse initially sharp interfaces, as we have seen in the images of the previous section.

On the other hand, the geodynamics community has a history of using tracers for this purpose. Historically, this may have been because it is conceptually simpler to advect along individual particles rather than whole fields, since it only requires an ODE integrator rather than the stabilization techniques necessary to advect fields. They also provide the appearance of no diffusion, though this is arguable. Leaving the debate whether fields or particles are the way to go aside, ASPECT supports using tracers.

In order to advect tracer particles along with the flow field, one just needs to add the `tracers` postprocessor to the list of postprocessors and specify a few parameters. We do so in the `cookbooks/composition-passive-tracers.prm` input file, which is otherwise just a minor variation of the `cookbooks/composition-passive.prm` case discussed in the previous Section 6.2.4. In particular, the postprocess section now looks like this:

```
subsection Postprocess
  set List of postprocessors = visualization, tracers

  subsection Visualization
    set Time between graphical output = 0.1
  end

  subsection Tracers
```

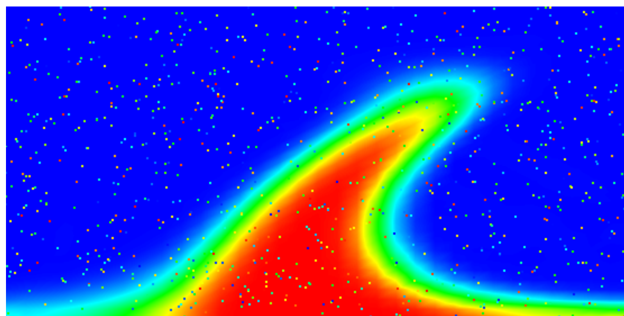


Figure 22: *Passively advected quantities visualized through both a compositional field and a set of 1,000 particles, at $t = 7.2$.*

```
    set Number of tracers        = 1000
    set Time between data output = 0.1
    set Data output format       = vtu
  end
end
```

The 1000 particles we are asking here are initially uniformly distributed throughout the domain and are, at the end of each time step, advected along with the velocity field just computed. (There are a number of options to decide which method to use for advecting particles, see Section 5.93.) We can visualize them by opening both the field-based output files and the ones that correspond to particles (for example, `output/solution-00072.visit` and `output/particles-00072.visit`) and using a pseudo-color plot for the particles, selecting the "id" of particles to color each particle. This results in a plot like the one shown in Fig. 22.

The particles shown here are not too impressive in still pictures since they are colorized by their particle number, which – since particles were initially randomly distributed – is essentially a random number. The purpose of using the particle id to colorize becomes more apparent if you use it when viewing an animation of time steps. There, the different colors of adjacent particles come in handy because they allow the eye to follow the motion of a single particle. This makes it rather intuitive to understand a flow field, but it can of course not be reproduced in a static medium such as this manual.

Using tracer properties The particles in the above example only fulfil the purpose of visualizing the convection pattern. A more meaningful use for particles is to attach properties to them, which may be set at the beginning of the model run, or updated during the model runtime. These properties can then be used for many applications, e.g. tracking an initial property (like the position), evaluating a property at a defined particle path (like the pressure-temperature evolution of a certain piece of rock), or by integrating a quantity along a particle parth (like the integrated strain a certain domain has experienced). We illustrate these properties in the cookbook `cookbooks/composition-passive-tracers-properties.prm`, in which we added the following lines to the `Tracers` subsection.

```
    set List of tracer properties = function, initial composition, initial position, pT path

    subsection Function
      set Variable names      = x,y
      set Function expression = if(y<0.2, 1, 0)
    end
```

These commands add the tracer properties `function`, `pT path`, `initial position` and `initial composition`, which can be used for various purposes. A full list of particle properties can be found in Section 5.93. New

tracer properties can be added as plugins as described in Section 7.2. The here selected properties allow us in this example:

- To compare the final position of a particle with its initial position and therefore determine, how far certain domains travelled during the model runtime.
- To compare the final composition of a particle with its initial composition and therefore determine, which regions underwent the reaction described in Section 6.2.4, and where the material that underwent this reaction got transported to.
- Note that the `function` property of the particles follows the same function as the compositional initial composition of field number one. Therefore, this property should behave identical to the compositional field, except for the reaction term of the compositional field. This allows to compare the error in tracer position to the numerical diffusion of the compositional field.
- If one selects a tracer of a particular id, an output of the pressure and temperature at its current position generated by the `pT path` property over time allows for the creation of a pressure-temperature curve of a certain piece of rock. This property is interesting in many lithosphere to crustal scale models, because it is determining the metamorphic reactions that happen during deformation processes (e.g. in a subduction zone).

> **Note:** ASPECT's tracer implementation is in a preliminary state. While the accuracy and scalability of the implementation is benchmarked, other limitations remain. This in particular means that it is not optimized for performance, and more than a few thousand tracers per process can slow down a model significantly. Moreover, models with an highly adaptive mesh and many tracers do encounter a significant slowdown, because ASPECT only considers the number of degrees of freedom for load balancing across processes and not the number of tracers. Therefore processes that compute the solution for coarse-grid regions have to process many more tracers than other processes. Additionally, the checkpoint/restart functionality for tracers is only implemented in models with a constant number of processes before and after the checkpoint and when the selected tracer properties do not change. These limitations might be removed over time, but for current models the user should be aware of them.

6.2.6 Using a free surface

This section was contributed by Ian Rose.

Free surfaces are numerically challenging but can be useful for self consistently tracking dynamic topography and may be quite important as a boundary condition for tectonic processes like subduction. The parameter file `cookbooks/free-surface.prm` provides a simple example of how to set up a model with a free surface, as well as demonstrates some of the challenges associated with doing so.

ASPECT supports models with a free surface using an Arbitrary Lagrangian-Eulerian framework (see Section 2.11). Most of this is done internally, so you do not need to worry about the details to run this cookbook. Here we demonstrate the evolution of surface topography that results when a blob of hot material rises in the mantle, pushing up the free surface as it does. Usually the amplitude of free surface topography will be small enough that it is difficult to see with the naked eye in visualizations, but the `topography` postprocessor can help by outputting the maximum and minumum topography on the free surface at every time step.

The bulk of the parameter file for this cookbook is similar to previous ones in this manual. We use initial temperature conditions that set up a hot blob of rock in the center of the domain. In the `Model settings` secion you need to give ASPECT a comma separated list of the free surface boundary indicators. In this case, we are dealing with the top boundary of a box in 2D, corresponding to boundary indicator 3.

The main addition is the `Free surface` subsection. There is one main parameter that needs to be set here: the value for the stabilization parameter "theta". If this parameter is zero, then there is no stabilization,

and you are likely to see instabilities develop in the free surface. If this parameter is one then it will do a good job of stabilizing the free surface, but it may overly damp its motions. The default value is 0.5.

Also worth mentioning is the change to the CFL number. Stability concerns typically mean that when making a model with a free surface you will want to take smaller time steps. In general just how much smaller will depend on the problem at hand as well as the desired accuracy.

Following are the sections in the input file specific to this testcase. The full parameter file may be found at cookbooks/free-surface.prm.

```
set CFL number                             = 0.1

subsection Initial conditions
  set Model name = function
  subsection Function
    set Variable names      = x,y
    set Function expression = if( sqrt( (x-250.e3)^2 + (y-100.e3)^2 ) < 25.e3, 200.0, 0.0)
  end
end

subsection Free surface
  set Free surface stabilization theta = 0.5
end

subsection Model settings
  set Include adiabatic heating                = false
  set Include shear heating                    = false
  set Fixed temperature boundary indicators    = left, right, bottom, top
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = left, right, bottom
  set Zero velocity boundary indicators        =
  set Free surface boundary indicators         = top
end

subsection Postprocess
  set List of postprocessors = visualization,topography,velocity statistics,
  subsection Visualization
    set Time between graphical output = 1.e6
  end
end
```

Running this input file will produce results like those in Figure 23. The model starts with a single hot blob of rock which rises in the domain. As it rises, it pushes up the free surface in the middle, creating a topographic high there. This is similar to the kind of dynamic topography that you might see above a mantle plume on Earth. As the blob rises and diffuses, it loses the buoyancy to push up the boundary, and the surface begins to relax.

After running the cookbook, you may modify it in a number of ways:

- Add a more complicated initial temperature field to see how that affects topography.
- Add a high-viscosity lithosphere to the top using a compositional field to tamp down on topography.
- Explore different values for the stabilization theta and the CFL number to understand the nature of when and why stabilization is necessary.
- Try a model in a different geometry, such as spherical shells.

Figure 23: *Evolution of surface topography due to a rising blob. On the left is a snapshot of the model setup. The right shows the value of the highest topography in the domain over 18 Myr of model time. The topography peaks at 165 meters after 5.2 Myr. This cookbook may be run with the* `cookbooks/free-surface.prm` *input file.*

6.2.7 Using a free surface in a model with a crust

This section was contributed by William Durkin.

This cookbook is a modification of the previous example that explores changes in the way topography develops when a highly viscous crust is added. In this cookbook, we use a material model in which the material changes from low viscosity mantle to high viscosity crust at $z = z_j =$ **jump height**, i.e., the piecewise viscosity function is defined as

$$\eta(z) = \begin{cases} \eta_U & \text{for } z > z_j, \\ \eta_L & \text{for } z \leq z_j. \end{cases}$$

where η_U and η_L are the viscosities of the upper and lower layers, respectively. This viscosity model can be implemented by creating a plugin that is a small modification of the `simpler` material model (from which it is otherwise simply copied). We call this material model "SimplerWithCrust". In particular, what is necessary is an evaluation function that looks like this:

```
template <int dim>
void
SimplerWithCrust<dim>::
evaluate(const typename Interface<dim>::MaterialModelInputs &in,
         typename Interface<dim>::MaterialModelOutputs &out ) const
{
  for (unsigned int i=0; i<in.position.size(); ++i)
    {
      const double z = in.position[i][1];

      if (z>jump_height)
        out.viscosities[i] = eta_U;
      else
        out.viscosities[i] = eta_L;

      out.densities[i] = reference_rho*(1.0-thermal_alpha*(in.temperature[i]-reference_T));
      out.thermal_expansion_coefficients[i] = thermal_alpha;
      out.specific_heat[i] = reference_specific_heat;
      out.thermal_conductivities[i] = k_value;
      out.compressibilities[i] = 0.0;
    }
}
```

Additional changes make the new parameters `Jump height`, `Lower viscosity`, and `Upper viscosity` available to the input parameter file, and corresponding variables available in the class and used in the code snippet above. The entire code can be found in `cookbooks/free-surface-with-crust/plugin/simpler-with-crust.cc`. Refer to Section 7.1 for more information about writing and running plugins.

The following changes are necessary compared to the input file from the cookbook shown in Section 6.2.6 to include a crust:

- Load the plugin implementing the new material model:

```
set Additional shared libraries            = ./plugin/libsimpler-with-crust.so
```

- Declare values for the new parameters:

```
subsection Material model
  set Model name = simpler with crust
  subsection Simpler with crust model
    set Reference density             = 3300
    set Reference specific heat       = 1250
    set Reference temperature         = 0.0
    set Thermal conductivity          = 1.0 # low thermal conductivity for a sharp blob
    set Thermal expansion coefficient = 4e-5

    # Parameters added for this cookbook:
    # The box is 200km high and has its origin set at the bottom left corner.
    # Setting the jump height to 170km creates a 30km thick crust
    set Lower viscosity               = 1e20
    set Upper viscosity               = 1e23
    set Jump height                   = 170e3
  end
end
```

  Note that the height of the interface at 170km is interpreted in the coordinate system in which the box geometry of this cookbook lives. The box has dimensions 500km $\times$ 200km, so an interface height of 170km implies a depth of 30km.

The entire script is located in `cookbooks/free-surface-with-crust/free-surface-with-crust.prm`.

Running this input file yields a crust that is 30km thick and 1000 times as viscous as the lower layer. Figure 24 shows that adding a crust to the model causes the maximum topography to both decrease and occur at a later time. Heat flows through the system primarily by advection until the temperature anomaly reaches the base of the crustal layer (approximately at the time for which Fig 24 shows the temperature profile). The crust's high viscosity reduces the temperature anomaly's velocity substantially, causing it to affect the surface topography at a later time. Just as the cookbook shown in Section 6.2.6, the topography returns to zero after some time.

6.2.8 Averaging material properties

The original motivation for the functionality discussed here, as well as the setup of the input file, were provided by Cedric Thieulot.

Geophysical models are often characterized by abrupt and large jumps in material properties, in particular in the viscosity. An example is a subducting, cold slab surrounded by the hot mantle: Here, the strong temperature-dependence of the viscosity will lead to a sudden jump in the viscosity between mantle and slab. The length scale over which this jump happens will be a few or a few tens of kilometers. Such length scales cannot be adequately resolved in three-dimensional computations with typical meshes for global computations.

Figure 24: *Adding a viscous crust to a model with surface topography. The thermal anomaly spreads horizontally as it collides with the highly viscous crust (left). The addition of a crustal layer both dampens and delays the appearance of the topographic maximum and minimum (right).*

Having large viscosity variations in models poses a variety of problems to numerical computations. First, you will find that they lead to very long compute times because our solvers and preconditioners break down. This may be acceptable if it would at least lead to accurate solution, but large viscosity gradients lead also to large pressure gradients, and this in turn leads to over- and undershots in the numerical approximation of the gradient. We will demonstrate both of these issues experimentally below.

One of the solution to such problems is the realization that one can mitigate some of the effects by averaging material properties on each cell somehow (see, for example, [SBE+08, DK08, DMGT11, Thi15, TMK14]). Before going into detail, it is important to realize that if we choose material properties not per quadrature point when doing the integrals for forming the finite element matrix, but per cell, then we will lose accuracy in the solution in those cases where the solution is smooth. More specifically, we will likely lose one or more orders of convergence. In other words, it would be a bad idea to do this averging unconditionally. On the other hand, if the solution has essentially discontinuous gradients and kinks in the velocity field, then at least at these locations we cannot expect a particularly high convergence order anyway, and the averaging will not hurt very much either. In cases where features of the solution that are due to strongly varying viscosities or other parameters, dominate, we may then as well do the averaging per cell.

To support such cases, ASPECT supports an operation where we evaluate the material model at every quadrature point, given the temperature, pressure, strain rate, and compositions at this point, and then either (i) use these values, (ii) replace the values by their arithmetic average $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$, (iii) replace the values by their harmonic average $\bar{x} = \left(\frac{1}{N}\sum_{i=1}^{N}\frac{1}{x_i}\right)^{-1}$, (iv) replace the values by their geometric average $\bar{x} = \left(\prod_{i=1}^{N}\frac{1}{x_i}\right)^{-1/N}$, or (v) replace the values by the largest value over all quadrature points on this cell. Option (vi) is to project the values from the quadrature points to a bi- (in 2d) or trilinear (in 3d) Q_1 finite element space on every cell, and then evaluate this finite element representation again at the quadrature points. Unlike the other five operations, the values we get at the quadrature points are not all the same here.

We do this operation for all quantities that the material model computes, i.e., in particular, the viscosity, the density, the compressibility, and the various thermal and thermodynamic properties. In the first 4 cases, the operation guarantees that the resulting material properties are bounded below and above by the minimum and maximum of the original data set. In the last case, the situation is a bit more complicated: The nodal values of the Q_1 projection are not necessarily bounded by the minimal or maximal original values at the quadrature points, and then neither are the output values after re-interpolation to the quadrature points. Consequently, after projection, we limit the nodal values of the projection to the minimal and maximal original values, and only then interpolate back to the quadrature points.

We demonstrate the effect of all of this with the "sinker" benchmark. This benchmark is defined by a high-viscosity, heavy sphere at the center of a two-dimensional box. This is achieved by defining a compositional

field that is one inside and zero outside the sphere, and assigning a compositional dependence to the viscosity and density. We run only a single time step for this benchmark. This is all modeled in the following input file that can also be found in cookbooks/sinker-with-averaging/sinker-with-averaging.prm:

```
set Dimension                              = 2
set Start time                             = 0
set End time                               = 0
set Output directory                       = output_sinker

set Linear solver tolerance                = 1e-7
set Pressure normalization                 = volume


subsection Geometry model
  set Model name = box
  subsection Box
    set X extent  = 1.0000
    set Y extent  = 1.0000
  end
end

subsection Model settings
  set Include adiabatic heating               = false
  set Include shear heating                   = false
  set Tangential velocity boundary indicators =
  set Zero velocity boundary indicators       = left, right, bottom, top
end

subsection Material model
  set Model name = simple

  subsection Simple model
    set Reference density             = 1
    set Viscosity                     = 1
    set Thermal expansion coefficient = 0
    set Composition viscosity prefactor = 1e6
    set Density differential for compositional field 1 = 10
  end

  set Material averaging = none
end

subsection Gravity model
  set Model name = vertical
  subsection Vertical
    set Magnitude = 1
  end
end


############### Parameters describing the temperature field
# Note: The temperature plays no role in this model

subsection Boundary temperature model
  set Model name = box
end
```

```
subsection Initial conditions
  set Model name = function
  subsection Function
    set Function expression = 0
  end
end


############### Parameters describing the compositional field
# Note: The compositional field is what drives the flow
# in this example

subsection Compositional fields
  set Number of fields = 1
end

subsection Compositional initial conditions
  set Model name = function
  subsection Function
    set Variable names      = x,y
    set Function expression = if( (sqrt((x-0.5)^2+(y-0.5)^2)>0.22) , 0 , 1 )
  end
end


############### Parameters describing the discretization

subsection Mesh refinement
  set Initial global refinement          = 6
  set Initial adaptive refinement        = 0
end



############### Parameters describing what to do with the solution

subsection Postprocess
  set List of postprocessors = visualization, velocity statistics, composition statistics
  subsection Visualization
    set Output format                 = vtu
    set Time between graphical output = 0
    set List of output variables = density, viscosity
  end
end
```

The type of averaging on each cell is chosen using this part of the input file:

```
subsection Material model
  set Material averaging = harmonic average
end
```

For the various different averaging options, and for different levels of mesh refinement, Fig. 25 shows pressure plots that illustrate the problem with oscillations of the discrete pressure. The important part of these plots is not that the solution looks discontinuous – in fact, the exact solution is discontinuous at the edge of the

circle[25] – but the spikes that go far above and below the “cliff” in the pressure along the edge of the circle. Without averaging, these spikes are obviously orders of magnitude larger than the actual jump height. The spikes do not disappear under mesh refinement nor averaging, but they become far less pronounced with averaging. The results shown in the figure do not really allow to draw conclusions as to which averaging approach is the best; a discussion of this question can also be found in [SBE+08, DK08, DMGT11, TMK14]).

A very pleasant side effect of averaging is that not only does the solution become better, but it also becomes cheaper to compute. Table 1 shows the number of outer GMRES iterations when solving the Stokes equations (1)–(2).[26] The implication of these results is that the averaging gives us a solution that not only reduces the degree of pressure over- and undershots, but is also significantly faster to compute: for example, the total run time for 8 global refinement steps is reduced from 5,250s for no averaging to 358s for harmonic averaging.

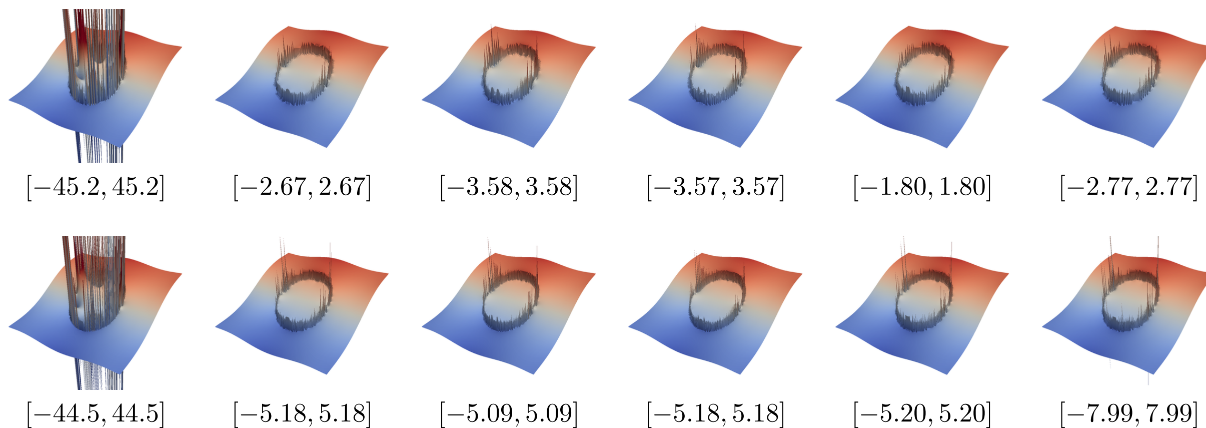


Figure 25: *Visualization of the pressure field for the “sinker” problem. Left to right: No averaging, arithmetic averaging, harmonic averaging, geometric averaging, pick largest, project to Q_1. Top: 7 global refinement steps. Bottom: 8 global refinement steps. The minimal and maximal pressure values are indicated below every picture. This range is symmetric because we enforce that the average of the pressure equals zero. The color scale is adjusted to show only values between $p = -3$ and $p = 3$.*

Such improvements carry over to more complex and realistic models. For example, in a simulation of flow under the East African Rift by Sarah Stamps, using approximately 17 million unknowns and run on 64 processors, the number of outer and inner iterations is reduced from 169 and 114,482 without averaging to 77 and 23,180 with harmonic averaging, respectively. This translates into a reduction of run-time from 145 hours to 17 hours. Assessing the accuracy of the answers is of course more complicated in such cases because we do not know the exact solution. However, the results without and with averaging do not differ in any significant way.

A final comment is in order. First, one may think that the results should be better in cases of discontinuous pressures if the numerical approximation actually allowed for discontinuous pressures. This is in fact possible: We can use a finite element in which the pressure space contains piecewise constants (see Section 5.29). To do so, one simply needs to add the following piece to the input file:

[25] This is also easy to try experimentally – use the input file from above and select 5 global and 10 adaptive refinement steps, with the refinement criteria set to `density`, then visualize the solution.

[26] The outer iterations are only part of the problem. As discussed in [KHB12], each GMRES iteration requires solving a linear system with the elliptic operator $-\nabla \cdot 2\eta\varepsilon(\cdot)$. For highly heterogeneous models, such as the one discussed in the current section, this may require a lot of Conjugate Gradient iterations. For example, for 8 global refinement steps, the 30+188 outer iterations without averaging shown in Table 1 require a total of 22,096 inner CG iterations for the elliptic block (and a total of 837 for the aproximate Schur complement). Using harmonic averaging, the 30+26 outer iterations require only 1258 iterations on the elliptic block (and 84 on the Schur complement). In other words, the number of inner iterations per outer iteration (taking into account the split into “cheap” and “expensive” outer iterations, see [KHB12]) is reduced from 117 to 47 for the elliptic block and from 3.8 to 1.5 for the Schur complement.

# of global refinement steps	no averaging	arithmetic averaging	harmonic averaging	geometric averaging	pick largest	project to Q_1
4	30+64	30+13	30+10	30+12	30+13	30+15
5	30+87	30+14	30+13	30+14	30+14	30+16
6	30+171	30+14	30+15	30+14	30+15	30+17
7	30+143	30+27	30+28	30+26	30+26	30+28
8	30+188	30+27	30+26	30+27	30+28	30+28

Table 1: *Number of outer GMRES iterations to solve the Stokes equations for various numbers of global mesh refinement steps and for different material averaging operations. The GMRES solver first tries to run 30 iterations with a cheaper preconditioner before switching to a more expensive preconditioner (see Section 5.2).*

```
subsection Discretization
  set Use locally conservative discretization = true
end
```

Disappointingly, however, this makes no real difference: the pressure oscillations are no better (maybe even worse) than for the standard Stokes element we use, as shown in Fig. 26 and Table 2. Furthermore, as shown in Table 3, the iteration numbers are also largely unaffected if any kind of averaging is used – though they are far worse using the locally conservative discretization if no averaging has been selected. On the positive side, the visualization of the discontinuous pressure finite element solution makes it much easier to see that the true pressure is in fact discontinuous along the edge of the circle.

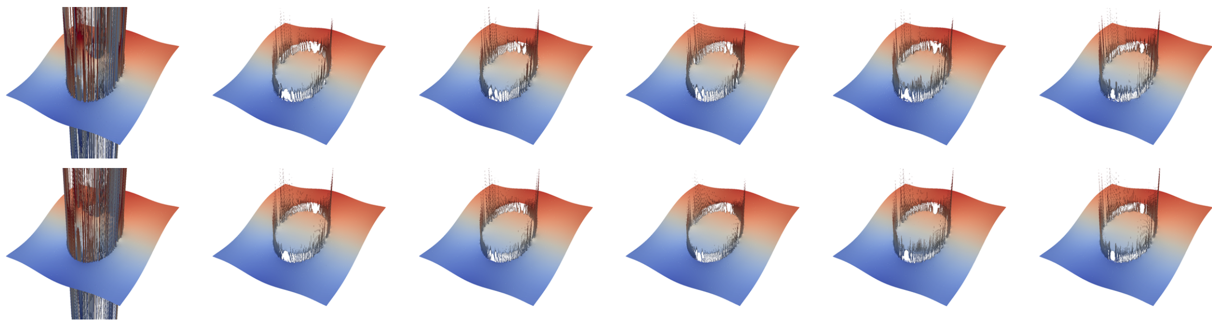


Figure 26: *Visualization of the pressure field for the "sinker" problem. Like Fig. 25 but using the locally conservative, enriched Stokes element. Pressure values are shown in Table 2.*

# of global refinement steps	no averaging	arithmetic averaging	harmonic averaging	geometric averaging	pick largest	project to Q_1
4	66.32	2.66	2.893	1.869	3.412	3.073
5	81.06	3.537	4.131	3.997	3.885	3.991
6	75.98	4.596	4.184	4.618	4.568	5.093
7	84.36	4.677	5.286	4.362	4.635	5.145
8	83.96	5.701	5.664	4.686	5.524	6.42

Table 2: *Maximal pressure values for the "sinker" benchmark, using the locally conservative, enriched Stokes element. The corresponding pressure solutions are shown in Fig. 26.*

# of global refinement steps	no averaging	arithmetic averaging	harmonic averaging	geometric averaging	pick largest	project to Q_1
4	30+376	30+16	30+12	30+14	30+14	30+17
5	30+484	30+16	30+14	30+14	30+14	30+16
6	30+583	30+16	30+17	30+14	30+17	30+17
7	30+1319	30+27	30+28	30+26	30+28	30+28
8	30+1507	30+28	30+27	30+28	30+28	30+29

Table 3: *Like Table 1, but using the locally conservative, enriched Stokes element.*

6.2.9 Prescribed internal velocity constraints

This section was contributed by Jonathan Perry-Houts

In cases where it is desirable to investigate the behavior of one part of the model domain but the controlling physics of another part is difficult to capture, such as corner flow in subduction zones, it may be useful to force the desired behavior in some parts of the model domain and solve for the resulting flow everywhere else. This is possible through the use of ASPECT's "signal" mechanism, as documented in Section 7.4.

Internally, ASPECT adds "constraints" to the finite element system for boundary conditions and hanging nodes. These are places in the finite element system where certain solution variables are required to match some prescribed value. Although it is somewhat mathematically inadmissible to prescribe constraints on nodes inside the model domain, Ω, it is nevertheless possible so long as the prescribed velocity field fits in to the finite element's solution space, and satisfies the other constraints (i.e., is divergence free).

Using ASPECT's signals mechanism, we write a shared library which provides a "slot" that listens for the signal which is triggered after the regular model constraints are set, but before they are "distributed."

As an example of this functionality, below is a plugin which allows the user to prescribe internal velocities with functions in a parameter file:

```
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/parsed_function.h>
#include <deal.II/fe/fe_values.h>
#include <aspect/global.h>
#include <aspect/simulator_signals.h>

namespace aspect
{
  using namespace dealii;

  // Global variables (to be set by parameters)
  bool prescribe_internal_velocities;

  // Because we do not initially know what dimension we're in, we need
  // function parser objects for both 2d and 3d.
  Functions::ParsedFunction<2> prescribed_velocity_indicator_function_2d (2);
  Functions::ParsedFunction<3> prescribed_velocity_indicator_function_3d (3);
  Functions::ParsedFunction<2> prescribed_velocity_function_2d (2);
  Functions::ParsedFunction<3> prescribed_velocity_function_3d (3);

  /**
   * Declare additional parameters.
   */
  void declare_parameters(const unsigned int dim,
                          ParameterHandler &prm)
  {
```

```
  prm.declare_entry ("Prescribe␣internal␣velocities", "false",
                     Patterns::Bool (),
                     "Whether␣or␣not␣to␣use␣any␣prescribed␣internal␣velocities.␣"
                     "Locations␣in␣which␣to␣prescribe␣velocities␣are␣defined␣"
                     "in␣section␣‘‘Prescribed␣velocities/Indicator␣function’’␣"
                     "and␣the␣velocities␣are␣defined␣in␣section␣‘‘Prescribed␣"
                     "velocities/Velocity␣function’’.␣Indicators␣are␣evaluated␣"
                     "at␣the␣center␣of␣each␣cell,␣and␣all␣DOFs␣associated␣with␣"
                     "the␣specified␣velocity␣component␣at␣the␣indicated␣cells␣"
                     "are␣constrained."
                    );

  prm.enter_subsection ("Prescribed␣velocities");
  {
    prm.enter_subsection ("Indicator␣function");
    {
      if (dim == 2)
        Functions::ParsedFunction<2>::declare_parameters (prm, 2);
      else
        Functions::ParsedFunction<3>::declare_parameters (prm, 3);
    }
    prm.leave_subsection ();

    prm.enter_subsection ("Velocity␣function");
    {
      if (dim == 2)
        Functions::ParsedFunction<2>::declare_parameters (prm, 2);
      else
        Functions::ParsedFunction<3>::declare_parameters (prm, 3);
    }
    prm.leave_subsection ();
  }
  prm.leave_subsection ();
}

template <int dim>
void parse_parameters(const Parameters<dim> &,
                      ParameterHandler &prm)
{
  prescribe_internal_velocities = prm.get_bool ("Prescribe␣internal␣velocities");
  prm.enter_subsection ("Prescribed␣velocities");
  {
    prm.enter_subsection("Indicator␣function");
    {
      try
        {
          if (dim == 2)
            prescribed_velocity_indicator_function_2d.parse_parameters (prm);
          else
            prescribed_velocity_indicator_function_3d.parse_parameters (prm);
        }
      catch (...)
        {
          std::cerr << "ERROR:␣FunctionParser␣failed␣to␣parse\n"
                    << "\t’Prescribed␣velocities.Indicator␣function’\n"
                    << "with␣expression\n"
```

```
                    << "\t'" << prm.get("Function␣expression") << "'";
          throw;
        }
    }
    prm.leave_subsection();

    prm.enter_subsection("Velocity␣function");
    {
      try
        {
          if (dim == 2)
            prescribed_velocity_function_2d.parse_parameters (prm);
          else
            prescribed_velocity_function_3d.parse_parameters (prm);
        }
      catch (...)
        {
          std::cerr << "ERROR:␣FunctionParser␣failed␣to␣parse\n"
                    << "\t'Prescribed␣velocities.Velocity␣function'\n"
                    << "with␣expression\n"
                    << "\t'" << prm.get("Function␣expression") << "'";
          throw;
        }
    }
    prm.leave_subsection();
  }
  prm.leave_subsection ();
}

/**
 * This function is called by a signal which is triggered after the other constraints
 * have been calculated. This enables us to define additional constraints in the mass
 * matrix on any arbitrary degree of freedom in the model space.
 */
template <int dim>
void constrain_internal_velocities (const SimulatorAccess<dim> &simulator_access,
                                    ConstraintMatrix &current_constraints)
{
  if (prescribe_internal_velocities)
    {
      Quadrature<dim> quadrature (simulator_access.get_fe().get_unit_support_points());
      FEValues<dim> fe_values (simulator_access.get_fe(), quadrature, update_q_points);
      typename DoFHandler<dim>::active_cell_iterator cell;

      // Loop over all cells
      for (cell = simulator_access.get_dof_handler().begin_active();
           cell != simulator_access.get_dof_handler().end();
           ++cell)
        if (! cell->is_artificial())
          {
            fe_values.reinit (cell);
            std::vector<unsigned int> local_dof_indices(simulator_access.get_fe().dofs_per_cell);
            cell->get_dof_indices (local_dof_indices);

            for (unsigned int q=0; q<quadrature.size(); q++)
              // If it's okay to constrain this DOF
```

```
              if (current_constraints.can_store_line(local_dof_indices[q]) &&
                  !current_constraints.is_constrained(local_dof_indices[q]))
                {
                  // Get the velocity component index
                  const unsigned int c_idx =
                    simulator_access.get_fe().system_to_component_index(q).first;

                  // If we're on one of the velocity DOFs
                  if (c_idx >= simulator_access.introspection().component_indices.velocities[0] &&
                      c_idx <= simulator_access.introspection().component_indices.velocities[dim-1])
                    {
                      // Which velocity component is this DOF associated with?
                      const unsigned int component_direction = c_idx
                                                               - simulator_access.introspection().component_indic

                      // we get time passed as seconds (always) but may want
                      // to reinterpret it in years
                      double time = simulator_access.get_time();
                      if (simulator_access.convert_output_to_years())
                        time /= year_in_seconds;

                      prescribed_velocity_indicator_function_2d.set_time (time);
                      prescribed_velocity_indicator_function_3d.set_time (time);
                      prescribed_velocity_function_2d.set_time (time);
                      prescribed_velocity_function_3d.set_time (time);

                      const Point<dim> p = fe_values.quadrature_point(q);

                      // Because we defined and parsed our parameter file differently for 2d and 3d
                      // we need to be sure to query the correct object for function values. The
                      // function parser objects expect points of a certain dimension, but Point p
                      // will be compiled for both 2d and 3d, so we need some careful casts to make
                      // this compile. Casting Point objects between 2d and 3d is somewhat meaningless
                      // but the offending casts will never actually be executed because they are
                      // protected by the if/else statements.
                      double indicator, u_i;
                      if (dim == 2)
                        {
                          indicator = prescribed_velocity_indicator_function_2d.value
                                      (reinterpret_cast<const Point<2>&>(p), component_direction);
                          u_i = prescribed_velocity_function_2d.value
                                (reinterpret_cast<const Point<2>&>(p), component_direction);
                        }
                      else
                        {
                          indicator = prescribed_velocity_indicator_function_3d.value
                                      (reinterpret_cast<const Point<3>&>(p), component_direction);
                          u_i = prescribed_velocity_function_3d.value
                                (reinterpret_cast<const Point<3>&>(p), component_direction);
                        }

                      if (indicator > 0.5)
                        {
                          // Add a constraint of the form dof[q] = u_i
                          // to the list of constraints.
                          current_constraints.add_line (local_dof_indices[q]);
```

```
                            current_constraints.set_inhomogeneity (local_dof_indices[q], u_i);
                          }
                      }
                  }
            }
      }
  }

  // Connect declare_parameters and parse_parameters to appropriate signals.
  void parameter_connector ()
  {
    SimulatorSignals<2>::declare_additional_parameters.connect (&declare_parameters);
    SimulatorSignals<3>::declare_additional_parameters.connect (&declare_parameters);

    SimulatorSignals<2>::parse_additional_parameters.connect (&parse_parameters<2>);
    SimulatorSignals<3>::parse_additional_parameters.connect (&parse_parameters<3>);
  }

  // Connect constraints function to correct signal.
  template <int dim>
  void signal_connector (SimulatorSignals<dim> &signals)
  {
    signals.post_constraints_creation.connect (&constrain_internal_velocities<dim>);
  }

  // Tell Aspect to send signals to the connector functions
  ASPECT_REGISTER_SIGNALS_PARAMETER_CONNECTOR(parameter_connector)
  ASPECT_REGISTER_SIGNALS_CONNECTOR(signal_connector<2>, signal_connector<3>)
}
```

The above plugin can be compiled with `cmake . && make` in the cookbooks/prescribed_velocity directory. It can be loaded in a parameter file as an "Additional shared library." By setting parameters like those shown below, it is possible to produce many interesting flow fields such as the ones visualized in (Figure 27).

```
# Load the signal library.
set Additional shared libraries = ./libprescribed_velocity.so

## Turn prescribed velocities on
set Prescribe internal velocities = true

subsection Prescribed velocities
  subsection Indicator function
    set Variable names = x,y,t
    # Return where to prescribe u_x; u_y; u_z
    # (last one only used if dimension = 3)
    # 1 if velocity should be prescribed, 0 otherwise
    set Function expression = if((x-.5)^2+(y-.5)^2<.125,1,0); \
                              if((x-.5)^2+(y-.5)^2<.125,1,0)
  end
  subsection Velocity function
    set Variable names = x,y,t
    # Return u_x; u_y; u_z (u_z only used if in 3d)
    set Function expression = 1;-1
  end
end
```

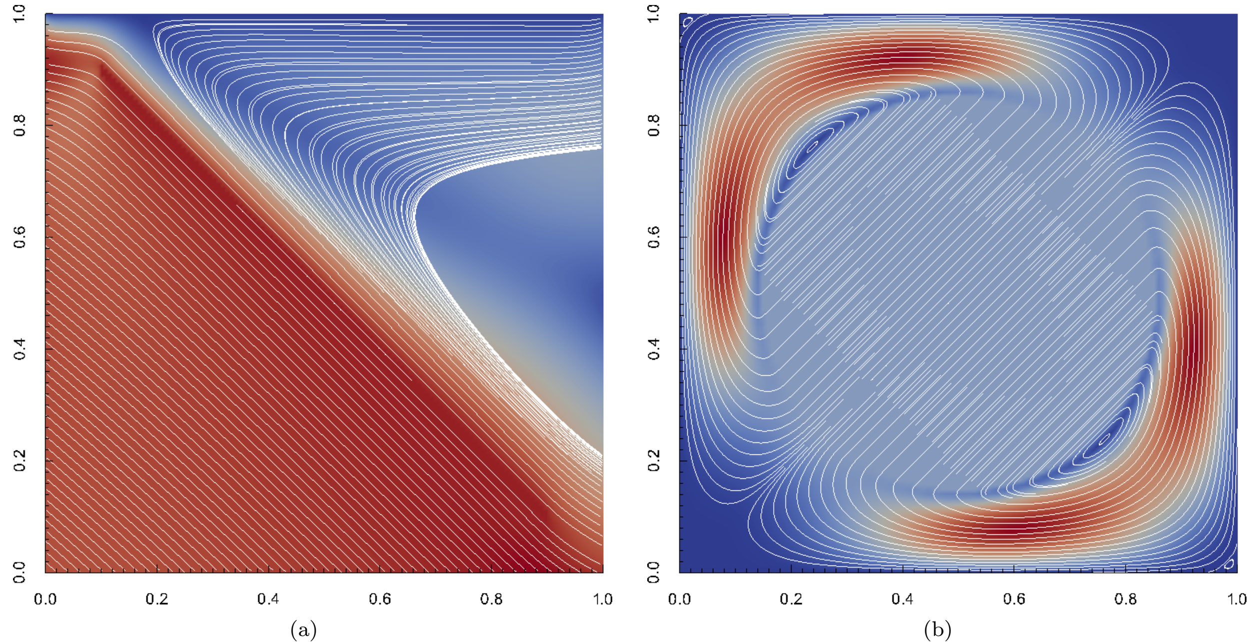


Figure 27: Examples of flows with prescribed internal velocities, as described in Section 6.2.9.

6.2.10 Artificial viscosity smoothing

This section was contributed by Ryan Grove

Standard finite element discretizations of advection-diffusion equations introduce unphysical oscillations around steep gradients. Therefore, stabilization must be added to the discrete formulation to obtain correct solutions. In ASPECT, we use the Entropy Viscosity scheme developed by Guermond et al. in the paper [Jea11]. In this scheme, an artificial viscosity is calculated on every cell and used to try to combat these oscillations that cause unwanted overshoot and undershoot. More information about how ASPECT does this is located at `https://dealii.org/developer/doxygen/deal.II/step_31.html`.

Instead of just looking at an individual cell's artificial viscosity, improvements in the minimizing of the oscillations can be made by smoothing. Smoothing is the act of finding the maximum artificial viscosity taken over a cell T and the neighboring cells across the faces of T, i.e.,

$$\bar{v_h}(T) = \max_{K \in N(T)} v_h(K)$$

where $N(T)$ is the set containing T and the neighbors across the faces of T.

This feature can be turned on by setting the Use artificial viscosity smoothing flag inside the Stabilization subsection inside the Discretization subsection in your parameter file.

To show how this can be used in practice, let us consider the simple convection in a quarter of a 2d annulus cookbook in Section 6.3.1, a radial compositional field was added to help show the advantages of using the artificial viscosity smoothing feature.

By applying the following changes shown below to the parameters of the already existing file

`cookbooks/shell_simple_2d.prm,`

```
subsection Discretization
  set Temperature polynomial degree = 2
```

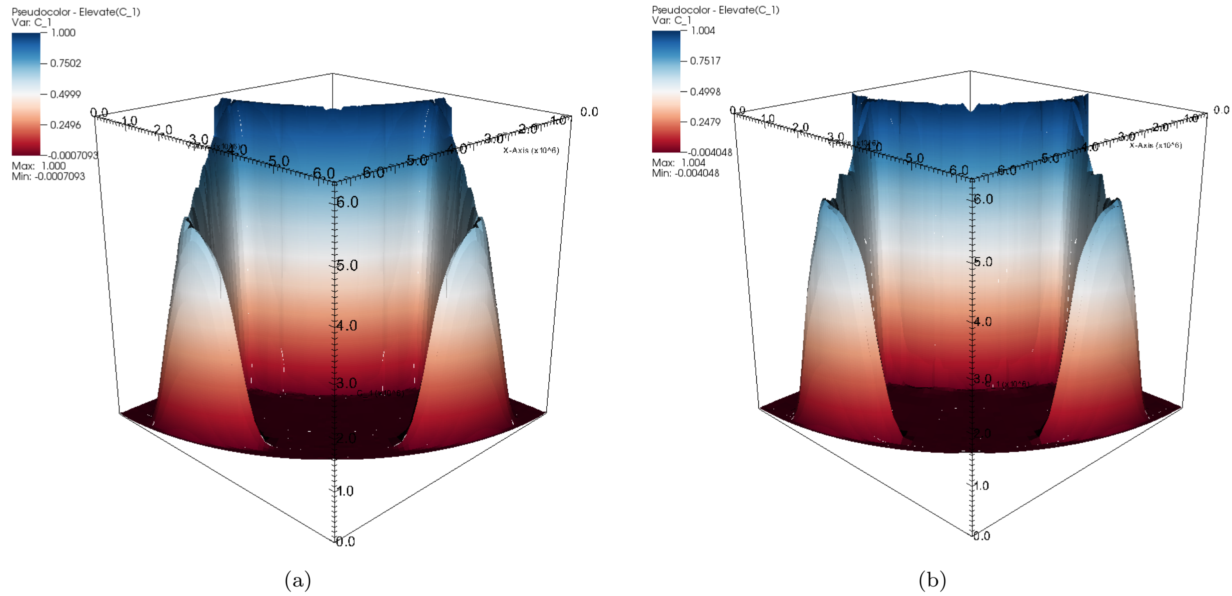


Figure 28: Artificial viscosity smoothing: Example of the output of two similar runs. The run on the left has the artificial viscosity smoothing turned on and the run on the right does not, as described in Section 6.2.10.

```
  subsection Stabilization parameters
    set Use artificial viscosity smoothing = true
  end
end

subsection Compositional fields
  set Number of fields = 1
end

subsection Compositional initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,y
    set Function expression = if(sqrt(x*x+y*y)<4000000,1,0)
  end
end
```

it is possible to produce pictures of the simple convection in a quarter of a 2d annulus such as the ones visualized in Figure 28.

6.2.11 Tracking finite strain

This section was contributed by Juliane Dannberg

Notation: Here, we denote the strain rate tensor as $\varepsilon(\mathbf{u})$*, where* $\varepsilon(\mathbf{u})_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$ *with the velocity* $\mathbf{u}$*, and the strain tensor as* e*, where* $e_{ij} = \frac{1}{2}\left(\frac{\partial d_i}{\partial x_j} + \frac{\partial d_j}{\partial x_i}\right)$ *with the displacement* $\mathbf{d}$*.*

In many geophysical settings, material properties, and in particular the rheology, do not only depend on the current temperature, pressure and strain rate, but also on the history of the system. This can

be incorporated in ASPECT models by tracking history variables through compositional fields. In this cookbook, we will show how to do this by tracking the strain that accumulates over time at every (Lagrangian) point in the model.

Here, we use a material model plugin that defines the compositional fields as the components of the strain tensor e_{ij}, and modifies the right-hand side of the corresponding advection equations to accumulate strain over time. This is done by adjusting the `out.reaction_terms` variable:

```
for (unsigned int q=0; q < in.position.size(); ++q)
  {
    // rotation tensor =
    // asymmetric part of the displacement in this time step (= velocity gradients tensor * time step)
    // + unit tensor
    const Tensor<2,dim> rotation = (velocity_gradients[q] - symmetrize(velocity_gradients[q]))
                                   * this->get_timestep()
                                   + unit_symmetric_tensor<dim>();

    Tensor<2,dim> accumulated_strain;
    for (unsigned int i=0; i<Tensor<2,dim>::n_independent_components; ++i)
      accumulated_strain[Tensor<2,dim>::unrolled_to_component_indices(i)] = in.composition[q][i];

    // the new strain is the rotated old strain plus the strain of the current time step
    const Tensor<2,dim> rotated_strain = rotation * accumulated_strain * transpose(rotation)
                                         + in.strain_rate[q] * this->get_timestep();

    for (unsigned int c=0; c<Tensor<2,dim>::n_independent_components; ++c)
      {
        out.reaction_terms[q][c] = - in.composition[q][c]
                                   + rotated_strain[Tensor<2,dim>::unrolled_to_component_indices(c)];
      }
  }
```

Let us denote the accumulated strain at time step n as e^n. We can express it as the sum of the strain e^{n-1} at the previous time step, rotated by the rotational component of the velocity field, plus the strain increment $\varepsilon(\mathbf{u}^n)\Delta t^n$ accumulated over the current time step. Hence, the right-hand side term of the advection equation for the accumulated strain consists of two parts: The first one, $Re^{n-1}R^T$, rotates e^{n-1}, the accumulated strain from all the previous time steps; and the second part adds the strain of the current time step.

The full plugin can be found in `cookbooks/finite_strain/finite_strain.cc` and can be compiled with `cmake . && make` in the `cookbooks/finite_strain` directory. It can be loaded in a parameter file as an "Additional shared library", and selected as material model. As it is derived from the "simple" material model, all input parameters for the material properties are read in from the subsection `Simple model`.

```
set Additional shared libraries            = ./libfinite_strain.so

subsection Material model
  set Model name = finite strain

  subsection Simple model
    set Thermal conductivity = 4.7
    set Reference density = 3400
    set Thermal expansion coefficient = 2e-5
    set Viscosity = 5e21
    set Thermal viscosity exponent = 7
    set Reference temperature = 1600
  end
end
```

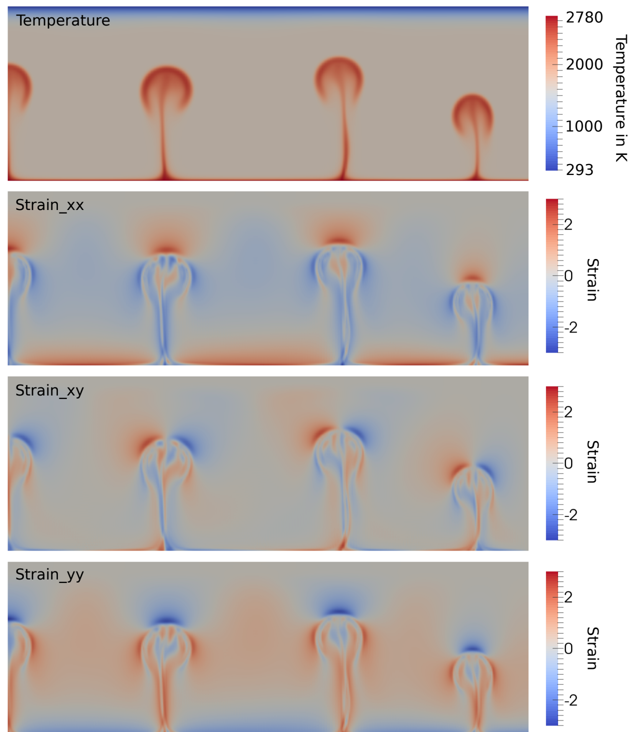


Figure 29: Accumulated finite strain in an example convection model, as described in Section 6.2.11.

We will demonstrate its use at the example of a 2D Cartesian convection model (Figure 29): Heating from the bottom leads to the the ascent of plumes from the bottom boundary layer, and the associated deformation is visible in the components of the accumulated finite strain. Material moves to the sides at the top of the plume head, so that it is shortened in vertical direction (blue areas in the yy component) and stretched in horizontal direction (red areas in the xx component). The sides of the plume head show the opposite effect. Shear occurs mostly at the edges of the plume head and in the plume tail (blue and red areas in the xy component).

The example used here shows how history variables can be integrated up over the model evolution. While we do not use these variables actively in the computation so far (in our example, there is no influence of the accumulated strain on the rheology or any other material property), it would be trivial to extend this material model in a way that material properties depend on the integrated strain: Because the values of the compositional fields are part of what the material model gets as inputs, they can easily be used for computing material model outputs such as the viscosity.

6.3 Geophysical setups

Having gone through the ways in which one can set up problems in rectangular geometries, let us now move on to situations that are directed more towards the kinds of things we want to use ASPECT for: the simulation of convection in the rocky mantles of planets or other celestial bodies.

To this end, we need to go through the list of issues that have to be described and that were outlined in Section 6.1, and address them one by one:

- *What internal forces act on the medium (the equation)?* This may in fact be the most difficult to answer part of it all. The real material in Earth's mantle is certainly no Newtonian fluid where the stress is a linear function of the strain with a proportionality constant (the viscosity) η that only depends on

the temperature. Rather, the real viscosity almost surely also depends on the pressure and the strain rate. Because the issue is complicated and the exact material model not entirely clear, for the next few subsections we will therefore ignore the issue and start with just using the "simple" material model where the viscosity is constant and most other coefficients depend at most on the temperature.

- *What external forces do we have (the right hand side)* There are of course other issues: for example, should the model include terms that describe shear heating? Should it be compressible? Adiabatic heating due to compression? Most of the terms that pertain to these questions appear on the right hand sides of the equations, though some (such as the compressibility) also affect the differential operators on the left. Either way, for the moment, let us just go with the simplest models and come back to the more advanced questions in later examples.

  One right hand side that will certainly be there is that due to gravitational acceleration. To first order, within the mantle gravity points radially inward and has a roughly constant magnitude. In reality, of course, the strength and direction of gravity depends on the distribution and density of materials in Earth – and, consequently, on the solution of the model at every time step. We will discuss some of the associated issues in the examples below.

- *What is the domain (geometry)?* This question is easier to answer. To first order, the domains we want to simulate are spherical shells, and to second order ellipsoid shells that can be obtained by considering the isopotential surface of the gravity field of a homogenous, rotating fluid. A more accurate description is of course the geoid for which several parameterizations are available. A complication arises if we ask whether we want to include the mostly rigid crust in the domain and simply assume that it is part of the convecting mantle, albeit a rather viscous part due to its low temperature and the low pressure there, or whether we want to truncate the computation at the asthenosphere.

- *What happens at the boundary for each variable involved (boundary conditions)?* The mantle has two boundaries: at the bottom where it contacts the outer core and at the top where it either touches the air or, depending on the outcome of the discussion of the previous question, where it contacts the lithospheric crust. At the bottom, a very good approximation of what is happening is certainly to assume that the velocity field is tangential (i.e., horizontal) and without friction forces due to the very low viscosity of the liquid metal in the outer core. Similarly, we can assume that the outer core is well mixed and at a constant temperature. At the top boundary, the situation is slightly more complex because in reality the boundary is not fixed but also allows vertical movement. If we ignore this, we can assume free tangential flow at the surface or, if we want, prescribe the tangential velocity as inferred from plate motion models. ASPECT has a plugin that allows to query this kind of information from the `GPlates` program.

- *How did it look at the beginning (initial conditions)?* This is of course a trick question. Convection in the mantle of earth-like planets did not start with a concrete initial temperature distribution when the mantle was already fully formed. Rather, convection already happened when primordial material was still separating into mantle and core. As a consequence, for models that only simulate convection using mantle-like geometries and materials, no physically reasonable initial conditions are possible that date back to the beginning of Earth. On the other hand, recall that we only need initial conditions for the temperature (and, if necessary, compositional fields). Thus, if we have a temperature profile at a given time, for example one inferred from seismic data at the current time, then we can use these as the starting point of a simulation.

This discussion shows that there are in fact many pieces with which one can play and for which the answers are in fact not always clear. We will address some of them in the cookbooks below. Recall in the descriptions we use in the input files that ASPECT uses physical units, rather than non-dimensionalizing everything. The advantage, of course, is that we can immediately compare outputs with actual measurements. The disadvantage is that we need to work a bit when asked for, say, the Rayleigh number of a simulation.

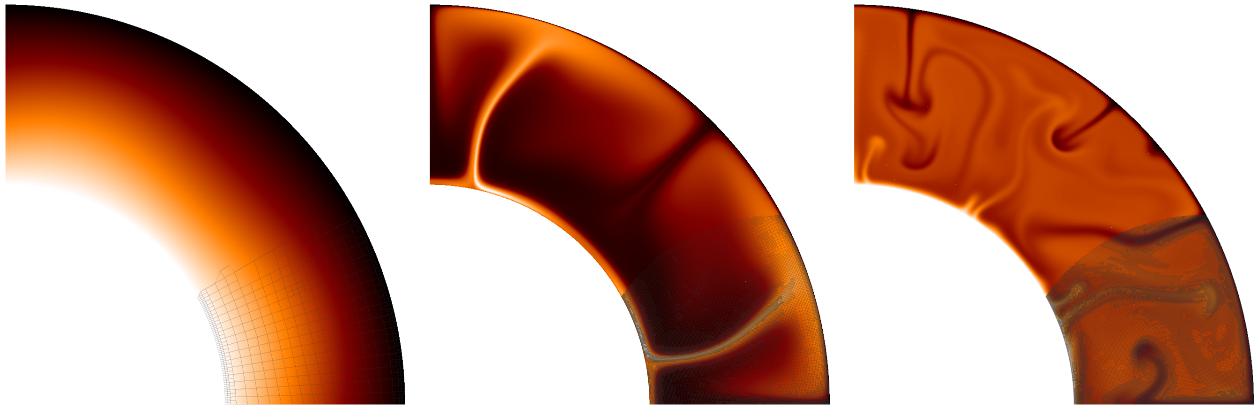


Figure 30: *Simple convection in a quarter of an annulus: Snapshots of the temperature field at times* $t = 0$, $t = 1.2 \cdot 10^7$ *years (time step 2135), and* $t = 10^9$ *years (time step 25,662). The bottom right part of each figure shows an overlay of the mesh used during that time step.*

6.3.1 Simple convection in a quarter of a 2d annulus

Let us start this sequence of cookbooks using a simpler situation: convection in a quarter of a 2d shell. We choose this setup because 2d domains allow for much faster computations (in turn allowing for more experimentation) and because using a quarter of a shell avoids a pitfall with boundary conditions we will discuss in the next section. Because it's simpler to explain what we want to describe in pictures than in words, Fig. 30 shows the domain and the temperature field at a few time steps. In addition, you can find a movie of how the temperature evolves over this time period at http://www.youtube.com/watch?v=d4AS1FmdarU.[27]

Let us just start by showing the input file (which you can find in cookbooks/shell_simple_2d.prm):

```
set Dimension                              = 2
set Use years in output instead of seconds = true
set End time                               = 1.5e9
set Output directory                       = output


subsection Material model
  set Model name = simple

  subsection Simple model
    set Thermal expansion coefficient = 4e-5
    set Viscosity                     = 1e22
  end
end


subsection Geometry model
  set Model name = spherical shell

  subsection Spherical shell
    set Inner radius  = 3481000
    set Outer radius  = 6336000
    set Opening angle = 90
  end
```

[27]In YouTube, click on the gear symbol at the bottom right of the player window to select the highest resolution to see all the details of this video.

```
end


subsection Model settings
  set Zero velocity boundary indicators       = inner
  set Tangential velocity boundary indicators = outer, left, right
  set Prescribed velocity boundary indicators =

  set Fixed temperature boundary indicators   = inner, outer

  set Include shear heating                   = true
end


subsection Boundary temperature model
  set Model name = spherical constant
  subsection Spherical constant
    set Inner temperature = 4273
    set Outer temperature = 973
  end
end


subsection Initial conditions
  set Model name = spherical hexagonal perturbation
end


subsection Gravity model
  set Model name = radial earth-like
end


subsection Mesh refinement
  set Initial global refinement          = 5
  set Initial adaptive refinement        = 4
  set Strategy                           = temperature
  set Time steps between mesh refinement = 15
end


subsection Postprocess
  set List of postprocessors = visualization, velocity statistics, temperature statistics, ...
                               ... heat flux statistics, depth average

  subsection Visualization
    set Output format                 = vtu
    set Time between graphical output = 1e6
    set Number of grouped files       = 0
  end

  subsection Depth average
    set Time between graphical output = 1e6
  end
end
```

In the following, let us pick apart this input file:

1. Lines 1–4 are just global parameters. Since we are interested in geophysically realistic simulations, we will use material parameters that lead to flows so slow that we need to measure time in years, and we will set the end time to 1.5 billion years – enough to see a significant amount of motion.

2. The next block (lines 7–14) describes the material that is convecting (for historical reasons, the remainder of the parameters that describe the equations is in a different section, see the fourth point below). We choose the simplest material model ASPECT has to offer where the viscosity is constant (here, we set it to $\eta = 10^{22}$Pa s) and so are all other parameters except for the density which we choose to be $\rho(T) = \rho_0(1 - \beta(T - T_{\text{ref}}))$ with $\rho_0 = 3300\text{kg m}^{-3}$, $\beta = 4 \cdot 10^{-5}\text{K}^{-1}$ and $T_{\text{ref}} = 293\text{K}$. The remaining material parameters remain at their default values and you can find their values described in the documentation of the `simple` material model in Sections 5.66 and 5.79.

3. Lines 17–25 then describe the geometry. In this simple case, we will take a quarter of a 2d shell (recall that the dimension had previously been set as a global parameter) with inner and outer radii matching those of a spherical approximation of Earth.

4. The second part of the model description and boundary values follows in lines 28–45. There, we describe that we want a model where equation (3) contains the shear heating term $2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u})$ (noting that the default is to use an incompressible model for which the term $\frac{1}{3}(\nabla \cdot \mathbf{u})\mathbf{1}$ in the shear heating contribution is zero).

   The boundary conditions require us to look up how the geometry model we chose (the `spherical shell` model) assigns boundary indicators to the four sides of the domain. This is described in Section 5.32 where the model announces that boundary indicator zero is the inner boundary of the domain, boundary indicator one is the outer boundary, and the left and right boundaries for a 2d model with opening angle of 90 degrees as chosen here get boundary indicators 2 and 3, respectively. In other words, the settings in the input file correspond to a zero velocity at the inner boundary and tangential flow at all other boundaries. We know that this is not realistic at the bottom, but for now there are of course many other parts of the model that are not realistic either and that we will have to address in subsequent cookbooks. Furthermore, the temperature is fixed at the inner and outer boundaries (with the left and right boundaries then chosen so that no heat flows across them, emulating symmetry boundary conditions) and, further down, set to values of 700 and 4000 degrees Celsius – roughly realistic for the bottom of the crust and the core-mantle boundary.

5. The description of what we want to model is complete by specifying that the initial temperature is a perturbation with hexagonal symmetry from a linear interpolation between inner and outer temperatures (see Section 5.51), and what kind of gravity model we want to choose (one reminiscent of the one inside the Earth mantle, see Section 5.39).

6. The remainder of the input file consists of a description of how to choose the initial mesh and how to adapt it (lines 58–63) and what to do at the end of each time step with the solution that ASPECT computes for us (lines 66–79). Here, we ask for a variety of statistical quantities and for graphical output in VTU format every million years.

Note: Having described everything to ASPECT, you may want to view the video linked to above again and compare what you see with what you expect. In fact, this is what one should always do having just run a model: compare it with expectations to make sure that we have not overlooked anything when setting up the model or that the code has produced something that doesn't match what we thought we should get. Any such mismatch between expectation and observed result is typically a learning opportunity: it either points to a bug in our input file, or it provides us with insight about an aspect of reality that we had not foreseen. Either way, accepting results uncritically is, more often than not, a way to scientifically invalid results.

The model we have chosen has a number of inadequacies that make it not very realistic (some of those happened more as an accident while playing with the input file and weren't a purposeful experiment, but we left them in because they make for good examples to discuss below). Let us discuss these issues in the following.

Dimension. This is a cheap shot but it is nevertheless true that the world is three-dimensional whereas the simulation here is 2d. We will address this in the next section.

Incompressibility, adiabaticity and the initial conditions. This one requires a bit more discussion. In the model selected above, we have chosen a model that is incompressible in the sense that the density does not depend on the pressure and only very slightly depends on the temperature (the Boussinesq approximation). In such models, material that rises up does not cool down due to expansion resulting from the pressure dropping, and material that is transported down does not adiabatically heat up. Consequently, the adiabatic temperature profile would be constant with depth, and a well-mixed model with hot inner and cold outer boundary would have a constant temperature with thin boundary layers at the bottom and top of the mantle. In contrast to this, our initial temperature field was a perturbation of a linear temperature profile.

There are multiple implications of this. First, the temperature difference between outer and inner boundary of 3300 K we have chosen in the input file is much too large. The temperature difference is what drives the convection, but what really counts is of course the difference *in addition* to the temperature increase a volume of material would experience if it were to be transported adiabatically from the surface to the core-mantle boundary. This difference is much smaller than 3300 K in reality, and we can expect convection to be significantly less vigorous than in the simulation here. Indeed, using the values in the input file shown above, we can compute the Rayleigh number for the current case to be[28]

$$\mathrm{Ra} = \frac{g\ \beta\ \Delta T\ \rho\ L^3}{\alpha\eta} \approx \frac{10\mathrm{m\,s^{-2}}\ 4\cdot 10^{-5}\mathrm{K^{-1}}\ 3300\mathrm{K}\ 3300\mathrm{kg\,m^{-2}}\ (2.86\cdot 10^6\mathrm{m})^3}{10^{22}\mathrm{kg\,m^{-1}\,s^{-1}}}.$$

Second, the initial temperature profile we chose is not realistic – in fact, it is a completely instable one: there is hot material underlying cold one, and this is not just the result of boundary layers. Consequently, what happens in the simulation is that we first overturn the entire temperature field with the hot material in the lower half of the domain swapping places with the colder material in the top, to achieve a stable layering except for the boundary layers. After this, hot blobs rise from the bottom boundary layer into the cold layer at the bottom of the mantle, and cold blobs sink from the top, but their motion is impeded about half-way through the mantle once they reach material that has roughly the same temperature as the plume material. This impedes convection until we reach a state where these plumes have sufficiently mixed the mantle to achieve a roughly constant temperature profile.

This effect is visible in the movie linked to above where convection does not penetrate the entire depth of the mantle for the first 20 seconds (corresponding to roughly the first 800 million years). We can also see this effect by plotting the root mean square velocity, see the left panel of Fig. 31. There, we can see how the average velocity picks up once the stable layering of material that resulted from the initial overturning has been mixed sufficiently to allow plumes to rise or sink through the entire depth of the mantle.

The right panel of Fig. 31 shows a different way of visualizing this, using the average temperature at various depths of the model (this is what the `depth average` postprocessor computes). The figure shows how the initially linear unstable layering almost immediately reverts completely, and then slowly equilibrates towards a temperature profile that is constant throughout the mantle (which in the incompressible model chosen here equates to an adiabatic layering) except for the boundary layers at the inner and outer boundaries. (The end points of these temperature profiles do not exactly match the boundary values specified in the input file because we average temperatures over shells of finite width.)

A conclusion of this discussion is that if we want to evaluate the statistical properties of the flow field, e.g., the number of plumes, average velocities or maximal velocities, then we need to restrict our efforts to

[28] Note that the density in 2d has units $\mathrm{kg\,m^{-2}}$

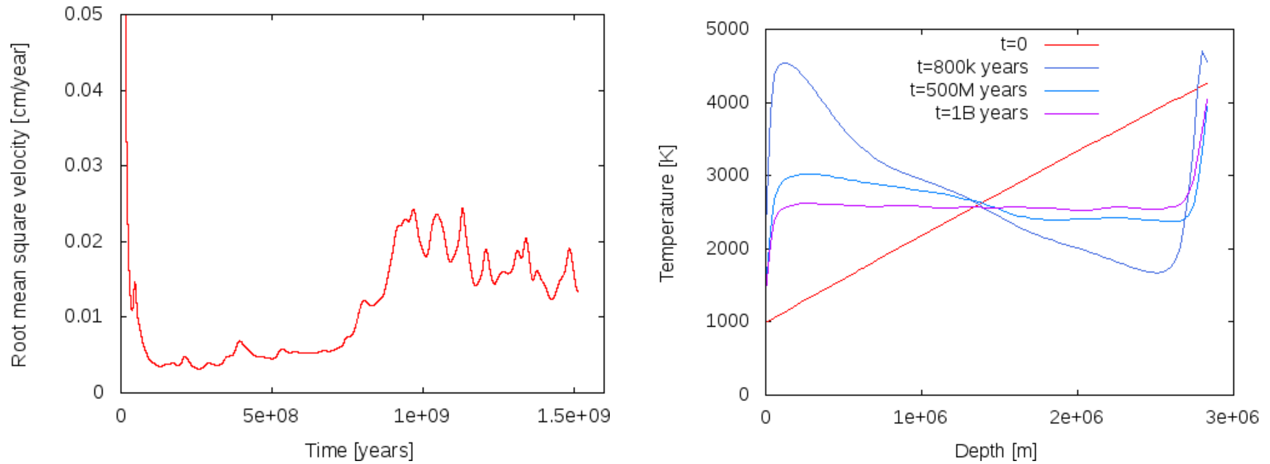


Figure 31: *Simple convection in a quarter of an annulus. Left: Root mean square values of the velocity field. The initial spike (off the scale) is due to the overturning of the instable layering of the temperature. Convection is suppressed for the first 800 million years due to the stable layering that results from it. The maximal velocity encountered follows generally the same trend and is in the range of 2–3 cm/year between 100 and 800 million years, and 4–8 cm/year following that. Right: Average temperature at various depths for* $t = 0$*,* $t = 800,000$ *years,* $t = 5 \cdot 10^8$ *years, and* $t = 10^9$ *years.*

times after approximately 800 million years in this simulation to avoid the effects of our inappropriately chosen initial conditions. Likewise, we may actually want to choose initial conditions more like what we see in the model for later times, i.e., constant in depth with the exception of thin boundary layers, if we want to stick to incompressible models.

Material model. The model we use here involves viscosity, density, and thermal property functions that do not depend on the pressure, and only the density varies (slightly) with the temperature. We know that this is not the case in nature.

Shear heating. When we set up the input file, we started with a model that includes the shear heating term $2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u})$ in eq. (3). In hindsight, this may have been the wrong decision, but it provides an opportunity to investigate whether we think that the results of our computations can possibly be correct.

We first realized the issue when looking at the heat flux that the `heat flux statistics` postprocessor computes. This is shown in the left panel of Fig. 32.[29] There are two issues one should notice here. The more obvious one is that the flux from the mantle to the air is consistently higher than the heat flux from core to mantle. Since we have no radiogenic heating model selected (see the `Model name` parameter in the `Heating model` section of the input file; see also Section 5.44), in the long run the heat output of the mantle must equal the input, unless is cools. Our misconception was that after the 800 million year transition, we believed that we had reached a steady state where the average temperature remains constant and convection simply moves heat from the core-mantle boundary the surface. One could also be tempted to believe this from the right panel in Fig. 31 where it looks like the average temperature does at least not change dramatically. But, it is easy to convince oneself that that is not the case: the `temperature statistics` postprocessor we had previously selected also outputs data about the mean temperature in the model, and it looks like shown in the left panel of Fig. 33. Indeed, the average temperature drops over the course of the 1.2 billion years shown here. We could now convince ourselves that indeed the loss of thermal energy in the mantle

[29]The `heat flux statistics` postprocessor computes heat fluxes through parts of the boundary in *outward* direction, i.e., from the mantle to the air and to the core. However, we are typically interested in the flux from the core into the mantle, so the figure plots the negative of the computed quantity.

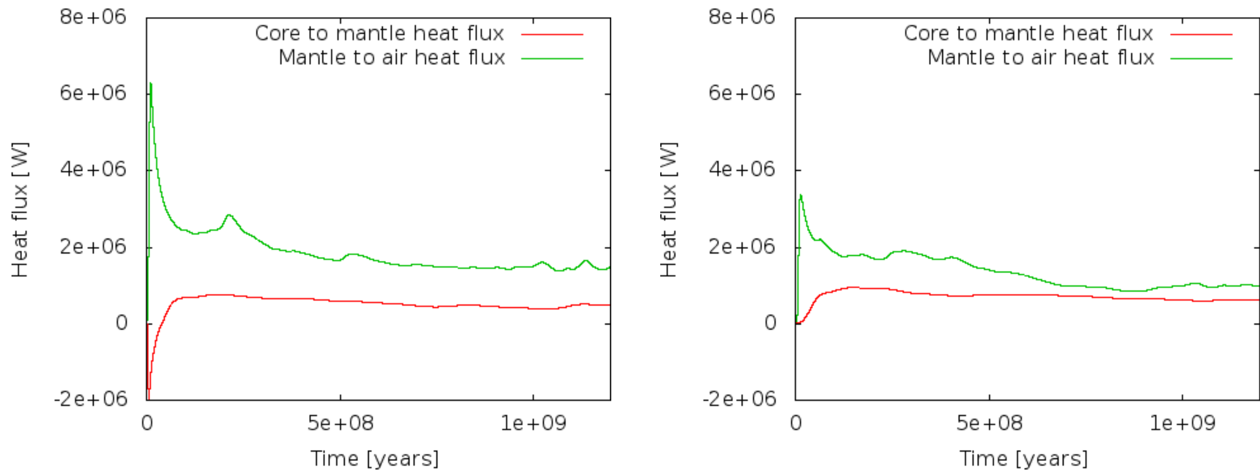


Figure 32: *Simple convection in a quarter of an annulus. Left: Heat flux through the core-mantle and mantle-air boundaries of the domain for the model with shear heating. Right: Same for a model without shear heating.*

due to the drop in average temperature is exactly what fuels the persistently imbalanced energy outflow. In essence, what this would show is that if we kept the temperature at the boundaries constant, we would have chosen a mantle that was initially too hot on average to be sustained by the boundary values and that will cool until it will be in energetic balance and on longer time scales, in- and outflow of thermal energy would balance each other.

However, there is a bigger problem. Fig. 32 shows that at the very beginning, there is a spike in energy flux through the outer boundary. We can explain this away with the imbalanced initial temperature field that leads to an overturning and, thus, a lot of hot material rising close to the surface that will then lead to a high energy flux towards the cold upper boundary. But, worse, there is initially a *negative* heat flux into the mantle from the core – in other words, the mantle is *losing* energy to the core. How is this possible? After all, the hottest part of the mantle in our initial temperature field is at the core-mantle boundary, no thermal energy should be flowing from the colder overlying material towards the hotter material at the boundary! A glimpse of the solution can be found in looking at the average temperature in Fig. 33: At the beginning, the average temperature *rises*, and apparently there are parts of the mantle that become hotter than the 4273 K we have given the core, leading to a downward heat flux. This heating can of course only come from the shear heating term we have accidentally left in the model: at the beginning, the instable layering leads to very large velocities, and large velocities lead to large velocity gradients that in turn lead to a lot of shear heating! Once the initial overturning has subsided, after say 100 million years (see the mean velocity in Fig. 31), the shear heating becomes largely irrelevant and the cooling of the mantle indeed begins.

Whether this is really the case is of course easily verified: The right panels of Fig.s 32 and 33 show heat fluxes and average temperatures for a model where we have switched off the shear heating by setting

```
subsection Model settings
  set Include shear heating                     = false
end
```

Indeed, doing so leads to a model where the heat flux from core to mantle is always positive, and where the average temperature strictly drops!

Summary. As mentioned, we will address some of the issues we have identified as unrealistic in the following sections. However, despite all of this, some things are at least at the right order of magnitude, confirming that what ASPECT is computing is reasonable. For example, the maximal velocities encounted

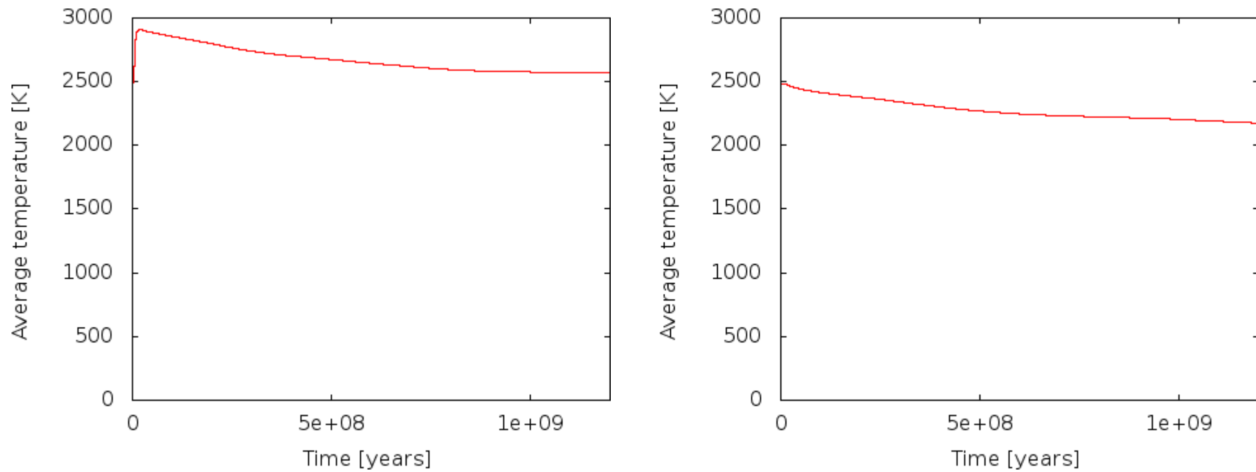


Figure 33: *Simple convection in a quarter of an annulus. Left: Average temperature throughout the model for the model with shear heating. Right: Same for a model without shear heating.*

in our model (after the 800 million year boundary) are in the range of 6–7cm per year, with occasional excursions up to 11cm. Clearly, something is going in the right direction.

6.3.2 Simple convection in a spherical 3d shell

The setup from the previous section can of course be extended to 3d shell geometries as well – though at significant computational cost. In fact, the number of modifications necessary is relatively small, as we will discuss below. To show an example up front, a picture of the temperature field one gets from such a simulation is shown in Fig. 34. The corresponding movie can be found at http://youtu.be/j63MkEc0RRw.

The input file. Compared to the input file discussed in the previous section, the number of changes is relatively small. However, when taking into account the various discussions about which parts of the model were or were not realistic, they go throughout the input file, so we reproduce it here in its entirety, interspersed with comments (the full input file can also be found in cookbooks/shell_simple_3d.prm). Let us start from the top where everything looks the same except that we set the dimension to 3:

```
set Dimension                              = 3
set Use years in output instead of seconds = true
set End time                               = 1.5e9
set Output directory                       = output


subsection Material model
  set Model name = simple

  subsection Simple model
    set Thermal expansion coefficient = 4e-5
    set Viscosity                     = 1e22
  end
end
```

The next section concerns the geometry. The geometry model remains unchanged at "spherical shell" but we omit the opening angle of 90 degrees as we would like to get a complete spherical shell. Such a shell of course also only has two boundaries (the inner one has indicator zero, the outer one indicator one) and consequently these are the only ones we need to list in the "Model settings" section:

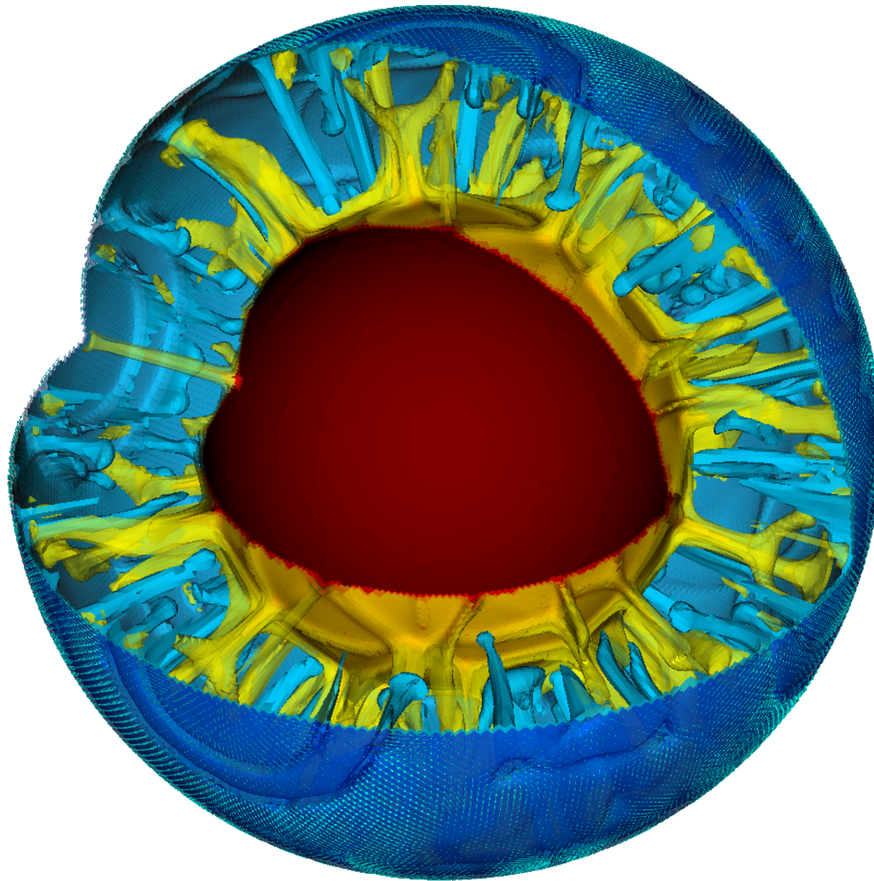


Figure 34: *Convection in a spherical shell: Snapshot of isosurfaces of the temperature field at time* $t \approx 1.06 \cdot 10^9$ *years with a quarter of the geometry cut away. The surface shows vectors indicating the flow velocity and direction.*

```
subsection Geometry model
  set Model name = spherical shell

  subsection Spherical shell
    set Inner radius  = 3481000
    set Outer radius  = 6336000
  end
end


subsection Model settings
  set Zero velocity boundary indicators       = inner
  set Tangential velocity boundary indicators = outer
  set Prescribed velocity boundary indicators =

  set Fixed temperature boundary indicators   = inner, outer

  set Include shear heating                   = false
end
```

Next, since we convinced ourselves that the temperature range from 973 to 4273 was too large given that we do not take into account adiabatic effects in this model, we reduce the temperature at the inner edge of the mantle to 1973. One can think of this as an approximation to the real temperature there minus the amount of adiabatic heating material would experience as it is transported from the surface to the core-mantle boundary. This is, in effect, the temperature difference that drives the convection (because a completely adiabatic temperature profile is stable despite the fact that it is much hotter at the core mantle boundary than at the surface). What the real value for this temperature difference is, is unclear from current research, but it is thought to be around 1000 Kelvin, so let us choose these values.

```
subsection Boundary temperature model
  set Model name = spherical constant
  subsection Spherical constant
    set Inner temperature = 1973
    set Outer temperature = 973
  end
end
```

The second component to this is that we found that without adiabatic effects, an initial temperature profile that decreases the temperature from the inner to the outer boundary makes no sense. Rather, we expected a more or less constant temperature with boundary layers at both ends. We could describe such an initial temperature field, but since any initial temperature is mostly arbitrary anyway, we opt to just assume a constant temperature in the middle between the inner and outer temperature boundary values and let the simulation find the exact shape of the boundary layers itself:

```
subsection Initial conditions
  set Model name = function
  subsection Function
    set Function expression = 1473
  end
end

subsection Gravity model
  set Model name = radial earth-like
end
```

As before, we need to determine how many mesh refinement steps we want. In 3d, it is simply not possible to have as much mesh refinement as in 2d, so we choose the following values that lead to meshes that have, after an initial transitory phase, between 1.5 and 2.2 million cells and 50–75 million unknowns:

```
subsection Mesh refinement
  set Initial global refinement          = 2
  set Initial adaptive refinement        = 3
  set Strategy                           = temperature
  set Time steps between mesh refinement = 15
end
```

Second to last, we specify what we want ASPECT to do with the solutions it computes. Here, we compute the same statistics as before, and we again generate graphical output every million years. Computations of this size typically run with 1000 MPI processes, and it is not efficient to let every one of them write their own file to disk every time we generate graphical output; rather, we group all of these into a single file to keep file systems reasonably happy. Likewise, to accomodate the large amount of data, we output depth averaged fields in VTU format since it is easier to visualize:

```
subsection Postprocess
  set List of postprocessors = visualization, velocity statistics, \
                temperature statistics, heat flux statistics, \
```

```
              depth average

  subsection Visualization
    set Output format                  = vtu
    set Time between graphical output = 1e6
    set Number of grouped files        = 1
  end

  subsection Depth average
    set Time between graphical output = 1.5e6
    set Output format                  = vtu
  end
end
```

Finally, we realize that when we run very large parallel computations, nodes go down or the scheduler aborts programs because they ran out of time. With computations this big, we cannot afford to just lose the results, so we checkpoint the computations every 50 time steps and can then resume it at the last saved state if necessary (see Section 4.5):

```
subsection Checkpointing
  set Steps between checkpoint = 50
end
```

Evaluation. Just as in the 2d case above, there are still many things that are wrong from a physical perspective in this setup, notably the no-slip boundary conditions at the bottom and of course the simplistic material model with its fixed viscosity and its neglect for adiabatic heating and compressibility. But there are also a number of things that are already order of magnitude correct here.

For example, if we look at the heat flux this model produces, we find that the convection here produces approximately the correct number. Wikipedia's article on Earth's internal heat budget[30] states that the overall heat flux through the Earth surface is about $47 \cdot 10^{12}$ W (i.e., 47 terawatts) of which an estimated 12–30 TW are primordial heat released from cooling the Earth and 15–41 TW from radiogenic heating.[31] Our model does not include radiogenic heating (though ASPECT has a number of `Heating models` to switch this on, see Section 5.44) but we can compare what the model gives us in terms of heat flux through the inner and outer boundaries of our shell geometry. This is shown in the left panel of Fig. 35 where we plot the heat flux through boundaries zero and one, corresponding to the core-mantle boundary and Earth's surface. ASPECT always computes heat fluxes in outward direction, so the flux through boundary zero will be negative, indicating the we have a net flux *into* the mantle as expected. The figure indicates that after some initial jitters, heat flux from the core to the mantle stabilizes at around 4.5 TW and that through the surface at around 10 TW, the difference of 5.5 TW resulting from the overall cooling of the mantle. While we cannot expect our model to be quantitatively correct, this can be compared with estimates heat fluxes of 5–15 TW for the core-mantle boundary, and an estimated heat loss due to cooling of the mantle of 7–15 TW (values again taken from Wikipedia).

A second measure of whether these results make sense is to compare velocities in the mantle with what is known from observations. As shown in the right panel of Fig. 35, the maximal velocities settle to values on the order of 3 cm/year (each of the peaks in the line for the maximal velocity corresponds to a particularly large plume rising or falling). This is, again, at least not very far from what we know to be correct and we should expect that with a more elaborate material model we should be able to get even closer to reality.

[30] Not necessarily the most scientific source, but easily accessible and typically about right in terms of numbers. The numbers stated here are those listed on Wikipedia at the time this section was written in March 2014.

[31] As a point of reference, for the mantle an often used number for the release of heat due to radioactive decay is $7.4 \cdot 10^{-12}$ W/kg. Taking a density of 3300 kg/m^3 and a volume of 10^{12} m^3 would yield roughly $2.4 \cdot 10^{13}$ W of heat produced. This back of the envelope calculation lies within the uncertain range stated above.

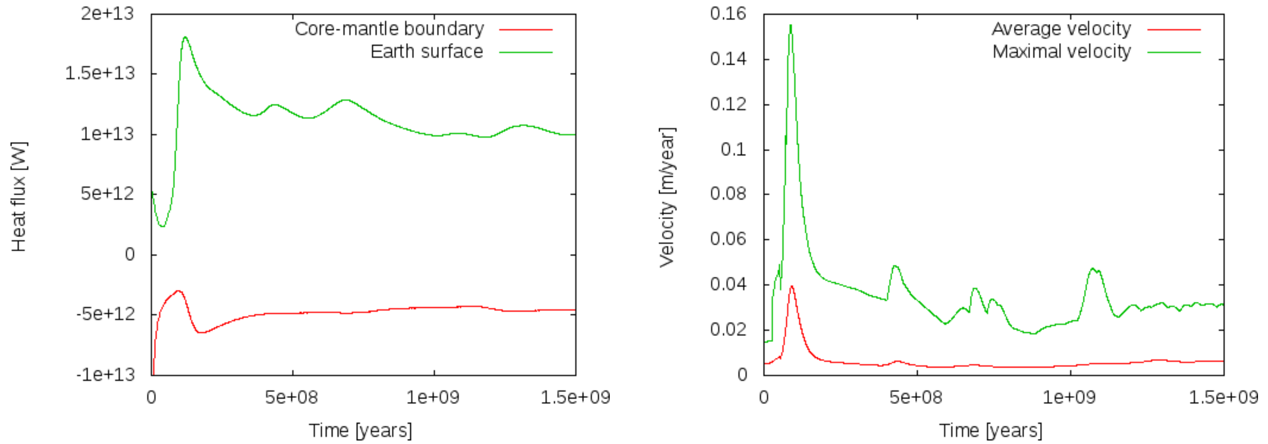


Figure 35: *Evaluating the 3d spherical shell model. Left: Outward heat fluxes through the inner and outer boundaries of the shell. Right: Average and maximal velocities in the mantle.*

6.3.3 3D convection with an Earth-like initial condition

This section was contributed by Jacqueline Austermann

For any model run with ASPECT we have to choose an initial condition for the temperature field. If we want to model convection in the Earth's mantle we want to choose an initial temperature distribution that captures the Earth's buoyancy structure. In this cookbook we present how to use temperature perturbations based on the shear wave velocity model S20RTS [RvH00] to initialize a mantle convection calculation.

The input shear wave model. The current version of ASPECT can read in the shear wave velocity models S20RTS [RvH00] and S40RTS [RDvHW11], which are located in `data/initial-conditions/S40RTS/`. Those models provide spherical harmonic coefficients up do degree 20 and 40, respectively, for 21 depth layers. The interpolation with depth is done through a cubic spline interpolation. The input files `S20RTS.sph` and `S40RTS.sph` were downloaded from `http://www.earth.lsa.umich.edu/~jritsema/Research.html` and have the following format (this example is S20RTS):

```
            20 111111111111111111111  24 000111111111111111111111
 0.1534E-01
 0.1590E-01 -0.1336E-01  0.3469E-02
-0.3480E-02  0.1165E-01  0.8376E-02  0.2158E-01 -0.9923E-02
 ...
```

The first number in the first line denotes the maximum degree. This is followed in the next line by the spherical harmonic coefficients from the surface down to the CMB. The coefficients are arranged in the following way:

a_{00}
a_{10} a_{11} b_{11}
a_{20} a_{21} b_{21} a_{22} b_{22}
...

a_{yz} is the cosine coefficient of degree y and order z and b_{yz} is the sine coefficient of degree y and order z. The depth layers are specified in the file `Spline_knots.txt` by a normalized depth value ranging from the CMB (3480km, normalized to -1) to the moho (6346km, normalized to 1). This is the original format provided on the homepage.

Any other perturbation model in this same format can also be used, one only has to specify the different filename in the parameter file (see next section). For models with different depth layers one has to adjust the `Spline_knots.txt` file as well as the number of depth layers, which is hard coded in the current code. A further note of caution when switching to a different input model concerns the normalization of the spherical harmonics, which might differ. After reading in the shear wave velocity perturbation one has several options to scale this into temperature differences, which are then used to initialize the temperature field.

Setting up the ASPECT model. For this cookbook we will use the parameter file provided in `cookbooks/S20RTS.prm`, which uses a 3d spherical shell geometry similar to section 6.3.2. This plugin is only sensible for a 3D spherical shell with Earth-like dimensions.

The relevant section in the input file is as follows:

```
subsection Initial conditions
  set Model name = S40RTS perturbation
  subsection S40RTS perturbation
    set Data directory                 = $ASPECT_SOURCE_DIR/data/initial-conditions/S40RTS/
    set Initial condition file name    = S20RTS.sph
    set Spline knots depth file name   = Spline_knots.txt
    set Remove degree 0 from perturbation = false
    set Vs to density scaling          = 0.2
    set Thermal expansion coefficient in initial temperature scaling = 3e-5
    set Reference temperature          = 1600
    set Remove temperature heterogeneity down to specified depth = 200000
  end
end
```

For this initial condition model we need to first specify the data directory in which the input files are located as well as the initial condition file (S20RTS.sph or S40RTS.sph) and the file that contains the normalized depth layers (Spline knots depth file name). We next have the option to remove the degree 0 perturbation from the shear wave model. This might be the case if we want to make sure that the depth average temperature follows the background (adiabatic or constant) temperature.

The next input parameters describe the scaling from the shear wave velocity perturbation to the final temperature field. The shear wave velocity perturbation $\delta v_s/v_s$ (that is provided by S20RTS) is scaled into a density perturbation $\delta\rho/\rho$ with a constant that is specified in the initial condition section of the input parameter file as 'Vs to density scaling'. Here we choose a constant scaling of 0.2. This perturbation is further translated into a temperature difference ΔT by multiplying it by the negative inverse of thermal expansion, which is also specified in this section of the parameter file as 'Thermal expansion coefficient in initial temperature scaling'. This temperature difference is then added to the background temperature, which is the adiabatic temperature for a compressible model or the reference temperature (as specified in this section of the parameter file) for an incompressible model. Features in the upper mantle such as cratons might be chemically buoyant and therefore isostatically compensated, in which case their shear wave perturbation would not contribute buoyancy variations. We therefore included an additional option to zero out temperature perturbations within a certain depth. In this example we set this parameter 'Remove temperature heterogeneity down to specified depth' to 200km. The chemical variation within the mantle might require a more sophisticated 'Vs to density' scaling that varies for example with depth or as a function of the perturbation itself, which is not captured in this model. The described procedure provides an absolute temperature for every point, which will only be adjusted at the boundaries if indicated in the Boundary temperature model.

Visualizing 3D models. In this cookbook we calculate the instantaneous solution to examine the flow field. Figure 36 shows the density field for a resolution of 2 global refinement steps (bottom right) as used in the cookbook, as well as 4 global refinement steps (other panels in this figure). The top panels show the density variation that has been obtained from scaling S20RTS and removing perturbations in the upper

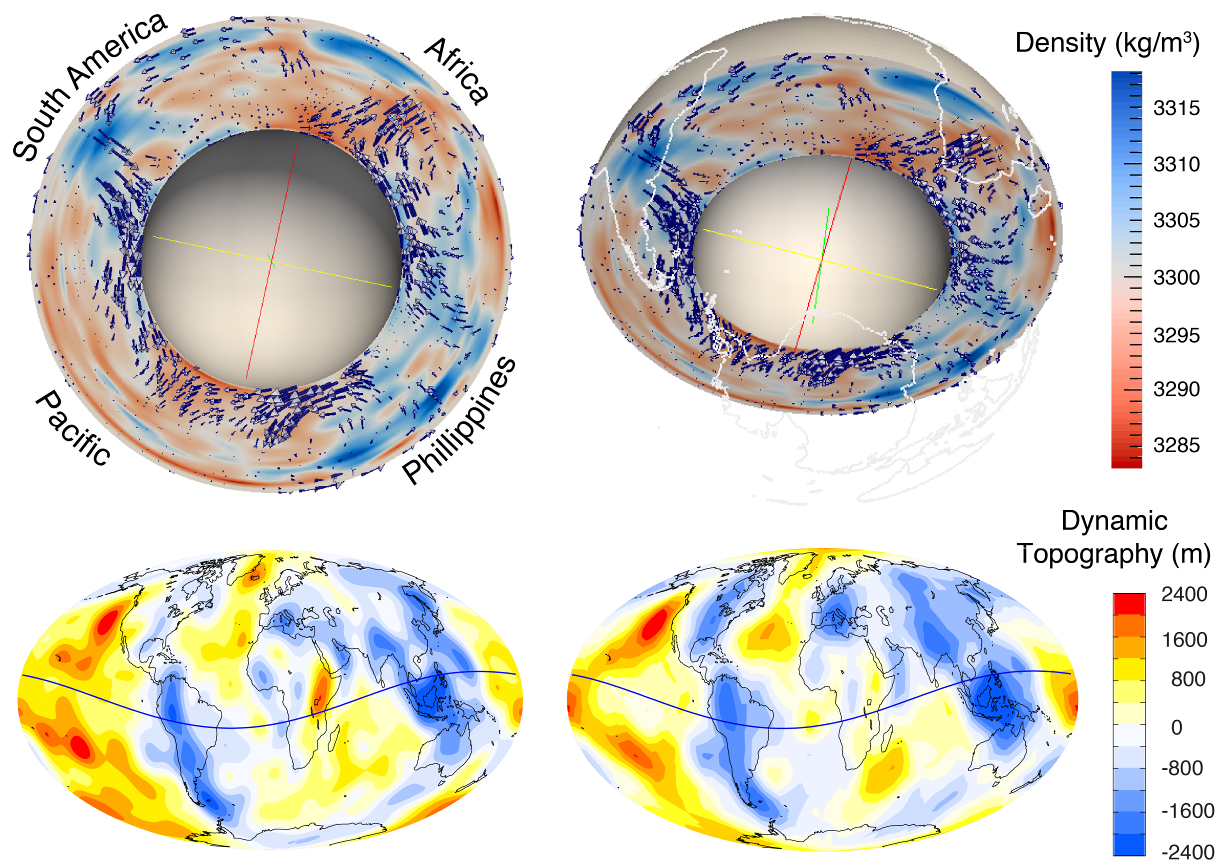


Figure 36: *The top panels show the density distribution as prescribed from the shear wave velocity model S20RTS and the resulting flow for a global refinement of 4. This model assumes a constant scaling between shear wave and density perturbations and zeros out perturbations in the upper 200km. The bottom figures show the great circle (blue line) along which the top slices are evaluated. They also show the resulting dynamic topography for a global refinement of 4 (left) and 2 (right, cookbook).*

200km. One can see the two large low shear wave velocity perturbations underneath Africa and the Pacific that lead to upwelling if they are assumed to be buoyant (as is done in this case). One can also see the subducting slabs underneath South America and the Philippine region that lead to local downwelling. This flow produces dynamic topography on the surface, which is shown in the bottom row (4 refinement steps on the left, 2 refinement steps on the right). One can see that subduction zones are visible as depressed topography due to the downward flow, while regions such as Iceland, Hawaii, or mid ocean ridges are elevated due to (deep and) shallow upward flow. This model uses a highly simplified material model that is incompressible and isoviscous and does therefore not represent real mantle flow. More realistic material properties, density scaling as well as boundary conditions will affect the magnitude and pattern shown here.

Use cookbooks/pe box.prm

6.3.4 Using reconstructed surface velocities by GPlates

Finish the GPlates sec tion

This section was contributed by René Gaßmöller

In a number of model setups one may want to include a surface velocity boundary condition that prescribes

the velocity according to a specific geologic reconstruction. The purpose of this kind of models is often to test a proposed geologic model and compare characteristic convection results to present-day observables in order to gain information about the initially assumed geologic input. In this cookbook we present ASPECT's interface to the widely used plate reconstruction software GPlates, and the steps to go from a geologic plate reconstruction to a geodynamic model incorporating these velocities as boundary condition.

Acquiring a plate reconstruction. The plate reconstruction that is used in this cookbook is included in the `data/velocity-boundary-conditions/gplates/` directory of your ASPECT installation. For a new model setup however, a user eventually needs to create her own data files, and so we will briefly discuss the process of acquiring a usable plate reconstruction and transferring it into a format usable by ASPECT. Both the necessary software and data are provided by the GPlates project. GPlates is an open-source software for interactive visualization of plate tectonics. It is developed by the EarthByte Project in the School of Geosciences at the University of Sydney, the Division of Geological and Planetary Sciences (GPS) at CalTech and the Center for Geodynamics at the Norwegian Geological Survey (NGU). For extensive documentation and support we refer to the GPlates website (`http://www.gplates.org`). Apart from the software one needs the actual plate reconstruction that consists of closed polygons covering the complete model domain. For our case we will use the data provided by [GTZ+12] that is available from the GPlates website under "Download → Download GPlates-compatible data → Global reconstructions with continuously closing plates from 140 Ma to the present". The data is provided under a Creative Commons Attribution 3.0 Unported License (`http://creativecommons.org/licenses/by/3.0/`).

Converting GPlates data to ASPECT input. After loading the data files into GPlates (*.gpml for plate polygons, *.rot for plate rotations over time) the user needs to convert the GPlates data to velocity information usable in ASPECT. The purpose of this step is to convert from the description GPlates uses internally (namely a representation of plates as polygons that rotate with a particular velocity around a pole) to one that can be used by ASPECT (which needs velocity vectors defined at individual points at the surface).

With loaded plate polygon and rotation information the conversion from GPlates data to ASPECT-readable velocity files is rather straightforward. First the user needs to generate (or import) so-called "velocity domain points", which are discrete sets of points at which GPlates will evaluate velocity information. This is done using the "Features → Generate Velocity Domain Points → Latitude Longitude" menu option. Because ASPECT is using an adaptive mesh it is not possible for GPlates to generate velocity information at the precise surface node positions like for CitcomS or Terra (the other currently available interfaces). Instead GPlates will output the information on a general Latitude/Longitude grid with nodes on all crossing points. ASPECT then internally interpolates this information to the current node locations during the model run. This requires the user to choose a sensible resolution of the GPlates output, which can be adjusted in the "Generate Latitude/Longitude Velocity Domain Points" dialog of GPlates. In general a resolution that resolves the important features is necessary, while a resolution that is higher than the maximal mesh size for the ASPECT model is unnecessary and only increases the computational cost and memory consumption of the model.

Important note: The Mesh creation routine in GPlates has significantly changed from version 1.3 to 1.4. In GPlates 1.4 and later the user has to make sure that the number of longitude intervals is set as twice the number of latitude intervals, the "Place node points at centre of latitude/longitude cells" box is **un**checked and the "Latitude/Longitude extents" are set to "Use Global Extents". ASPECT does check for most possible combinations that can not be read and will cancel the calculation in these cases, however some mistakes can not be checked against from the information provided in the GPlates file.

After creating the Velocity Domain Points the user should see the created points and their velocities indicated as points and arrows in GPlates. To export the calculated velocities one would use the "Reconstruction → Export" menu. In this dialog the user may specify the time instant or range at which the velocities shall be exported. The only necessary option is to include the "Velocities" data type in the "Add Export" sub-dialog. The velocities need to be exported in the native GPlates `*.gpml` format, which is based

on XML and can be read by ASPECT. In case of a time-range the user needs to add a pattern specifier to the name to create a series of files. The `%u` flag is especially suited for the interaction with ASPECT, since it can easily be replaced by a calculated file index (see also 6.3.4).

Setting up the ASPECT model. For this cookbook we will use the parameter file provided in `cookbooks/gplates-2d.prm` which uses the 2d shell geometry previously discussed in Section 6.3.1. ASPECT's GPlates plugin allows for the use of two- and three-dimensional models incorporating the GPlates velocities. Since the output by GPlates is threedimensional in any case, ASPECT internally handles the 2D model by rotating the model plane to the orientation specified by the user and projecting the plate velocities into this plane. The user specifies the orientation of the model plane by prescribing two points that define a plane together with the coordinate origin (i.e. in the current formulation only great-circle slices are allowed). The coordinates need to be in spherical coordinates θ and ϕ with θ being the colatitude (0 at north pole) and ϕ being the longitude (0 at Greenwich meridian, positive eastwards) both given in radians. The approach of identifying two points on the surface of the Earth along with its center allows to run computations on arbitrary two-dimensional slices through the Earth with realistic boundary conditions.

The relevant section of the input file is then as follows:

```
subsection Model settings
  set Prescribed velocity boundary indicators = outer:gplates
  set Tangential velocity boundary indicators = inner

  set Fixed temperature boundary indicators   = inner, outer
  set Include shear heating                   = false
end

subsection Boundary velocity model
  subsection GPlates model
    set Data directory = $ASPECT_SOURCE_DIR/data/velocity-boundary-conditions/gplates/
    set Velocity file name = current_day.gpml
    set Time step = 1e6
    set Point one = 1.5708,4.87
    set Point two = 1.5708,5.24
    set Interpolation width = 2000000
  end
end
```

In the model settings subsection the user prescribes the boundary that is supposed to use the GPlates plugin. Although currently nothing forbids the user to use GPlates plugin for other boundaries than the surface, its current usage and the provided sample data only make sense for the surface of a spherical shell (boundary number 1 in the above provided parameter file). In case you are familiar with this kind of modelling and the plugin you could however also use it to prescribe mantle movements *below* a lithosphere model. All plugin specific options may be set in section 5.20. Possible options include the data directory and file name of the velocity file/files, the time step (in model units, mostly seconds or years depending on the "`Use years in output instead of seconds`" flag) and the points that define the 2D plane. The parameter "`Interpolation width`" is used to smooth the provided velocity files by a moving average filter. All velocity data points within this distance are averaged to determine the actual boundary velocity at a certain mesh point. This parameter is usually set to 0 (no interpolation, use nearest velocity point data) and is only needed in case the original setting is unstable or slowly converging.

Comparing and visualizing 2D and 3D models. The implementation of plate velocities in both two- and three-dimensional model setups allows for an easy comparison and test for common sources of error in the interpretation of model results. The left top figure in Fig. 37 shows a modification of the above presented parameter file by setting "`Dimension = 3`" and "`Initial global refinement = 3`". The top right plot of

Fig. 37 shows an example of three independent two-dimensional computations of the same reduced resolution. The models were prescribed to be orthogonal slices by setting:

```
    set Point one = 3.1416,0.0
    set Point two = 1.5708,0.0
```

and

```
    set Point one = 3.1416,1.5708
    set Point two = 1.5708,1.5708
```

The results of these models are plotted simultaneously in a single three-dimensional figure in their respective model planes. The necessary information to rotate the 2D models to their respective planes (rotation axis and angle) is provided by the GPlates plugin in the beginning of the model output. The bottom plot of Fig. 37 finally shows the results of the original cookbooks/gplates-2d.prm also in the three mentioned planes.

Now that we have model output for otherwise identical 2D and 3D models with equal resolution and additional 2D output for a higher resolution an interesting question to ask would be: What additional information can be created by either using three-dimensional geometry or higher resolution in mantle convection models with prescribed boundary velocities. As one can see in the comparison between the top right and bottom plot in Fig. 37 additional resolution clearly improves the geometry of small scale features like the shape of up- and downwellings as well as the maximal temperature deviation from the background mantle. However, the limitation to two dimensions leads to inconsistencies, that are especially apparent at the cutting lines of the individual 2D models. Note for example that the Nacza slab of the South American subduction zone is only present in the equatorial model plane and is not captured in the polar model plane west of the South American coastline. The (coarse) three-dimensional model on the other hand shows the same location of up- and downwellings but additionally provides a consistent solution that is different from the two dimensional setups. Note that the Nazca slab is subducting eastward, while all 2D models (even in high resolution) predict a westward subduction.

Finally we would like to emphasize that these models (especially the used material model) are way too simplified to draw any scientific conclusion out of it. Rather it is thought as a proof-of-concept what is possible with the dimension independent approach of ASPECT and its plugins.

Time-dependent boundary conditions. The example presented above uses a constant velocity boundary field that equals the present day plate movements. For a number of purposes one may want to use a prescribed velocity boundary condition that changes over time, for example to investigate the effect of trench migration on subduction. Therefore ASPECT's GPlates plugin is able to read in multiple velocity files and linearly interpolate between pairs of files to the current model time. To achieve this, one needs to use the `%d` wildcard in the velocity file name, which represents the current velocity file index (e.g. `time_dependent.%d.gpml`). This index is calculated by dividing the current model time by the user-defined time step between velocity files (see parameter file above). As the model time progresses the plugin will update the interpolation accordingly and if necessary read in new velocity files. In case it can not read the next velocity file, it assumes the last velocity file to be the constant boundary condition until the end of the model run. One can test this behavior with the provided data files `data/velocity_boundary_conditions/gplates/time_dependent.%d.gpml` with the index `d` ranging from 0 to 3 and representing the plate movements of the last 3 million years corresponding to the same plate reconstruction as used above. Additionally, the parameter `Velocity file start time` allows for a period of no-slip boundary conditions before starting the use of the GPlates plugin. This is a comfort implementation, which could also be achieved by using the checkpointing possibility described in section 4.5.

6.3.5 Reproducing rheology of Morency and Doin, 2004

This section was contributed by Jonathan Perry-Houts

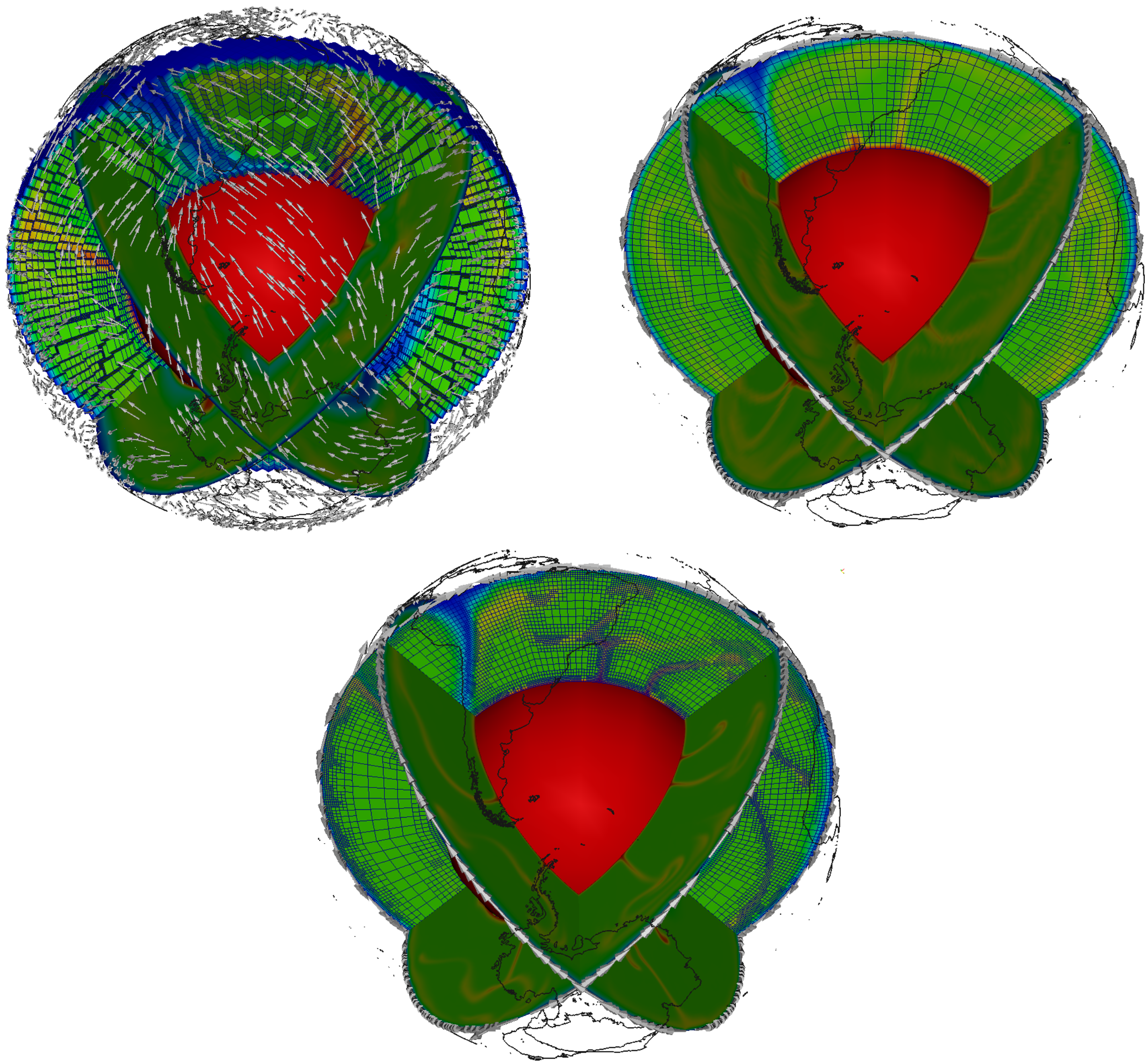


Figure 37: *Using GPlates for velocity boundary conditions: The top left figure shows the results of a three-dimensional model using the present day plate velocities provided by GPlates as surface boundary condition. The top right figure shows three independent computations on two-dimensional slices through Earth. The boundary conditions for each of these slices (white arrows) are tangential to the slices and are projections of the three-dimensional velocity vectors into the two-dimensional plane occupied by the slice. While the two top models are created with the same mesh resolution the bottom figure shows three independent two-dimensional models using a higher resolution. The view is centered on South America with Antarctica being near the bottom of the figure (coastlines provided by NGU and the GPlates project).*

Modeling interactions between the upper mantle and the lithosphere can be difficult because of the dynamic range of temperatures and pressures involved. Many simple material models will assign very high viscosities at low temperature thermal boundary layers. The pseudo-brittle rheology described in [MD04] was developed to limit the strength of lithosphere at low temperature. The effective viscosity can be described as the harmonic mean of two non-Newtonian rheologies:

$$v_{\text{eff}} = \left(\frac{1}{v_{\text{eff}}^v} + \frac{1}{v_{\text{eff}}^p}\right)^{-1}$$

where

$$v_{\text{eff}}^v = B\left(\frac{\dot{\epsilon}}{\dot{\epsilon}_{ref}}\right)^{-1+1/n_v} \exp\left(\frac{E_a + V_a\rho_m g z}{n_v R T}\right),$$

$$v_{\text{eff}}^p = (\tau_0 + \gamma\rho_m g z)\left(\frac{\dot{\epsilon}^{-1+1/n_p}}{\dot{\epsilon}_{ref}^{1/n_p}}\right),$$

where B is a scaling constant; $\dot{\epsilon}$ is defined as the quadratic sum of the second invariant of the strain rate tensor and a minimum strain rate, $\dot{\epsilon}_0$; $\dot{\epsilon}_{ref}$ is a reference strain rate; n_v, and n_p are stress exponents; E_a is the activation energy; V_a is the activation volume; ρ_m is the mantle density; R is the gas constant; T is temperature; τ_0 is the cohesive strength of rocks at the surface; γ is a coefficient of yield stress increase with depth; and z is depth.

By limiting the strength of the lithosphere at low temperature, this rheology allows one to more realistically model processes like lithospheric delamination and foundering in the presence of weak crustal layers. A similar model setup to the one described in [MD04] can be reproduced with the following parameters.

Note: [MD04] defines the second invariant of the strain rate in a nonstandard way. The formulation in the paper is given as $\epsilon_{II} = \sqrt{\frac{1}{2}(\epsilon_{11}^2 + \epsilon_{12}^2)}$, where ϵ is the strain rate tensor. For consistency, that is also the formulation implemented in ASPECT. Because of this irregularity it is inadvisable to use this material model for purposes beyond reproducing published results.

Note: The viscosity profile in Figure 1 of [MD04] appears to be wrong. The published parameters do not reproduce those viscosities; it is unclear why. The values used here get very close. See Figure 38 for an approximate reproduction of the original figure.

```
set Dimension = 2
set Maximum time step = 1e4
set Nonlinear solver scheme = iterated IMPES

subsection Geometry model
  set Model name = box

  subsection Box
    set X extent      = 3000e3
    set Y extent      = 750e3
    set X repetitions = 4
  end
end

subsection Model settings
  set Fixed temperature boundary indicators   = top, bottom
  set Tangential velocity boundary indicators = top, bottom, left, right
```

```
end

subsection Compositional fields
  set Number of fields = 2
  set Names of fields  = upper_crust, lower_crust
end

subsection Compositional initial conditions
  set Model name = function

  subsection Function
    set Variable names = x,y
    set Function expression = if(y>=725e3,1,0);if((y<725e3&y>700e3),1,0)
  end
end

subsection Initial conditions
  set Model name = function

  subsection Function
    set Variable names = x,y
    set Function constants = h=750e3, w=3000e3, mantleT=1350 # deg C
    set Function expression = \
      if( y < 100e3, \
        (100e3-y)/100e3*(1600-mantleT)+mantleT+293, \
        if(y>650e3, \
          (h-y)/(100e3)*mantleT+293, \
          mantleT+293))
  end
end

subsection Material model
  set Model name = Morency and Doin

  subsection Morency and Doin
    set Densities = 3300,2920,2920
    set Activation energies = 500,320,320
    set Coefficient of yield stress increase with depth = 0.25
    set Thermal expansivities = 3.5e-5
    set Stress exponents for viscous rheology = 3
    set Stress exponents for plastic rheology = 30
    set Thermal diffusivity = 0.8e-6
    set Heat capacity = 1.25e3
    set Activation volume = 6.4e-6
    set Reference strain rate = 6.4e-16
    set Preexponential constant for viscous rheology law = 7e11 ## Value used in paper is 1.24e14
    set Cohesive strength of rocks at the surface = 117
    set Reference temperature = 293
    set Minimum strain rate = 5e-19                              ## Value used in paper is 1.4e-20
  end
end

subsection Boundary temperature model
  set Model name = initial temperature
end
```

```
subsection Boundary composition model
  set Model name = initial composition
end

subsection Gravity model
  set Model name = vertical
end

subsection Mesh refinement
  set Initial global refinement                = 5
  set Initial adaptive refinement              = 3
  set Strategy                                 = minimum refinement function
  subsection Minimum refinement function
    set Variable names = d,ignored
    set Function expression = if(d<100e3,8,5)
  end
end

subsection Postprocess
  set List of postprocessors = depth average

  subsection Depth average
    set Number of zones = 500
    set Output format = gnuplot
  end
end

subsection Termination criteria
  set Termination criteria = end step
  set End step = 0
end
```

6.3.6 Crustal deformation

This section was contributed by Cedric Thieulot, and makes use of the Drucker-Prager material model written by Anne Glerum and the free surface plugin by Ian Rose.

This is a simple example of an upper-crust undergoing compression or extension. It is characterized by a single layer of visco-plastic material subjected to basal kinematical boundary conditions. In compression, this setup is somewhat analogous to [Wil99], and in extension to [AHT11].

Brittle failure is approximated by adapting the viscosity to limit the stress that is generated during deformation. This "cap" on the stress level is parameterized in this experiment by the pressure-dependent Drucker Prager yield criterion and we therefore make use of the Drucker-Prager material model (see Section 5.66) in the `cookbooks/crustal_model_2D.prm`.

The layer is assumed to have dimensions of 80km × 16km and to have a density $\rho = 2800$ kg/m^3. The plasticity parameters are specified as follows:

```
subsection Material model
  set Model name = drucker prager
  subsection Drucker Prager
    set Reference density = 2800
    subsection Viscosity
       set Minimum viscosity = 1e19
       set Maximum viscosity = 1e25
       set Reference strain rate = 1e-20
       set Angle internal friction = 30
```

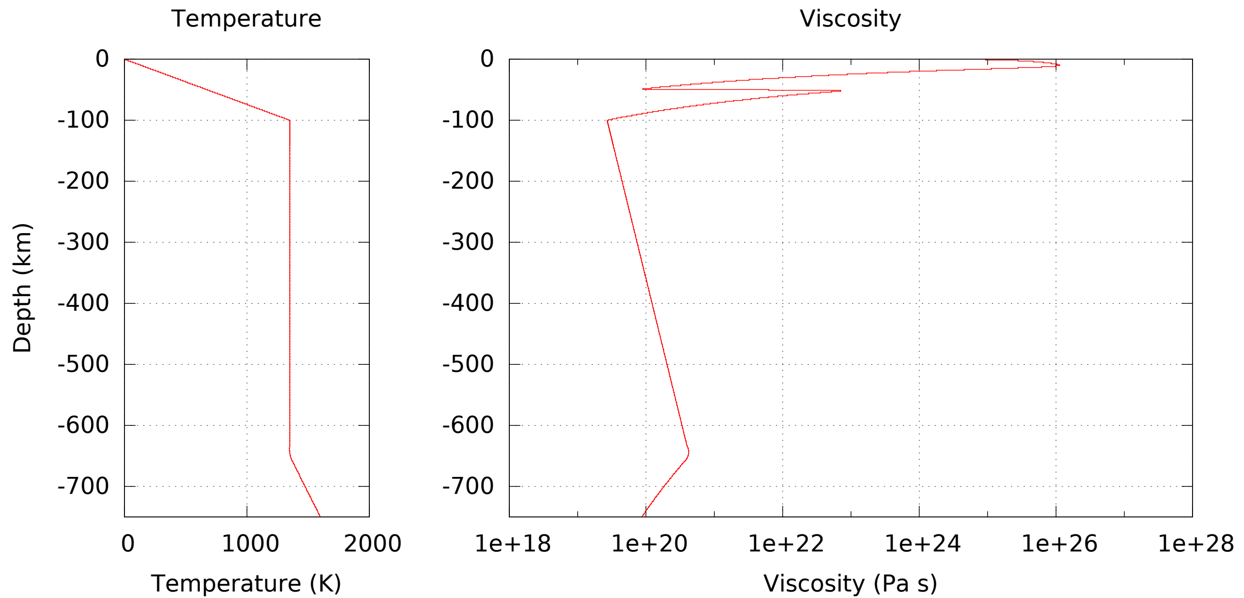


Figure 38: *Approximate reproduction of figure 1 from [MD04] using the 'morency doin' material model with almost all default parameters. Note the low-viscosity Moho, enabled by the low activation energy of the crustal component.*

```
      set Cohesion = 20e6
    end
  end
end
```

The yield strength σ_y is a function of pressure, cohesion and angle of friction (see Drucker-Prager material model in Section 5.66), and the effective viscosity is then computed as follows:

$$\mu_{\text{eff}} = \left(\frac{1}{\frac{\sigma_y}{2\dot{\epsilon}} + \mu_{\text{min}}} + \frac{1}{\mu_{\text{max}}} \right)^{-1}$$

where $\dot{\epsilon}$ is the square root of the second invariant of the deviatoric strain rate. The viscosity cutoffs insure that the viscosity remains within computationally acceptable values.

During the first iteration of the first timestep, the strain rate is zero, so we avoid dividing by zero by setting the strain rate to `Reference strain rate`.

The top boundary is a free surface while the left, right and bottom boundaries are subjected to the following boundary conditions:

```
subsection Boundary velocity model
  subsection Function
    set Variable names      = x,y
    set Function constants  = cm=0.01, year=1
    set Function expression =  if (x<40e3 , 1*cm/year, -1*cm/year) ; 0
  end
end
```

Note that compressive boundary conditions are simply achieved by reversing the sign of the imposed velocity.

The free surface will be advected up and down according to the solution of the Stokes solve. We have a choice whether to advect the free surface in the direction of the surface normal or in the direction of the local vertical (i.e., in the direction of gravity). For small deformations, these directions are almost the same, but in this example the deformations are quite large. We have found that when the deformation is large, advecting the surface vertically results in a better behaved mesh, so we set the following in the free surface subsection:

```
subsection Free surface
  set Surface velocity projection = vertical
end
```

We also make use of the strain rate-based mesh refinement plugin:

```
subsection Mesh refinement
  set Initial adaptive refinement        = 1
  set Initial global refinement          = 3
  set Refinement fraction                = 0.95
  set Strategy                           = strain rate
  set Coarsening fraction                = 0.05
  set Time steps between mesh refinement = 1
  set Run postprocessors on initial refinement = true
end
```

Setting `set Initial adaptive refinement = 4` yields a series of meshes as shown in Fig. (39), all produced during the first timestep. As expected, we see that the location of the highest mesh refinement corresponds to the location of a set of conjugated shear bands.

If we now set this parameter to 1 and allow the simulation to evolve for 500kyr, a central graben or plateau (depending on the nature of the boundary conditions) develops and deepens/thickens over time, nicely showcasing the unique capabilities of the code to handle free surface large deformation, localised strain rates through visco-plasticity and adaptive mesh refinement as shown in Fig. (40).

Deformation localizes at the basal velocity discontinuity and plastic shear bands form at an angle of approximately 53° to the bottom in extension and 35° in compression, both of which correspond to the reported Arthur angle [Kau10, Bui12].

Extension to 3D We can easily modify the previous input file to produce `crustal_model_3D.prm` which implements a similar setup, with the additional constraint that the position of the velocity discontinuity varies with the y-coordinate, as shown in Fig. (41). The domain is now $128 \times 96 \times 16$km and the boundary conditions are implemented as follows:

```
subsection Boundary velocity model
  subsection Function
    set Variable names      = x,y,z
    set Function constants  = cm=0.01, year=1
    set Function expression =  if (x<56e3 && y<=48e3 | x<72e3 && y>48e3,-1*cm/year,1*cm/year);0;0
  end
end
```

The presence of an offset between the two velocity discontinuity zones leads to a transform fault which connects them.

The Finite Element mesh, the velocity, viscosity and strain rate fields are shown in Fig. (42) at the end of the first time steps. The reader is encouraged to run this setup in time to look at how the two grabens interact as a function of their initial offset [AHT11, AHT12, AHFT13].

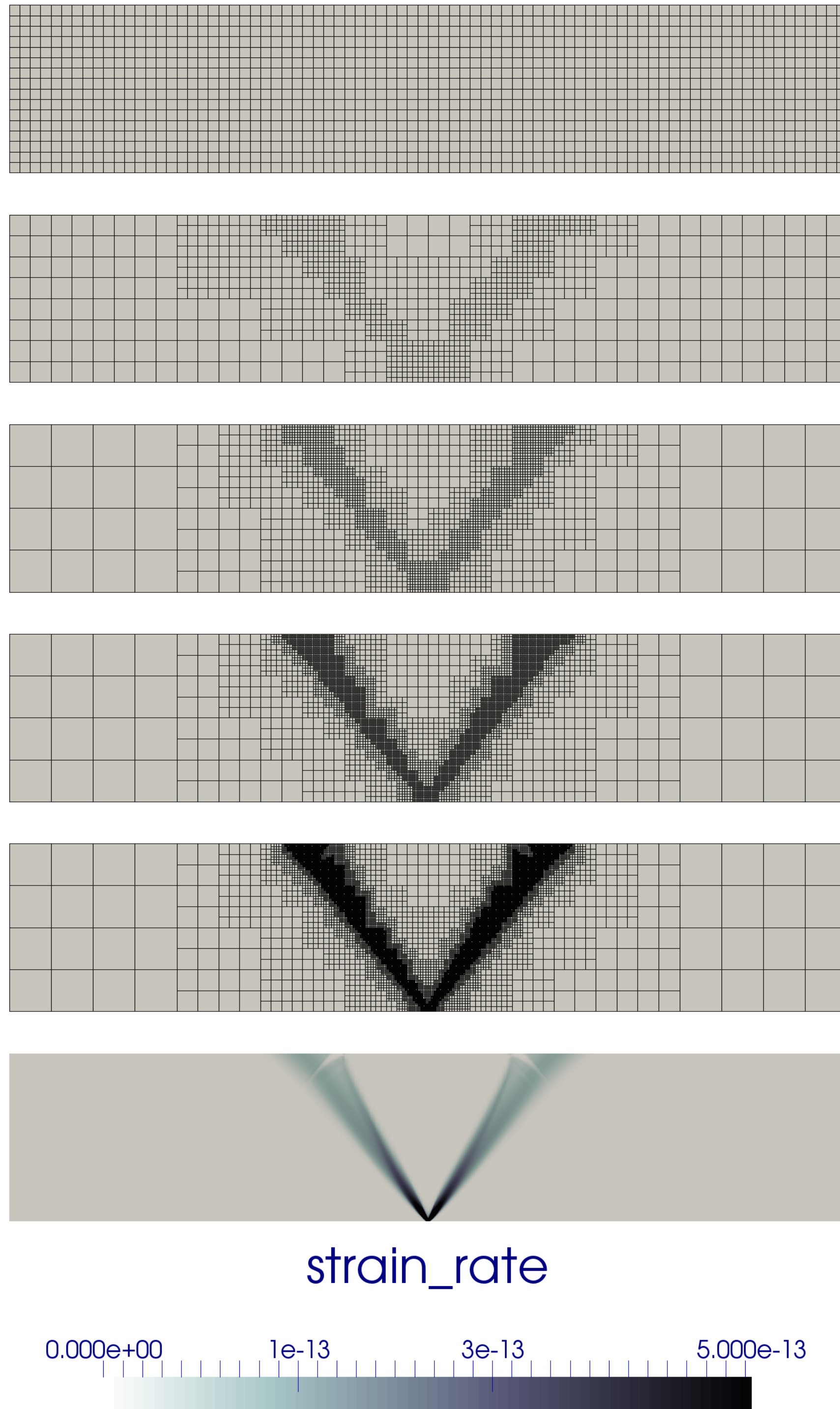


Figure 39: *Mesh evolution during the first timestep (refinement is based on strain rate).*

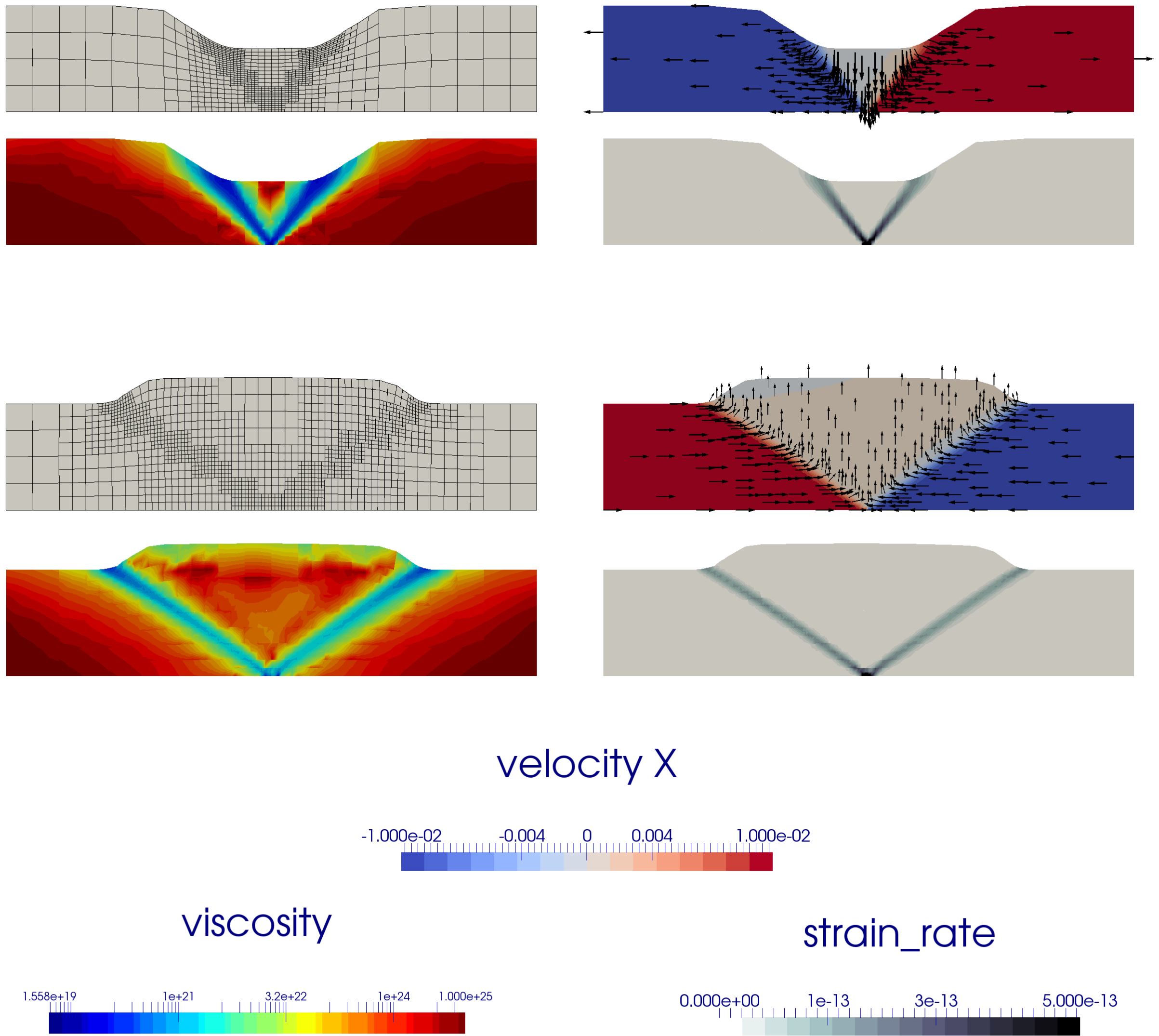


Figure 40: *Finite element mesh, velocity, viscosity and strain rate fields in the case of extensional boundary conditions (top) and compressive boundary conditions (bottom) at t=500kyr.*

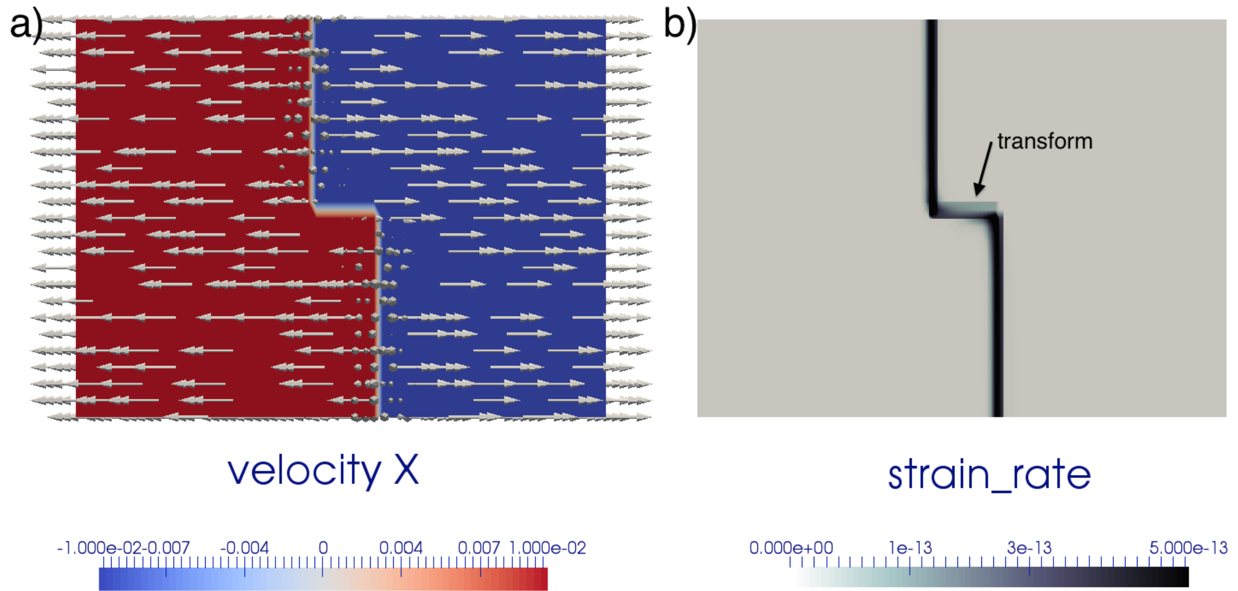


Figure 41: *Basal velocity boundary conditions and corresponding strain rate field for the 3D model.*

6.4 Benchmarks

Benchmarks are used to verify that a solver solves the problem correctly, i.e., to *verify* correctness of a code.[32] Over the past decades, the geodynamics community has come up with a large number of benchmarks. Depending on the goals of their original inventors, they describe stationary problems in which only the solution of the flow problem is of interest (but the flow may be compressible or incompressible, with constant or variable viscosity, etc), or they may actually model time-dependent processes. Some of them have solutions that are analytically known and can be compared with, while for others, there are only sets of numbers that are approximately known. We have implemented a number of them in ASPECT to convince ourselves (and our users) that ASPECT indeed works as intended and advertised. Some of these benchmarks are discussed below. Numerical results for several of these benchmarks are also presented in [KHB12] in much more detail than shown here.

6.4.1 Running benchmarks that require code

Some of the benchmarks require plugins like custom material models, boundary conditions, or postprocessors. To not pollute ASPECT with all these purpose-built plugins, they are kept separate from the more generic plugins in the normal source tree. Instead, the benchmarks have all the necessary code in `.cc` files in the benchmark directories. Those are then compiled into a shared library that will be used by ASPECT if it is referenced in a `.prm` file. Let's take the SolCx benchmark as an example (see Section 6.4.3). The directory contains:

- `solcx.cc` – the code file containing a material model "SolCxMaterial" and a postprocessor "SolCx-Postprocessor",
- `solcx.prm` – the parameter file referencing these plugins,
- `CMakeLists.txt` – a cmake configuration that allows you to compile `solcx.cc`.

[32]Verification is the first half of the *verification and validation* (V&V) procedure: *verification* intends to ensure that the mathematical model is solved correctly, while *validation* intends to ensure that the mathematical model is correct. Obviously, much of the aim of computational geodynamics is to validate the models that we have.

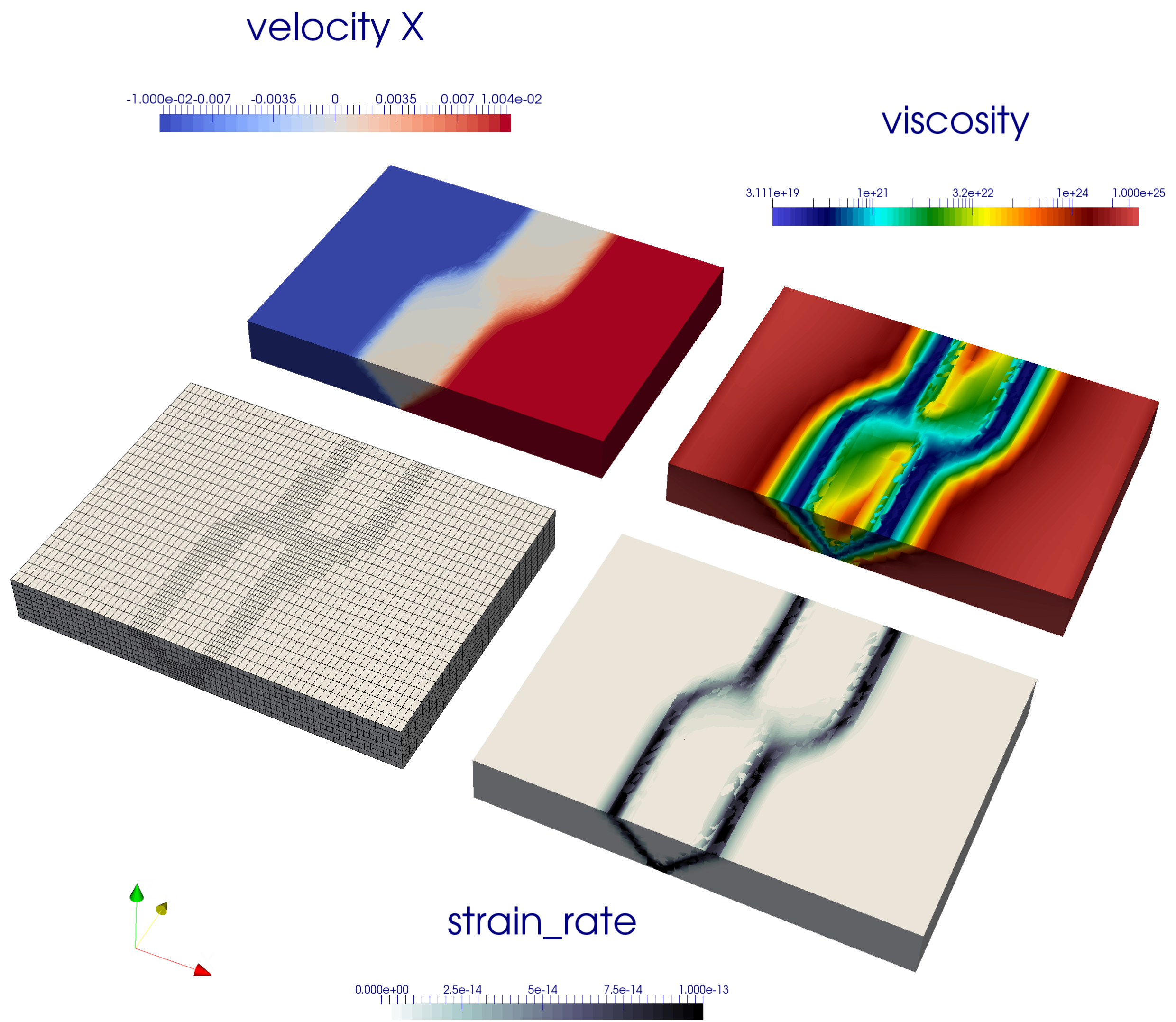


Figure 42: *Finite element mesh, velocity, viscosity and strain rate fields at the end of the first time step after one level of strain rate-based adaptive mesh refinement.*

To run this benchmark you need to follow the general outline of steps discussed in Section 7.2. For the current case, this amounts to the following:

1. Move into the directory of that particular benchmark:

   ```
   $ cd benchmark/solcx
   ```

2. Set up the project:

   ```
   $ cmake .
   ```

   By default, `cmake` will look for the ASPECT binary and other information in a number of directories relative to the current one. If it is unable to pick up where ASPECT was built and installed, you can specify this directory explicitly this using `-D ASPECT_DIR=<...>` as an additional flag to `cmake`, where <...> is the path to the build directory.

3. Build the library:

   ```
   $ make
   ```

   This will generate the file `libsolcx.so`.

Finally, you can run ASPECT with `solcx.prm`:

```
$ ../../aspect solcx.prm
```

where again you may have to use the appropriate path to get to the ASPECT executable. You will need to run ASPECT from the current directory because `solcx.prm` refers to the plugin as `./libsolcx.so`, i.e., in the current directory.

6.4.2 The van Keken thermochemical composition benchmark

This section is a co-production of Cedric Thieulot, Juliane Dannberg, Timo Heister and Wolfgang Bangerth with an extension to this benchmark provided by the Virginia Tech Dept. of Geosciences class Geodynamics and ASPECT co-taught by Scott King and D. Sarah Stamps.

One of the most widely used benchmarks for mantle convection codes is the isoviscous Rayleigh-Taylor case ("case 1a") published by van Keken *et al.* in [vKKS+97]. The benchmark considers a 2d situation where a lighter fluid underlies a heavier one with a non-horizontal interface between the two of them. This unstable layering causes the lighter fluid to start rising at the point where the interface is highest. Fig. 43 shows a time series of images to illustrate this.

Although van Keken's paper title suggests that the paper is really about thermochemical convection, the part we look here can equally be considered as thermal or chemical convection: all that is necessary is that we describe the fluid's density somehow. We can do that by using an inhomogenous initial temperature field, or an inhomogenous initial composition field. We will use the input file in `cookbooks/van-keken-discontinuous.prm` as input, the central piece of which is as follows (go to the actual input file to see the remainder of the input parameters):

```
subsection Material model
  set Model name = simple
  subsection Simple model
    set Viscosity                     = 1e2
    set Thermal expansion coefficient = 0
    set Density differential for compositional field 1 = -10
  end
end
```

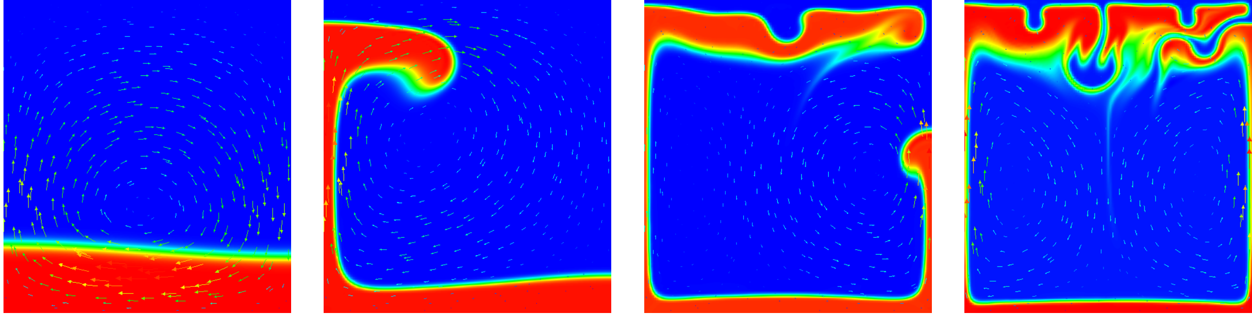


Figure 43: *Van Keken benchmark (using a smoothed out interface, see the main text): Compositional field at times* $t = 0, 300, 900, 1800$.

```
subsection Compositional initial conditions
  set Model name = function
  subsection Function
    set Variable names      = x,z
    set Function constants  = pi=3.14159
    set Function expression = if( (z>0.2+0.02*cos(pi*x/0.9142)) , 0 , 1 )
  end
end
```

The first part of this selects the `simple` material model and sets the thermal expansion to zero (resulting in a density that does not depend on the temperature, making the temperature a passively advected field) and instead makes the density depend on the first compositional field. The second section prescribes that the first compositional field's initial conditions are 0 above a line describes by a cosine and 1 below it. Because the dependence of the density on the compositional field is negative, this means that a lighter fluid underlies a heavier one.

The dynamics of the resulting flow have already been shown in Fig. 43. The measure commonly considered in papers comparing different methods is the root mean square of the velocity, which we can get using the following block in the input file (the actual input file also enables other postprocessors):

```
subsection Postprocess
  set List of postprocessors = velocity statistics
end
```

Using this, we can plot the evolution of the fluid's average velocity over time, as shown in the left panel of Fig. 44. Looking at this graph, we find that both the timing and the height of the first peak is already well converged on a simple 32×32 mesh (5 global refinements) and is very consistent (to better than 1% accuracy) with the results in the van Keken paper.

That said, it is startling that the second peak does not appear to converge despite the fact that the various codes compared in [vKKS+97] show good agreement in this comparison. Tracking down the cause for this proved to be a lesson in benchmark design; in hindsight, it may also explain why van Keken *et al.* stated presciently in their abstract that "*... good agreement is found for the initial rise of the unstable lower layer; however, the timing and location of the later smaller-scale instabilities may differ between methods.*" To understand what is happening here, note that the first peak in these plots corresponds to the plume that rises along the left edge of the domain and whose evolution is primarily determined by the large-scale shape of the initial interface (i.e., the cosine used to describe the initial conditions in the input file). This is a first order deterministic effect, and is obviously resolved already on the coarsest mesh shown used. The second peak corresponds to the plume that rises along the right edge, and its origin along the interface is much harder to trace – its position and the timing when it starts to rise is certainly not obvious from the initial

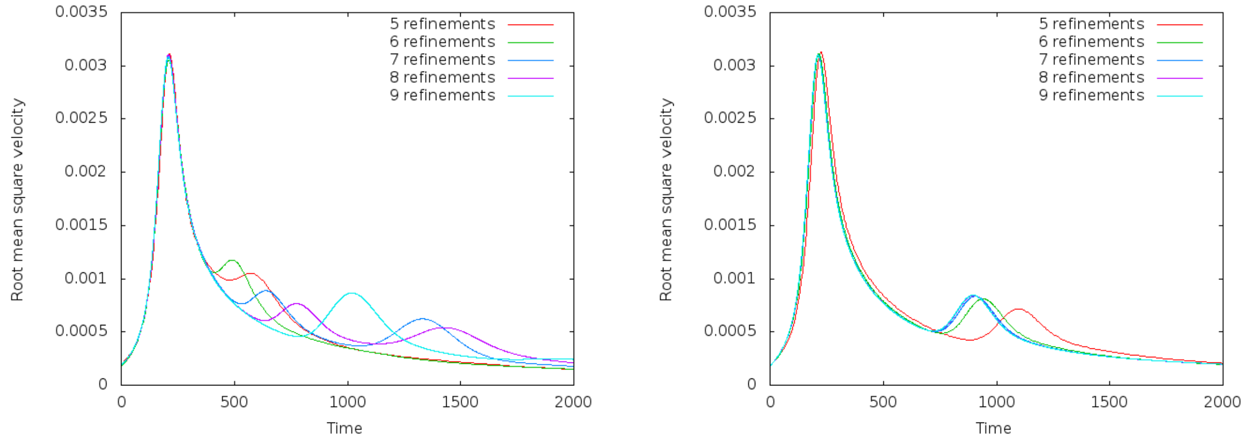


Figure 44: *Van Keken benchmark with discontinuous (left) and smoothed, continuous (right) initial conditions for the compositional field: Evolution of the root mean square velocity* $\left(\frac{1}{|\Omega|}\int_\Omega |\mathbf{u}(\mathbf{x},t)|^2\,dx\right)^{1/2}$ *as a function of time for different numbers of global mesh refinements. 5 global refinements correspond to a* 32×32 *mesh, 9 refinements to a* 512×512 *mesh.*

location of the interface. Now recall that we are using a finite element field using continuous shape functions for the composition that determines the density differences that drive the flow. But this interface is neither aligned with the mesh, nor can a discontinuous function be represented by continuous shape functions to begin with. In other words, we may *input* the initial conditions as a discontinuous functions of zero and one in the parameter file, but the initial conditions used in the program are in fact different: they are the *interpolated* values of this discontinuous function on a finite element mesh. This is shown in Fig. 45. It is obvious that these initial conditions agree on the large scale (the determinant of the first plume), but not in the steps that may (and do, in fact) determine when and where the second plume will rise. The evolution of the resulting compositional field is shown in Fig. 46 and it is obvious that the second, smaller plume starts to rise from a completely different location – no wonder the second peak in the root mean square velocity plot is in a different location and with different height!

The conclusion one can draw from this is that if the outcome of a computational experiment depends so critically on very small details like the steps of an initial condition, then it's probably not a particularly good measure to look at in a benchmark. That said, the benchmark is what it is, and so we should try to come up with ways to look at the benchmark in a way that allows us to reproduce what van Keken *et al.* had agreed upon. To this end, note that the codes compared in that paper use all sorts of different methods, and one can certainly agree on the fact that these methods are not identical on small length scales. One approach to make the setup more mesh-independent is to replace the original discontinuous initial condition with a smoothed out version; of course, we can still not represent it exactly on any given mesh, but we can at least get closer to it than for discontinuous variables. Consequently, let us use the following initial conditions instead (see also the file cookbooks/van-keken-smooth.prm):

```
subsection Compositional initial conditions
  set Model name = function
  subsection Function
    set Variable names      = x,z
    set Function constants  = pi=3.14159
    set Function expression = 0.5*(1+tanh((0.2+0.02*cos(pi*x/0.9142)-z)/0.02))
  end
end
```

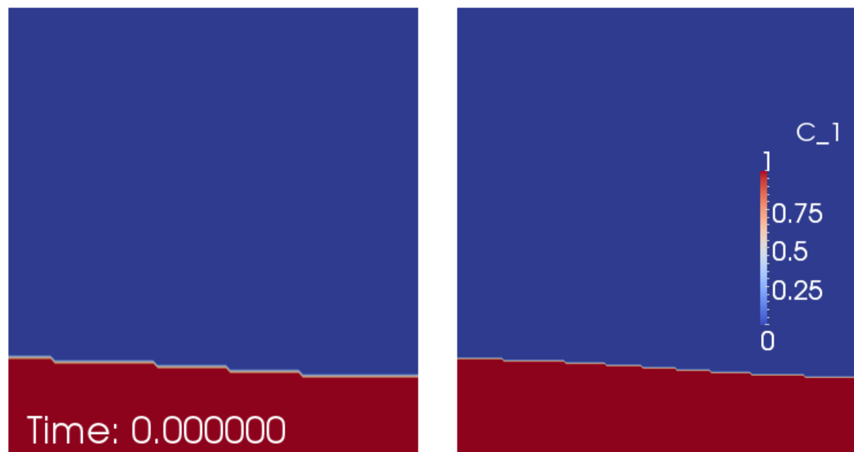


Figure 45: *Van Keken benchmark with discontinuous initial conditions for the compositional field: Initial compositional field interpolated onto a* 32×32 *(left) and* 64×64 *finite element mesh (right).*

This replaces the discontinuous initial conditions with a smoothed out version with a half width of around 0.01. Using this, the root mean square plot now looks as shown in the right panel of Fig. 44. Here, the second peak also converges quickly, as hoped for.

The exact location and height of the two peaks is in good agreement with those given in the paper by van Keken *et al.*, but not exactly where desired (the error is within a couple of per cent for the first peak, and probably better for the second, for both the timing and height of the peaks). This has to do with the fact that they depend on the exact size of the smoothing parameter (the division by 0.02 in the formula for the smoothed initial condition). However, for more exact results, one can choose this half width parameter proportional to the mesh size and thereby get more accurate results. The point of the section was to demonstrate the reason for the lack of convergence.

In this section we extend the van Keken cookbook following up the work previously completed by Cedric Thieulot, Juliane Dannberg, Timo Heister and Wolfgang Bangerth. *This section contributed by Grant Euen, Tahiry Rajaonarison, and Shangxin Liu as part of the Geodynamics and ASPECT class at Virginia Tech.*

As already mentioned above, using a half width parameter proportional to the mesh size allows for more accurate results. We test the effect of the half width size of the smoothed discontinuity by changing the division by 0.02, the smoothing parameter, in the formula for the smoothed initial conditions into values proportional to the mesh size. We use 7 global refinements because the root mean square velocity converges at greater resolution while keeping average runtime around 5 to 25 minutes. These runtimes were produced by the BlueRidge cluster of the Advanced Research Computing (ARC) program at Virginia Tech. BlueRidge is a 408-node Cray CS-300 cluster; each node outfitted with two octa-core Intel Sandy Bridge CPUs and 64 GB of memory. A chart of average runtimes for 5 through 10 global refinements on one node can be seen in Table 4. For 7 global refinements (128×128 mesh size), the size of the mesh is 0.0078 corresponding to a half width parameter of 0.0039. The smooth model allows for much better convergence of the secondary plumes, although they are still more scattered than the primary plumes.

This convergence is due to changing the smoothing parameter, which controls how much of the problem is smoothed over. As the parameter is increased, the smoothed boundary grows and vice versa. As the smoothed boundary shrinks it becomes sharper until the original discontinuous behavior is revealed. As it grows, the two layers eventually become one large, transitioning layer rather than two distinct layers separated by a boundary. These effects can be seen in Fig. 47. The overall effect is that the secondary rise is at different times based on these conditions. In general, as the smoothing parameter is decreased, the smoothed boundary shrinks and the plumes rise more quickly. As it is increased, the boundary grows and the plumes rise more slowly. This trend can be used to force a more accurate convergence from the secondary

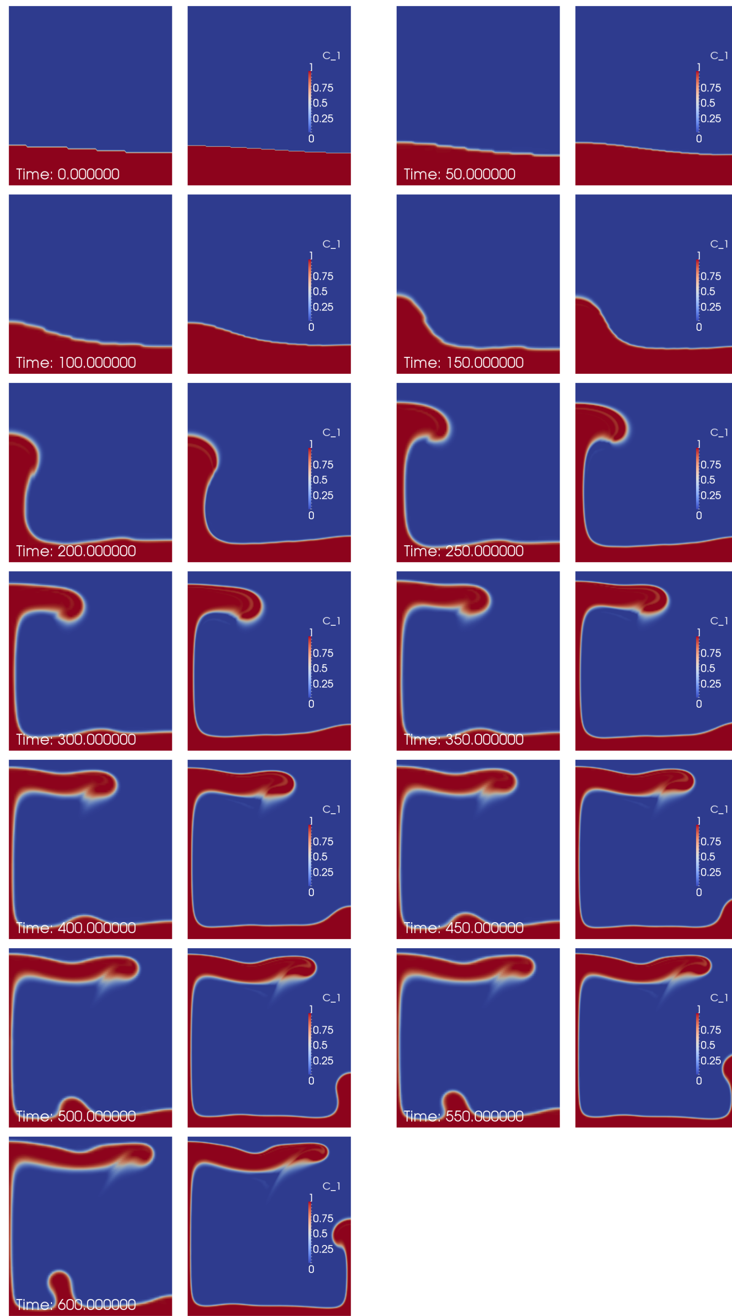


Figure 46: *Van Keken benchmark with discontinuous initial conditions for the compositional field: Evolution of the compositional field over time on a* 32×32 *(first and third column; left to right and top to bottom) and* 64×64 *finite element mesh (second and fourth column).*

Global Refinements	Number of Processors			
	4	8	12	16
5	28.1 seconds	19.8 seconds	19.6 seconds	17.1 seconds
6	3.07 minutes	1.95 minutes	1.49 minutes	1.21 minutes
7	23.33 minutes	13.92 minutes	9.87 minutes	7.33 minutes
8	3.08 hours	1.83 hours	1.30 hours	56.33 minutes
9	1.03 days	15.39 hours	10.44 hours	7.53 hours
10	More than 6 days	More than 6 days	3.39 days	2.56 days

Table 4: *Average runtimes for the van Keken Benchmark with smoothed initial conditions. These times are for the entire computation, a final time step number of 2000. All of these tests were run using ASPECT version 1.3 in release mode, and used different numbers of processors on one node on the BlueRidge cluster of ARC at Virginia Tech.*

plumes.

The evolution in time of the resulting compositional fields (Fig. 48) shows that the first peak converges as the smoothed interface decreases. There is a good agreement for the first peak for all smoothing parameters. As the width of the discontinuity increases, the second peak rises both later and more slowly.

Now let us further add a two-layer viscosity model to the domain. This is done to recreate the two non-isoviscous Rayleigh-Taylor instability cases ("cases 1b and 1c") published in van Keken *et al.* in [vKKS+97]. Let's assume the viscosity value of the upper heavier layer is η_t and the viscosity value of the lower lighter layer is η_b. Based on the initial constant viscosity value 1×10^2 Pa s, we set the viscosity proportion $\frac{\eta_t}{\eta_b} = 0.1, 0.01$, meaning the viscosity of the upper, heavier layer is still 1×10^2 Pa s, but the viscosity of the lower, lighter layer is now either 10 or 1 Pa s, respectively. The viscosity profiles of the discontinuous and smooth models are shown in Fig. 49.

For both benchmark cases, discontinuous and smooth, and both viscosity proportions, 0.1 and 0.01, the results are shown at the end time step number, 2000, in Fig. 50. This was generated using the original input parameter file, running the cases with 8 global refinement steps, and also adding the two-layer viscosity model.

Compared to the results of the constant viscosity throughout the domain, the plumes rise faster when adding the two-layer viscosity model. Also, the larger the viscosity difference is, the earlier the plumes appear and the faster their ascent. To further reveal the effect of the two-layer viscosity model, we also plot the evolution of the fluids' average velocity over time, as shown in Fig. 51.

We can observe that when the two-layer viscosity model is added, there is only one apparent peak for each case. The first peaks of the 0.01 viscosity contrast tests appear earlier and are larger in magnitude than those of 0.1 viscosity contrast tests. There are no secondary plumes and the whole system tends to reach stability after around 500 time steps.

6.4.3 The SolCx Stokes benchmark

The SolCx benchmark is intended to test the accuracy of the solution to a problem that has a large jump in the viscosity along a line through the domain. Such situations are common in geophysics: for example, the viscosity in a cold, subducting slab is much larger than in the surrounding, relatively hot mantle material.

The SolCx benchmark computes the Stokes flow field of a fluid driven by spatial density variations, subject to a spatially variable viscosity. Specifically, the domain is $\Omega = [0,1]^2$, gravity is $\mathbf{g} = (0,-1)^T$ and the density is given by $\rho(\mathbf{x}) = \sin(\pi x_1)\cos(\pi x_2)$; this can be considered a density perturbation to a constant background density. The viscosity is

$$\eta(\mathbf{x}) = \begin{cases} 1 & \text{for } x_1 \leq 0.5, \\ 10^6 & \text{for } x_1 > 0.5. \end{cases}$$

This strongly discontinuous viscosity field yields an almost stagnant flow in the right half of the domain and

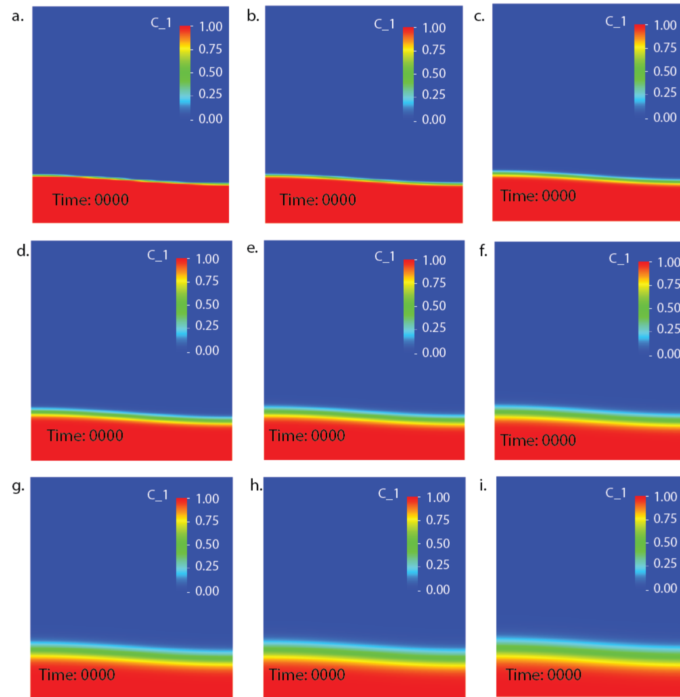


Figure 47: *Van Keken Benchmark using smoothed out interface at 7 global refinements: compositional field at time* $t = 0$ *using smoothing parameter size: a) 0.0039, b) 0.0078, c) 0.0156, d) 0.0234, e) 0.0312, f) 0.0390, g) 0.0468, h) 0.0546, i) 0.0624.*

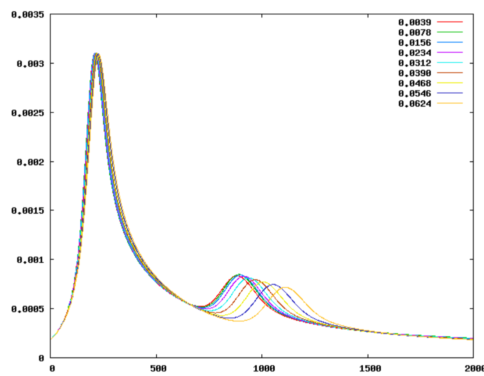


Figure 48: *Van Keken benchmark with smoothed initial conditions for the compositional field using 7 global refinements for different smoothing parameters. Number of the time step is shown on the x-axis, while root mean square velocity is shown on the y-axis.*

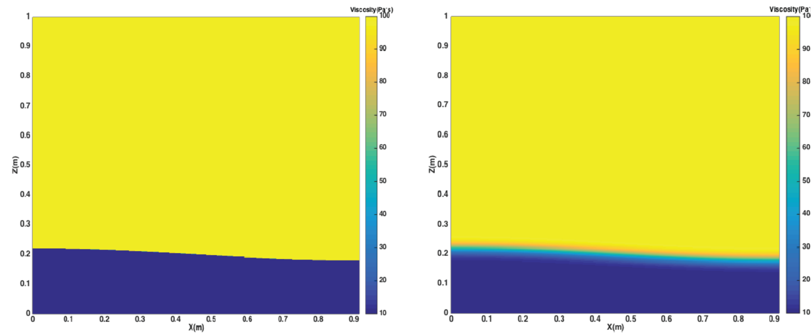


Figure 49: *Van Keken benchmark using layers of different viscosities. The left image is the discontinuous case, while right is the smooth. Both are shown at t=0.*

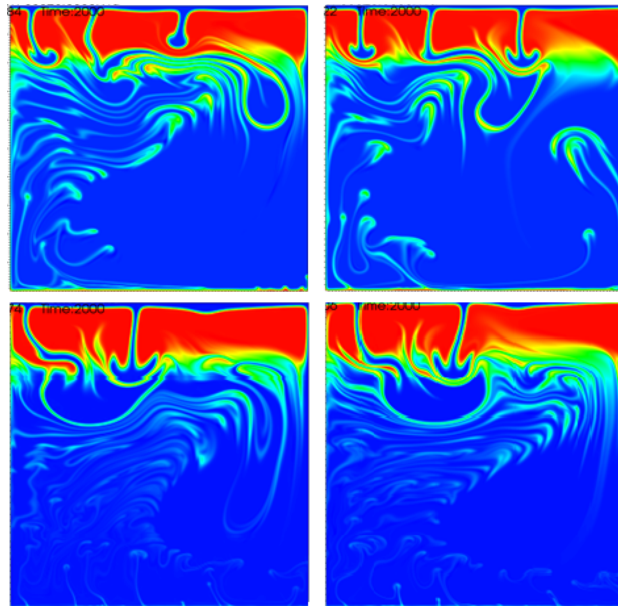


Figure 50: *Van Keken benchmark two-layer viscosity model at final time step number, 2000. These images show layers of different compositions and viscosities. Discontinuous cases are the left images, smooth cases are the right. The upper images are* $\frac{\eta_t}{\eta_b} = 0.1$*, and the lower are* $\frac{\eta_t}{\eta_b} = 0.01$.

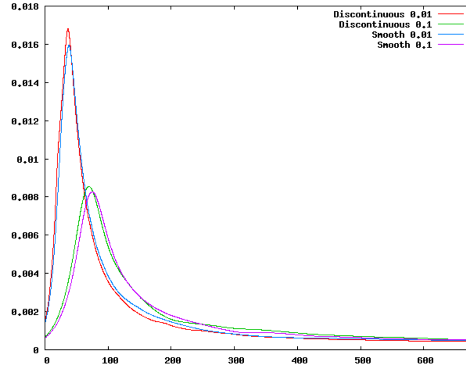


Figure 51: *Van Keken benchmark: Evolution of the root mean square velocity as a function of time for different viscosity contrast proportions (0.1/0.01) for both discontinuous and smooth models.*

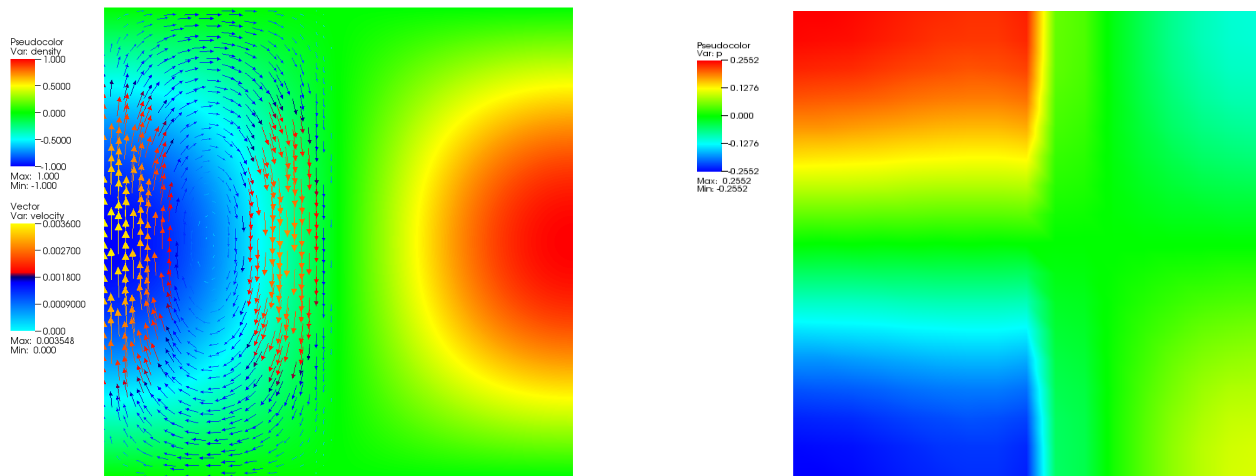


Figure 52: *SolCx Stokes benchmark. Left: The density perturbation field and overlaid to it some velocity vectors. The viscosity is very large in the right hand, leading to a stagnant flow in this region. Right: The pressure on a relatively coarse mesh, showing the internal layer along the line where the viscosity jumps.*

consequently a singularity in the pressure along the interface. Boundary conditions are free slip on all of $\partial\Omega$. The temperature plays no role in this benchmark. The prescribed density field and the resulting velocity field are shown in Fig. 52.

The SolCx benchmark was previously used in [DMGT11, Section 4.1.1] (references to earlier uses of the benchmark are available there) and its analytic solution is given in [Zho96]. ASPECT contains an implementation of this analytic solution taken from the Underworld package (see [MQL+07] and `http://www.underworldproject.org/`, and correcting for the mismatch in sign between the implementation and the description in [DMGT11]).

To run this benchmark, the following input file will do (see the files in `benchmark/solcx/` to rerun the benchmark):

```
set Additional shared libraries            = ./libsolcx.so

############### Global parameters
```

```
set Dimension                              = 2

set Start time                             = 0
set End time                               = 0

set Output directory                       = output

set Pressure normalization                 = volume


############### Parameters describing the model

subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 1
    set Y extent = 1
  end
end


subsection Model settings
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = left, right, bottom, top
  set Zero velocity boundary indicators       =
end


subsection Material model
  set Model name = SolCxMaterial

  subsection SolCx
    set Viscosity jump = 1e6
  end
end


subsection Gravity model
  set Model name = vertical
end


############### Parameters describing the temperature field

subsection Boundary temperature model
  set Model name = box
end


subsection Initial conditions
  set Model name = perturbed box
end
```

```
############### Parameters describing the discretization

subsection Discretization
  set Stokes velocity polynomial degree       = 2
  set Use locally conservative discretization = false
end


subsection Mesh refinement
  set Initial adaptive refinement              = 0
  set Initial global refinement                = 4
end


############### Parameters describing what to do with the solution

subsection Postprocess
  set List of postprocessors = SolCxPostprocessor, visualization
end
```

Since this is the first cookbook in the benchmarking section, let us go through the different parts of this file in more detail:

- The material model and the postprocessor
- The first part consists of parameter setting for overall parameters. Specifically, we set the dimension in which this benchmark runs to two and choose an output directory. Since we are not interested in a time dependent solution, we set the end time equal to the start time, which results in only a single time step being computed.

  The last parameter of this section, `Pressure normalization`, is set in such a way that the pressure is chosen so that its *domain* average is zero, rather than the pressure along the surface, see Section 2.5.
- The next part of the input file describes the setup of the benchmark. Specifically, we have to say how the geometry should look like (a box of size 1×1) and what the velocity boundary conditions shall be (tangential flow all around – the box geometry defines four boundary indicators for the left, right, bottom and top boundaries, see also Section 5.32). This is followed by subsections choosing the material model (where we choose a particular model implemented in ASPECT that describes the spatially variable density and viscosity fields, along with the size of the viscosity jump) and finally the chosen gravity model (a gravity field that is the constant vector $(0, -1)^T$, see Section 5.39).
- The part that follows this describes the boundary and initial values for the temperature. While we are not interested in the evolution of the temperature field in this benchmark, we nevertheless need to set something. The values given here are the minimal set of inputs.
- The second-to-last part sets discretization parameters. Specifically, it determines what kind of Stokes element to choose (see Section 5.29 and the extensive discussion in [KHB12]). We do not adaptively refine the mesh but only do four global refinement steps at the very beginning. This is obviously a parameter worth playing with.
- The final section on postprocessors determines what to do with the solution once computed. Here, we do two things: we ask ASPECT to compute the error in the solution using the setup described in the Duretz et al. paper [DMGT11], and we request that output files for later visualization are generated and placed in the output directory. The functions that compute the error automatically query which kind of material model had been chosen, i.e., they can know whether we are solving the SolCx benchmark or one of the other benchmarks discussed in the following subsections.

Upon running ASPECT with this input file, you will get output of the following kind (obviously with different timings, and details of the output may also change as development of the code continues):

```
aspect/cookbooks> ../aspect solcx.prm
Number of active cells: 256 (on 5 levels)
Number of degrees of freedom: 3,556 (2,178+289+1,089)

*** Timestep 0:  t=0 years
   Solving temperature system... 0 iterations.
   Rebuilding Stokes preconditioner...
   Solving Stokes system... 30+3 iterations.

   Postprocessing:
     Errors u_L1, p_L1, u_L2, p_L2: 1.125997e-06, 2.994143e-03, 1.670009e-06, 9.778441e-03
     Writing graphical output:      output/solution-00000



+---------------------------------------------+------------+------------+
| Total wallclock time elapsed since start    |      1.51s |            |
|                                             |            |            |
| Section                         | no. calls |  wall time | % of total |
+---------------------------------+-----------+------------+------------+
| Assemble Stokes system          |         1 |     0.114s |       7.6% |
| Assemble temperature system     |         1 |     0.284s |        19% |
| Build Stokes preconditioner     |         1 |    0.0935s |       6.2% |
| Build temperature preconditioner|         1 |    0.0043s |      0.29% |
| Solve Stokes system             |         1 |    0.0717s |       4.8% |
| Solve temperature system        |         1 |  0.000753s |      0.05% |
| Postprocessing                  |         1 |     0.627s |        42% |
| Setup dof systems               |         1 |      0.19s |        13% |
+---------------------------------+-----------+------------+------------+
```

One can then visualize the solution in a number of different ways (see Section 4.4), yielding pictures like those shown in Fig. 52. One can also analyze the error as shown in various different ways, for example as a function of the mesh refinement level, the element chosen, etc.; we have done so extensively in [KHB12].

6.4.4 The SolKz Stokes benchmark

The SolKz benchmark is another variation on the same theme as the SolCx benchmark above: it solves a Stokes problem with a spatially variable viscosity but this time the viscosity is not a discontinuous function but grows exponentially with the vertical coordinate so that it's overall variation is again 10^6. The forcing is again chosen by imposing a spatially variable density variation. For details, refer again to [DMGT11].

The following input file, only a small variation of the one in the previous section, solves the benchmark (see `benchmark/solkz/`):

```
# A description of the SolKZ benchmark for which a known solution
# is available. See the manual for more information.

set Additional shared libraries            = ./libsolkz.so

############### Global parameters

set Dimension                              = 2

set Start time                             = 0
```

```
set End time                                = 0

set Output directory                        = output

set Pressure normalization                  = volume


############### Parameters describing the model

subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 1
    set Y extent = 1
  end
end


subsection Model settings
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = left, right, bottom, top
  set Zero velocity boundary indicators       =
end


subsection Material model
  set Model name = SolKzMaterial
end


subsection Gravity model
  set Model name = vertical
end


############### Parameters describing the temperature field

subsection Boundary temperature model
  set Model name = box
end


subsection Initial conditions
  set Model name = perturbed box
end



############### Parameters describing the discretization

subsection Discretization
  set Stokes velocity polynomial degree       = 2
  set Use locally conservative discretization = false
end
```

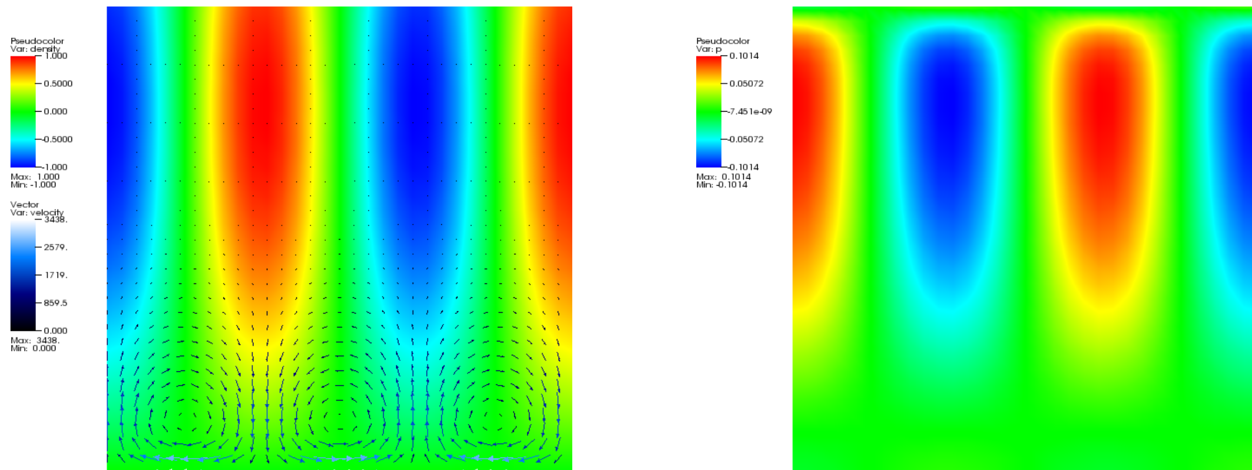


Figure 53: *SolKz Stokes benchmark. Left: The density perturbation field and overlaid to it some velocity vectors. The viscosity grows exponentially in the vertical direction, leading to small velocities at the top despite the large density variations. Right: The pressure.*

```
subsection Mesh refinement
  set Initial adaptive refinement              = 0
  set Initial global refinement                = 4
end


############### Parameters describing what to do with the solution

subsection Postprocess
  set List of postprocessors = SolKzPostprocessor, visualization
end
```

The output when running ASPECT on this parameter file looks similar to the one shown for the SolCx case. The solution when computed with one more level of global refinement is visualized in Fig. 53.

6.4.5 The "inclusion" Stokes benchmark

The "inclusion" benchmark again solves a problem with a discontinuous viscosity, but this time the viscosity is chosen in such a way that the discontinuity is along a circle. This ensures that, unlike in the SolCx benchmark discussed above, the discontinuity in the viscosity never aligns to cell boundaries, leading to much larger difficulties in obtaining an accurate representation of the pressure. Specifically, the almost discontinuous pressure along this interface leads to oscillations in the numerical solution. This can be seen in the visualizations shown in Fig. 54. As before, for details we refer to [DMGT11]. The analytic solution against which we compare is given in [SP03]. An extensive discussion of convergence properties is given in [KHB12].

The benchmark can be run using the parameter files in benchmark/inclusion/. The material model, boundary condition, and postprocessor are defined in benchmark/inclusion/inclusion.cc. Consequently, this code needs to be compiled into a shared lib before you can run the tests.

Link to a general sectio on how you can compil libs for th benchmarks. Revisit thi once we hav the machin- ery in place to choose nonzero boundary con- ditions in

```
############### Global parameters

set Additional shared libraries             = ./libinclusion.so
```

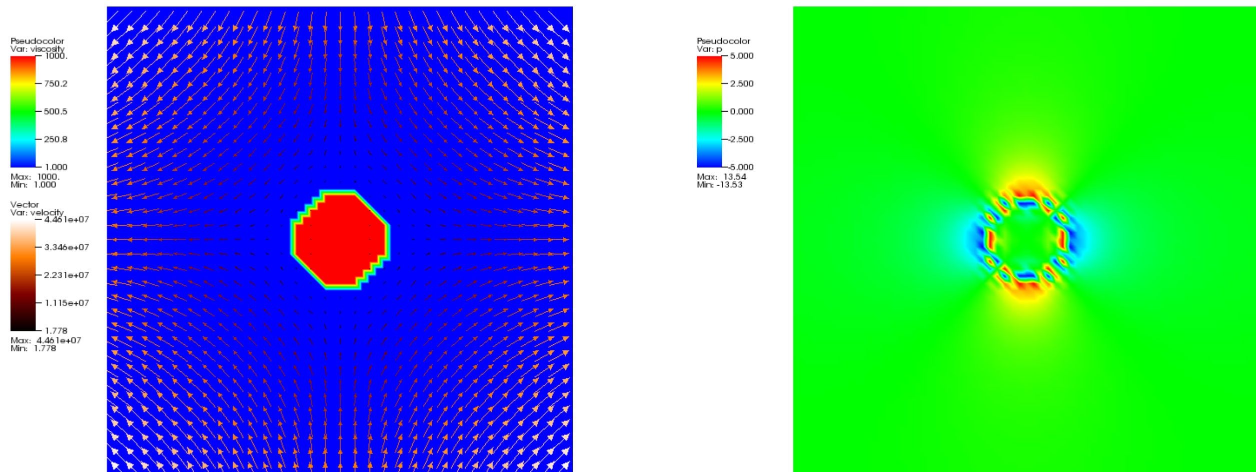


Figure 54: *Inclusion Stokes benchmark. Left: The viscosity field when interpolated onto the mesh (internally, the "exact" viscosity field – large inside a circle, small outside – is used), and overlaid to it some velocity vectors. Right: The pressure with its oscillations along the interface. The oscillations become more localized as the mesh is refined.*

```
set Dimension                              = 2

set Start time                             = 0
set End time                               = 0

set Output directory                       = output

set Pressure normalization                 = volume


############### Parameters describing the model

subsection Geometry model
  set Model name = box

  subsection Box
    set X extent = 2
    set Y extent = 2
  end
end


subsection Model settings
  set Prescribed velocity boundary indicators = left  : InclusionBoundary, \
                                                right : InclusionBoundary, \
                                                bottom: InclusionBoundary, \
                                                top   : InclusionBoundary
  set Tangential velocity boundary indicators =
  set Zero velocity boundary indicators       =
end
```

```
subsection Material model
  set Model name = InclusionMaterial

  subsection Inclusion
    set Viscosity jump = 1e3
  end
end


subsection Gravity model
  set Model name = vertical
end


############### Parameters describing the temperature field

subsection Boundary temperature model
  set Model name = box
end


subsection Initial conditions
  set Model name = perturbed box
end



############### Parameters describing the discretization

subsection Discretization
  set Stokes velocity polynomial degree       = 2
  set Use locally conservative discretization = false
end


subsection Mesh refinement
  set Initial adaptive refinement              = 0
  set Initial global refinement                = 6
end


############### Parameters describing what to do with the solution

subsection Postprocess
  set List of postprocessors = InclusionPostprocessor, visualization
end
```

6.4.6 The Burstedde variable viscosity benchmark

This section was contributed by Iris van Zelst.

This benchmark is intended to test solvers for variable viscosity Stokes problems. It begins with postulating a smooth exact polynomial solution to the Stokes equation for a unit cube, first proposed by [DB14]

and also described by [BSA+13]:

$$\mathbf{u} = \begin{pmatrix} x + x^2 + xy + x^3y \\ y + xy + y^2 + x^2y^2 \\ -2z - 3xz - 3yz - 5x^2yz \end{pmatrix} \tag{38}$$

$$p = xyz + x^3y^3z - \frac{5}{32}. \tag{39}$$

It is then trivial to verify that the velocity field is divergence-free. The constant $-\frac{5}{32}$ has been added to the expression of p to ensure that the volume pressure normalization of ASPECT can be used in this benchmark (in other words, to ensure that the exact pressure has mean value zero and, consequently, can easily be compared with the numerically computed pressure). Following [BSA+13], the viscosity μ is given by the smoothly varying function

$$\mu = \exp\left\{1 - \beta\left[x(1-x) + y(1-y) + z(1-z)\right]\right\}. \tag{40}$$

The maximum of this function is $\mu = e$, for example at $(x, y, z) = (0, 0, 0)$, and the minimum of this function is $\mu = \exp\left(1 - \frac{3\beta}{4}\right)$ at $(x, y, z) = (0.5, 0.5, 0.5)$. The viscosity ratio μ^* is then given by

$$\mu^* = \frac{\exp\left(1 - \frac{3\beta}{4}\right)}{\exp(1)} = \exp\left(\frac{-3\beta}{4}\right). \tag{41}$$

Hence, by varying β between 1 and 20, a difference of up to 7 orders of magnitude viscosity is obtained. β will be one of the parameters that can be selected in the input file that accompanies this benchmark.

The corresponding body force of the Stokes equation can then be computed by inserting this solution into the momentum equation,

$$\nabla p - \nabla \cdot (2\mu\epsilon(\mathbf{u})) = \rho\mathbf{g}. \tag{42}$$

Using equations (38), (39) and (40) in the momentum equation (42), the following expression for the body force $\rho\mathbf{g}$ can be found:

$$\begin{aligned} \rho\mathbf{g} = \begin{pmatrix} yz + 3x^2y^3z \\ xz + 3x^3y^2z \\ xy + x^3y^3 \end{pmatrix} - \mu \begin{pmatrix} 2 + 6xy \\ 2 + 2x^2 + 2y^2 \\ -10yz \end{pmatrix} \\ + (1-2x)\beta\mu \begin{pmatrix} 2 + 4x + 2y + 6x^2y \\ x + y + 2xy^2 + x^3 \\ -3z - 10xyz \end{pmatrix} + (1-2y)\beta\mu \begin{pmatrix} x + y + 2xy^2 + x^3 \\ 2 + 2x + 4y + 4x^2y \\ -3z - 5x^2z \end{pmatrix} \\ + (1-2z)\beta\mu \begin{pmatrix} -3z - 10xyz \\ -3z - 5x^2z \\ -4 - 6x - 6y - 10x^2y \end{pmatrix} \end{aligned} \tag{43}$$

Assuming $\rho = 1$, the above expression translates into an expression for the gravity vector $\mathbf{g}$. This expression for the gravity (even though it is completely unphysical), has consequently been incorporated into the `BursteddeGravity` gravity model that is described in the `benchmarks/burstedde/burstedde.cc` file that accompanies this benchmark.

We will use the input file `benchmark/burstedde/burstedde.prm` as input, which is very similar to the input file `benchmark/inclusion/adaptive.prm` discussed above in Section 6.4.5. The major changes for the 3D polynomial Stokes benchmark are listed below:

```
set Linear solver tolerance                = 1e-12


# Boundary conditions
subsection Model settings
```

```
  set Tangential velocity boundary indicators =
  set Prescribed velocity boundary indicators = left  : BursteddeBoundary, \
                                                right : BursteddeBoundary, \
                                                front : BursteddeBoundary, \
                                                back  : BursteddeBoundary, \
                                                bottom: BursteddeBoundary, \
                                                top   : BursteddeBoundary
end

subsection Material model
  set Model name = BursteddeMaterial
end

subsection Gravity model
  set Model name = BursteddeGravity
end

subsection Burstedde benchmark
   # Viscosity parameter is beta
   set Viscosity parameter             = 20
end

subsection Postprocess
  set List of postprocessors = visualization, velocity statistics, BursteddePostprocessor
end
```

The boundary conditions that are used are simply the velocities from equation (38) prescribed on each boundary. The viscosity parameter in the input file is β. Furthermore, in order to compute the velocity and pressure L_1 and L_2 norm, the postprocessor `BursteddePostprocessor` is used. Please note that the linear solver tolerance is set to a very small value (deviating from the default value), in order to ensure that the solver can solve the system accurately enough to make sure that the iteration error is smaller than the discretization error.

Expected analytical solutions at two locations are summarised in Table 5 and can be deduced from equations (38) and (39). Figure 55 shows that the analytical solution is indeed retrieved by the model.

Table 5: Analytical solutions

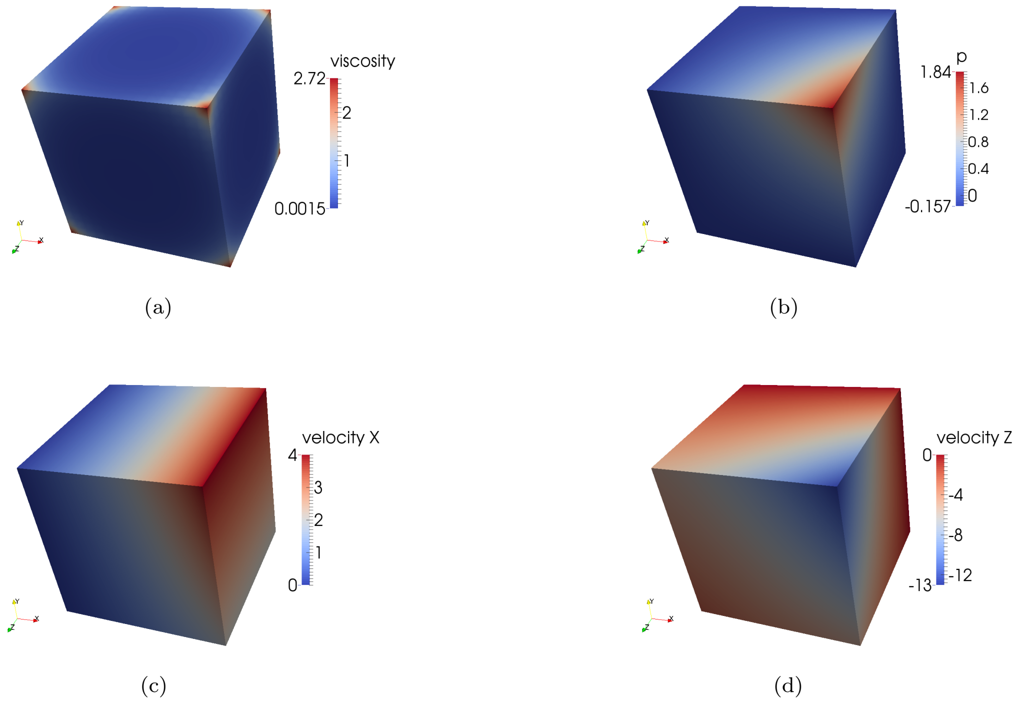
Quantity	$\mathbf{r} = (0,0,0)$	$\mathbf{r} = (1,1,1)$
p	-0.15625	1.84375
$\mathbf{u}$	$(0,0,0)$	$(4,4,-13)$
$\lvert\mathbf{u}\rvert$	0	14.177

The convergence of the numerical error of this benchmark has been analysed by playing with the mesh refinement level in the input file, and results can be found in Figure 56. The velocity shows cubic error convergence, while the pressure shows quadratic convergence in the L_1 and L_2 norms, as one would hope for using Q_2 elements for the velocity and Q_1 elements for the pressure.

6.4.7 The "Stokes' law" benchmark

This section was contributed by Juliane Dannberg.

Stokes' law was derived by George Gabriel Stokes in 1851 and describes the frictional force a sphere with a density different than the surrounding fluid experiences in a laminar flowing viscous medium. A setup for testing this law is a sphere with the radius r falling in a highly viscous fluid with lower density. Due to its higher density the sphere is accelerated by the gravitational force. While the frictional force increases with



(a) (b) (c) (d)

Figure 55: Burstedde benchmark: Results for the 3D polynomial Stokes benchmark, obtained with a resolution of 16×16 elements, with $\beta = 10$.

the velocity of the falling particle, the buoyancy force remains constant. Thus, after some time the forces will be balanced and the settling velocity of the sphere v_s will remain constant:

$$\underbrace{6\pi\,\eta\,r\,v_s}_{\text{frictional force}} = \underbrace{4/3\pi\,r^3\,\Delta\rho\,g,}_{\text{buoyancy force}} \tag{44}$$

where η is the dynamic viscosity of the fluid, $\Delta\rho$ is the density difference between sphere and fluid and g the gravitational acceleration. The resulting settling velocity is then given by

$$v_s = \frac{2}{9}\frac{\Delta\rho\,r^2\,g}{\eta}. \tag{45}$$

Because we do not take into account inertia in our numerical computation, the falling particle will reach the constant settling velocity right after the first timestep.

For the setup of this benchmark, we chose the following parameters:

$$\begin{aligned} r &= 200\,\text{km} \\ \Delta\rho &= 100\,\text{kg/m}^3 \\ \eta &= 10^{22}\,\text{Pa s} \\ g &= 9.81\,\text{m/s}^2. \end{aligned}$$

With these values, the exact value of sinking velocity is $v_s = 8.72 \cdot 10^{-10}$ m/s.

To run this benchmark, we need to set up an input file that describes the situation. In principle, what we need to do is to describe a spherical object with a density that is larger than the surrounding material. There

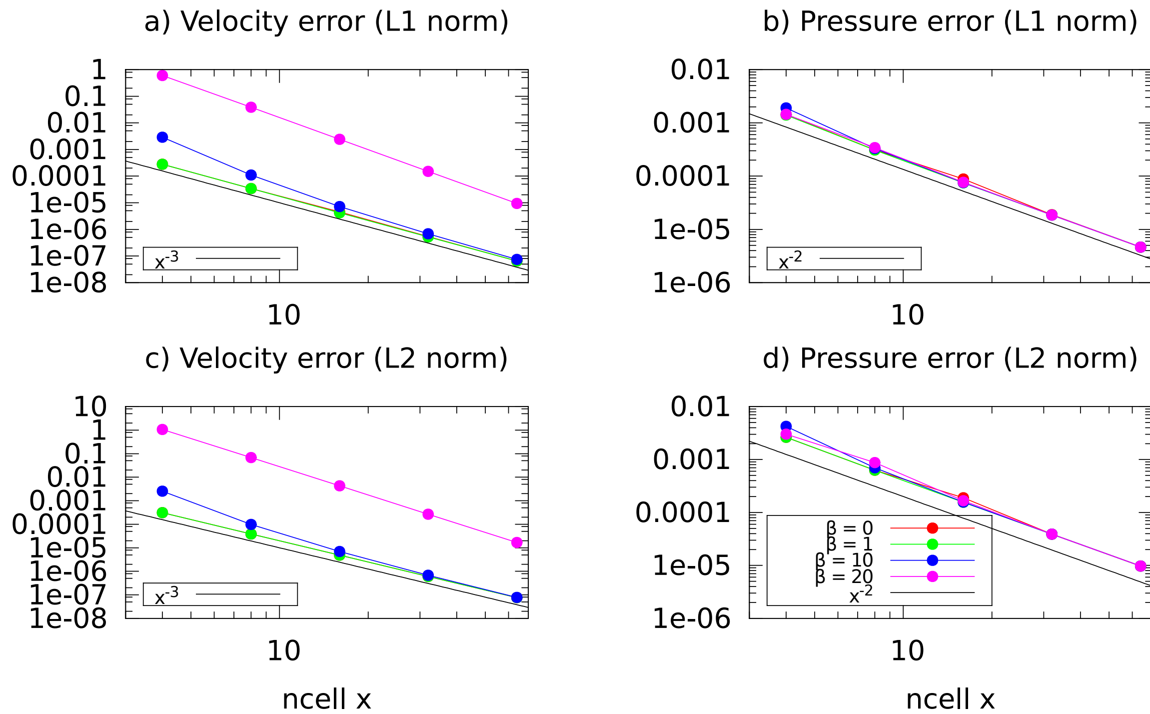


Figure 56: Burstedde benchmark: Error convergence for the 3D polynomial Stokes benchmark.

are multiple ways of doing this. For example, we could simply set the initial temperature of the material in the sphere to a lower value, yielding a higher density with any of the common material models. Or, we could use ASPECT's facilities to advect along what are called "compositional fields" and make the density dependent on these fields.

We will go with the second approach and tell ASPECT to advect a single compositional field. The initial conditions for this field will be zero outside the sphere and one inside. We then need to also tell the material model to increase the density by $\Delta\rho = 100 kg\, m^{-3}$ times the concentration of the compositional field. This can be done, like everything else, from the input file.

All of this setup is then described by the following input file. (You can find the input file to run this cookbook example in cookbooks/stokes.prm. For your first runs you will probably want to reduce the number of mesh refinement steps to make things run more quickly.)

```
############### Global parameters
# We use a 3d setup. Since we are only interested
# in a steady state solution, we set the end time
# equal to the start time to force a single time
# step before the program terminates.

set Dimension                              = 3

set Start time                             = 0
set End time                               = 0
set Use years in output instead of seconds = false

set Output directory                       = output
```

```
############### Parameters describing the model
# The setup is a 3d box with edge length 2890000 in which
# all 6 sides have free slip boundary conditions. Because
# the temperature plays no role in this model we need not
# bother to describe temperature boundary conditions or
# the material parameters that pertain to the temperature.


subsection Geometry model
  set Model name = box

  subsection Box
    set X extent  = 2890000
    set Y extent  = 2890000
    set Z extent  = 2890000
  end
end


subsection Model settings
  set Tangential velocity boundary indicators = left, right, front, back, bottom, top
end


subsection Material model
  set Model name = simple

  subsection Simple model
    set Reference density             = 3300
    set Viscosity                     = 1e22
  end
end


subsection Gravity model
  set Model name = vertical

  subsection Vertical
    set Magnitude = 9.81
  end
end


############### Parameters describing the temperature field
# As above, there is no need to set anything for the
# temperature boundary conditions.

subsection Boundary temperature model
  set Model name = box
end

subsection Initial conditions
  set Model name = function

  subsection Function
```

```
    set Function expression = 0
  end
end

############### Parameters describing the compositional field
# This, however, is the more important part: We need to describe
# the compositional field and its influence on the density
# function. The following blocks say that we want to
# advect a single compositional field and that we give it an
# initial value that is zero outside a sphere of radius
# r=200000m and centered at the point (p,p,p) with
# p=1445000 (which is half the diameter of the box) and one inside.
# The last block re-opens the material model and sets the
# density differential per unit change in compositional field to
# 100.

subsection Compositional fields
  set Number of fields = 1
end

subsection Compositional initial conditions
  set Model name = function

  subsection Function
    set Variable names      = x,y,z
    set Function constants  = r=200000,p=1445000
    set Function expression = if(sqrt((x-p)*(x-p)+(y-p)*(y-p)+(z-p)*(z-p)) > r, 0, 1)
  end
end

subsection Material model
  subsection Simple model
    set Density differential for compositional field 1 = 100
  end
end




############### Parameters describing the discretization
# The following parameters describe how often we want to refine
# the mesh globally and adaptively, what fraction of cells should
# be refined in each adaptive refinement step, and what refinement
# indicator to use when refining the mesh adaptively.

subsection Mesh refinement
  set Initial adaptive refinement        = 4
  set Initial global refinement          = 4
  set Refinement fraction                = 0.2
  set Strategy                           = velocity
end


############### Parameters describing what to do with the solution
# The final section allows us to choose which postprocessors to
# run at the end of each time step. We select to generate graphical
```

```
# output that will consist of the primary variables (velocity, pressure,
# temperature and the compositional fields) as well as the density and
# viscosity. We also select to compute some statistics about the
# velocity field.

subsection Postprocess
  set List of postprocessors = visualization, velocity statistics

  subsection Visualization
    set List of output variables = density, viscosity
  end
end
```

Using this input file, let us try to evaluate the results of the current computations for the settling velocity of the sphere. You can visualize the output in different ways, one of it being ParaView and shown in Fig. 57 (an alternative is to use Visit as described in Section 4.4; 3d images of this simulation using Visit are shown in Fig. 58). Here, Paraview has the advantage that you can calculate the average velocity of the sphere using the following filters:

1. Threshold (Scalars: C_1, Lower Threshold 0.5, Upper Threshold 1),
2. Integrate Variables,
3. Cell Data to Point Data,
4. Calculator (use the formula sqrt(velocity_x^2+ velocity_y^2+velocity_z^2)/Volume).

If you then look at the Calculator object in the Spreadsheet View, you can see the average sinking velocity of the sphere in the column "Result" and compare it to the theoretical value $v_s = 8.72 \cdot 10^{-10}$ m/s. In this case, the numerical result is $8.865 \cdot 10^{-10}$ m/s when you add a few more refinement steps to actually resolve the 3d flow field adequately. The "velocity statistics" postprocessor we have selected above also provides us with a maximal velocity that is on the same order of magnitude. The difference between the analytical and the numerical values can be explained by different at least the following three points: (i) In our case the sphere is viscous and not rigid as assumed in Stokes' initial model, leading to a velocity field that varies inside the sphere rather than being constant. (ii) Stokes' law is derived using an infinite domain but we have a finite box instead. (iii) The mesh may not yet fine enough to provide a fully converges solution. Nevertheless, the fact that we get a result that is accurate to less than 2% is a good indication that ASPECT implements the equations correctly.

6.4.8 Latent heat benchmark

This section was contributed by Juliane Dannberg.

The setup of this benchmark is taken from Schubert, Turcotte and Olson [STO01] (part 1, p. 194) and is illustrated in Fig. 59. It tests whether the latent heat production when material crosses a phase transition is calculated correctly according to the laws of thermodynamics. The material model defines two phases in the model domain with the phase transition approximately in the center. The material flows in from the top due to a prescribed downward velocity, and crosses the phase transition before it leaves the model domain at the bottom. As initial condition, the model uses a uniform temperature field, however, upon the phase change, latent heat is released. This leads to a characteristic temperature profile across the phase transition with a higher temperature in the bottom half of the domain. To compute it, we have to solve equation (3) or its reformulation (5). For steady-state one-dimensional downward flow with vertical velocity v_y, it simplifies to the following:

$$\rho C_p v_y \frac{\partial T}{\partial y} = \rho T \Delta S v_y \frac{\partial X}{\partial y} + \rho C_p \kappa \frac{\partial^2 T}{\partial y^2}.$$

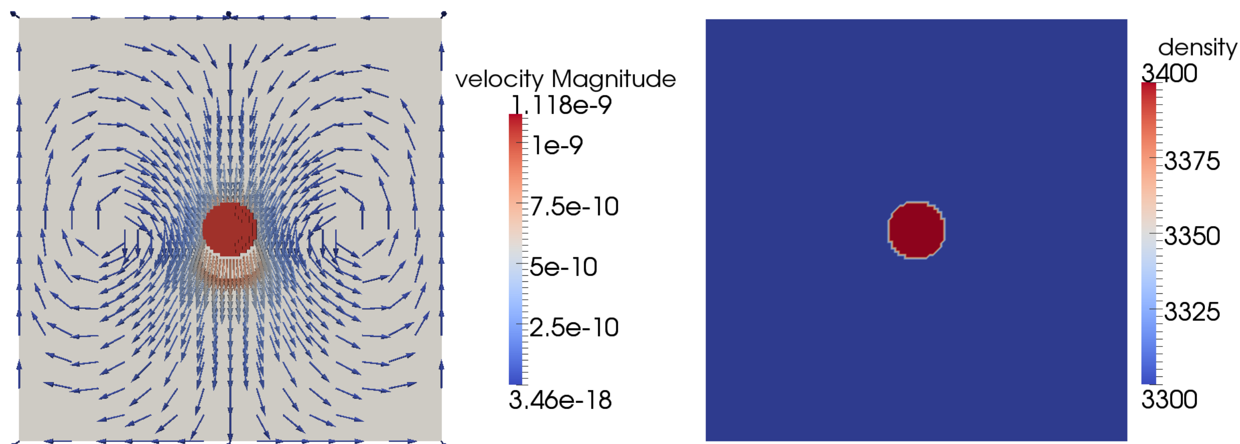


Figure 57: *Stokes benchmark. Both figures show only a 2D slice of the three-dimensional model. Left: The compositional field and overlaid to it some velocity vectors. The composition is 1 inside a sphere with the radius of 200 km and 0 outside of this sphere. As the velocity vectors show, the sphere sinks in the viscous medium. Right: The density distribution of the model. The compositional density contrast of 100 kg/m*^{3} *leads to a higher density inside of the sphere.*

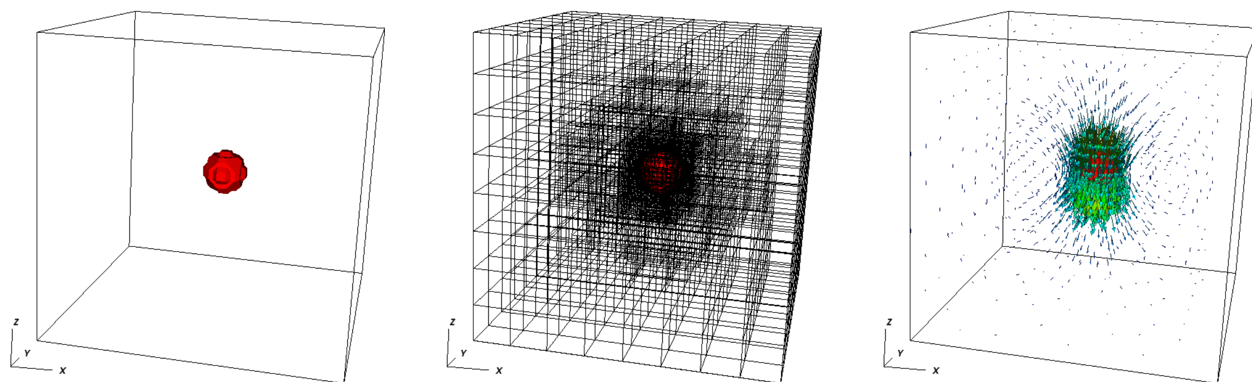


Figure 58: *Stokes benchmark. Three-dimensional views of the compositional field (left), the adaptively refined mesh (center) and the resulting velocity field (right).*

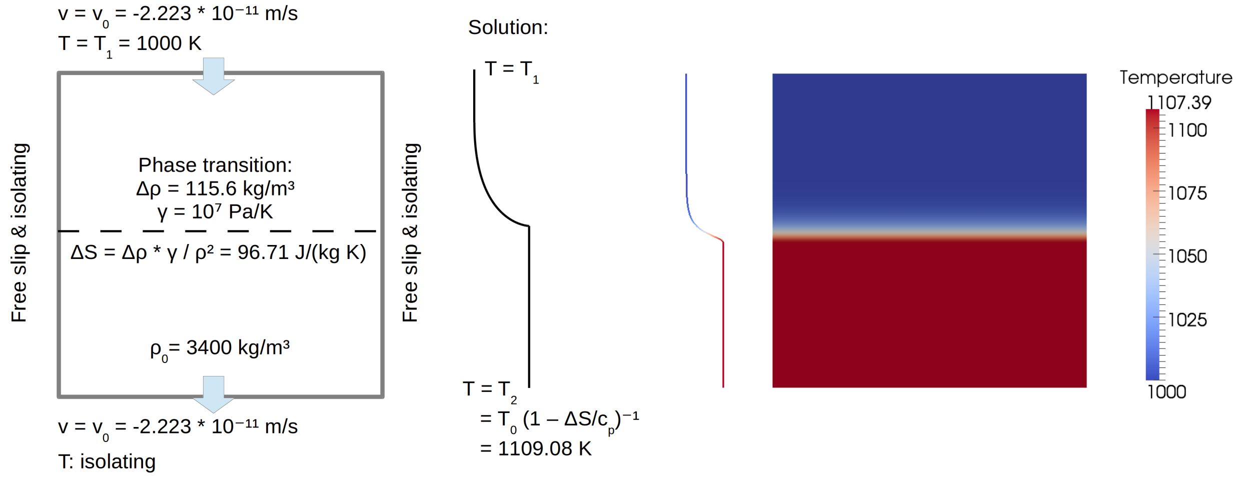


Figure 59: *Latent heat benchmark. Both figures show the 2D box model domain. Left: Setup of the benchmark together with a sketch of the expected temperature profile across the phase transition. The dashed line marks the phase transition. Material flows in with a prescribed temperature and velocity at the top, crosses the phase transition in the center and flows out at the bottom. The predicted bottom temperature is* $T_2 = 1109.08\,K$. *Right: Temperature distribution of the model together with the associated temperature profile across the phase transition. The modelled bottom temperature is* $T_2 = 1107.39\,K$.

Here, $\rho C_p \kappa = k$ with k the thermal conductivity and κ the thermal diffusivity. The first term on the right-hand side of the equation describes the latent heat produced at the phase transition: It is proportional to the temperature T, the entropy change ΔS across the phase transition divided by the specific heat capacity and the derivative of the phase function X. If the velocity is smaller than a critical value, and under the assumption of a discontinuous phase transition (i.e. with a step function as phase function), this latent heating term will be zero everywhere except for the one point y_{tr} where the phase transition takes place. This means, we have a region above the phase transition with only phase 1, and below a certain depth a jump to a region with only phase 2. Inside of these one-phase regions, we can solve the equation above (using the boundary conditions $T = T_1$ for $y \to \infty$ and $T = T_2$ for $y \to -\infty$) and get

$$T(y) = \begin{cases} T_1 + (T_2 - T_1) e^{\frac{v_y (y - y_{tr})}{\kappa}}, & y > y_{tr} \\ T_2, & y < y_{tr} \end{cases}$$

While it is not entirely obvious while this equation for $T(y)$ should be correct (in particular why it should be asymmetric), it is not difficult to verify that it indeed satisfies the equatoin stated above for both $y < y_{tr}$ and $y > y_{tr}$. Furthermore, it indeed satisfies the jump condition we get by evaluating the equation at $y = y_{tr}$. Indeed, the jump condition can be reinterpreted as a balance of heat conduction: We know the amount of heat that is produced at the phase boundary, and as we consider only steady-state, the same amount of heat is conducted upwards from the transition:

$$\underbrace{\rho v_y T \Delta S}_{\text{latent heat release}} = \underbrace{\frac{\kappa}{\rho_0 c_p} \frac{\partial T}{\partial y}|_{y = y_{tr}-} = \frac{v_y}{\rho_0 c_p}(T_2 - T_1)}_{\text{heat conduction}}$$

In contrast to [STO01], we also consider the density change $\Delta\rho$ across the phase transition: While the heat conduction takes place above the transition and the density can be assumed as $\rho = \rho_0 =$ const., the

latent heat is released directly at the phase transition. Thus, we assume an average density $\rho = \rho_0 + 0.5\Delta\rho$ for the left side of the equation. Rearranging this equation gives

$$T_2 = \frac{T_1}{1 - (1 + \frac{\Delta\rho}{2\rho_0})\frac{\Delta S}{c_p}}$$

In addition, we have tested the approach exactly as it is described in [STO01] by setting the entropy change to a specific value and in spite of that using a constant density. However, this is physically inconsistent, as the entropy change is proportional to the density change across the phase transition. With this method, we could reproduce the analytic results from [STO01].

The exact values of the parameters used for this benchmark can be found in Fig. 59. They result in a predicted value of $T_2 = 1109.08\,\text{K}$ for the temperature in the bottom half of the model, and we will demonstrate below that we can match this value in our numerical computations. However, it is not as simple as suggested above. In actual numerical computations, we can not exactly reproduce the behavior of Dirac delta functions as would result from taking the derivative $\frac{\partial X}{\partial y}$ of a discontinuous function $X(y)$. Rather, we have to model $X(y)$ as a function that has a smooth transition from one value to another, over a depth region of a certain width. In the material model plugin we will use below, this depth is an input parameter and we will play with it in the numerical results shown after the input file.

To run this benchmark, we need to set up an input file that describes the situation. In principle, what we need to do is to describe the position and entropy change of the phase transition in addition to the previously outlined boundary and initial conditions. For this purpose, we use the "latent heat" material model that allows us to set the density change $\Delta\rho$ and Clapeyron slope γ (which together determine the entropy change via $\Delta S = \gamma\frac{\Delta\rho}{\rho^2}$) as well as the depth of the phase transition as input parameters.

All of this setup is then described by the input file cookbooks/latent-heat.prm that models flow in a box of 10^6 meters of height and width, and a fixed downward velocity. The following section shows the central part of this file:

```
subsection Material model
  set Model name = latent heat
  subsection Latent heat

    # The change of density across the phase transition. Together with the
    # Clapeyron slope, this is what determines the entropy change.
    set Phase transition density jumps                  = 115.6
    set Corresponding phase for density jump            = 0

    # If the temperature is equal to the phase transition temperature, the
    # phase transition will occur at the phase transition depth. However,
    # if the temperature deviates from this value, the Clapeyron slope
    # determines how much the pressure (and depth) of the phase boundary
    # changes. Here, the phase transition will be in the middle of the box
    # for T=T1.
    set Phase transition depths                         = 500000
    set Phase transition temperatures                   = 1000
    set Phase transition Clapeyron slopes               = 1e7

    # We set the width of the phase transition to 5 km. You may want to
    # change this parameter to see how latent heating depends on the width
    # of the phase transition.
    set Phase transition widths                         = 5000

    set Reference density                               = 3400
    set Reference specific heat                         = 1000
    set Reference temperature                           = 1000
```

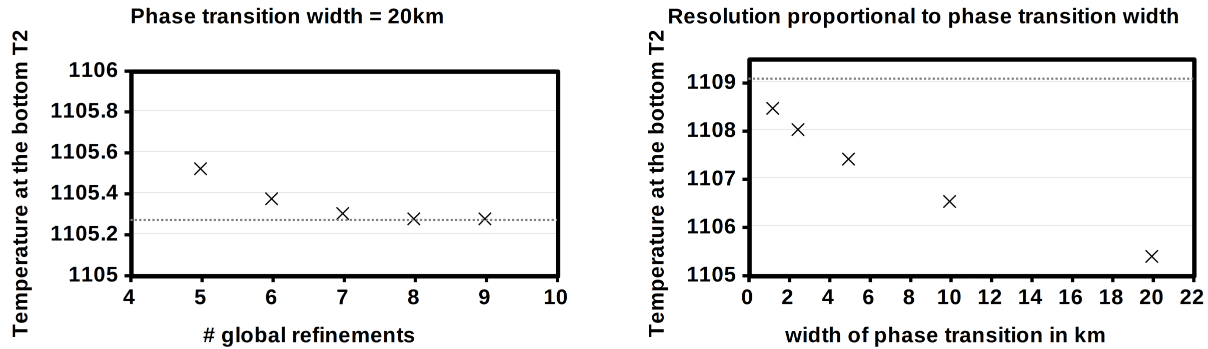


Figure 60: *Results of the latent heat benchmark. Both figures show the modelled temperature T_2 at the bottom of the model domain. Left: T_2 in dependence of resolution using a constant phase transition width of 20 km. With an increasing number of global refinements of the mesh, the bottom temperature converges against a value of $T_2 = 1105.27\,K$. Right: T_2 in dependence of phase transition width. The model resolution is chosen proportional to the phase transition width, starting with 5 global refinements for a width of 20 km. With decreasing phase transition width, T_2 approaches the theoretical value of $1109.08\,K$*

```
    set Thermal conductivity                              = 2.38

    # We set the thermal expansion amd the compressibility to zero, so that
    # all temperature (and density) changes are caused by advection, diffusion
    # and latent heating.
    set Thermal expansion coefficient                     = 0.0
    set Compressibility                                   = 0.0

    # Viscosity is constant.
    set Thermal viscosity exponent                        = 0.0
    set Viscosity                                         = 8.44e21
    set Viscosity prefactors                              = 1.0, 1.0
    set Composition viscosity prefactor                   = 1.0
  end
end
```

The complete input file referenced above also sets the number of mesh refinement steps. For your first runs you will probably want to reduce the number of mesh refinement steps to make things run more quickly. Later on, you might also want to change the phase transition width to look how this influences the result.

Using this input file, let us try to evaluate the results of the current computations. We note that it takes some time for the model to reach a steady state and only then does the bottom temperature reach the theoretical value. Therefore, we use the last output step to compare predicted and computed values. You can visualize the output in different ways, one of it being ParaView and shown in Fig. 59 on the right side (an alternative is to use Visit as described in Section 4.4). In ParaView, you can plot the temperature profile using the filter "Plot Over Line" (Point1: 500000,0,0; Point2: 500000,1000000,0, then go to the "Display" tab and select "T" as only variable in the "Line series" section) or "Calculator" (as seen in Fig. 59). In Fig. 60 (left) we can see that with increasing resolution, the value for the bottom temperature converges to a value of $T_2 = 1105.27\,\mathrm{K}$.

However, this is not what the analytic solution predicted. The reason for this difference is the width of the phase transition with which we smooth out the Dirac delta function that results from differentiating the $X(y)$ we would have liked to use in an ideal world. (In reality, however, for the Earth's mantle we also expect

phase transitions that are distributed over a certain depth range and so the smoothed out approach may not be a bad approximation.) Of course, the results shown above result from an the analytical approach that is only correct if the phase transition is discontinuous and constrained to one specific depth $y = y_{tr}$. Instead, we chose a hyperbolic tangent as our phase function. Moreover, Fig. 60 (right) illustrates what happens to the temperature at the bottom when we vary the width of the phase transition: The smaller the width, the closer the temperature gets to the predicted value of $T_2 = 1109.08\,\mathrm{K}$, demonstrating that we converge to the correct solution.

6.4.9 The 2D cylindrical shell benchmarks by Davies et al.

This section was contributed by William Durkin and Wolfgang Bangerth.

All of the benchmarks presented so far take place in a Cartesian domain. Davies et al. describe a benchmark (in a paper that is currently still being written) for a 2D spherical Earth that is nondimensionalized such that

$$\begin{aligned} r_{\text{min}} &= 1.22 & T|_{r_{min}} &= 1 \\ r_{\text{max}} &= 2.22 & T|_{r_{max}} &= 0 \end{aligned}$$

The benchmark is run for a series of approximations (Boussinesq, Extended Boussinesq, Truncated Anelastic Liquid, and Anelastic Liquid), and temperature, velocity, and heat flux calculations are compared with the results of other mantle modeling programs. ASPECT will output all of these values directly except for the Nusselt number, which we must calculate ourselves from the heat fluxes that ASPECT can compute. The Nusselt number of the top and bottom surfaces, Nu_T and Nu_B, respectively, are defined by the authors of the benchmarks as

$$Nu_T = \frac{\ln(f)}{2\pi r_{\text{max}}(1-f)} \int_0^{2\pi} \frac{\partial T}{\partial r}\, d\theta \tag{46}$$

and

$$Nu_B = \frac{f\ln(f)}{2\pi r_{\text{min}}(1-f)} \int_0^{2\pi} \frac{\partial T}{\partial r}\, d\theta$$

where f is the ratio $\frac{r_{\text{min}}}{r_{\text{max}}}$.

We can put this in terms of heat flux

$$q_r = -k\frac{\partial T}{\partial r}$$

through the inner and outer surfaces, where q_r is heat flux in the radial direction. Let Q be the total heat that flows throug a surface,

$$Q = \int_0^{2\pi} q_r\, d\theta,$$

then (46) becomes

$$Nu_T = \frac{-Q_T \ln(f)}{2\pi r_{\text{max}}(1-f)k}$$

and similarly

$$Nu_B = \frac{-Q_B f \ln(f)}{2\pi r_{\text{min}}(1-f)k}.$$

Q_T and Q_B are heat fluxes that ASPECT can readily compute through the `heat flux statistics` postprocessor (see Section 5.88). For further details on the nondimensionalization and equations used for each approximation, refer to Davies et al.

The series of benchmarks is then defined by a number of cases relating to the exact equations chosen to model the fluid. We will discuss these in the following.

Case 1.1: BA_Ra104_Iso_ZS This case is run with the following settings:

- Boussinesq Approximation
- Boundary Condition: Zero-Slip
- Rayleigh Number = 10^4
- Initial Conditions: $D = 0, O = 4$
- $\eta(T) = 1$

where D and O refer to the degree and order of a spherical harmonic that describes the initial temperature. While the initial conditions matter, what is important here though is that the system evolve to four convective cells since we are only interested in the long term, steady state behavior.

The model is relatively straightforward to set up, basing the input file on that discussed in Section 6.3.1. The full input file can be found at `benchmark/davies_et_al/case-1.1.prm`, with the interesting parts excerpted as follows:

```
############### Parameters describing the model

subsection Geometry model
  set Model name = spherical shell
  subsection Spherical shell
    set Inner radius  = 1.22
    set Opening angle = 360
    set Outer radius  = 2.22
  end
end

# [...]

subsection Material model
  set Model name = simple
  subsection Simple model
    set Reference density             = 1
    set Reference specific heat       = 1.
    set Reference temperature         = 0
    set Thermal conductivity          = 1
    set Thermal expansion coefficient = 1e-6
    set Viscosity                     = 1
  end
end


############### Parameters describing the temperature field
# Angular mode is set to 4 in order to match the number of
# convective cells reported by Davies et al.

subsection Initial conditions
  set Model name = spherical hexagonal perturbation
  subsection Spherical hexagonal perturbation
    set Angular mode          = 4
    set Rotation offset       = 0
  end
end
```

```
############### Prescribe the Rayleigh number as g*alpha
# Here, Ra = 10^4 and alpha was chosen as 10^-6 above.
subsection Gravity model
  set Model name = radial constant
  subsection Radial constant
    set Magnitude =  1e10
  end
end


# [...]
```

We use the same trick here as in Section 6.2.1 to produce a model in which the density $\rho(T)$ in the temperature equation (3) is almost constant (namely, by choosing a very small thermal expansion coefficient) as required by the benchmark, and instead prescribe the desired Rayleigh number by choosing a correspondingly large gravity.

Results for this and the other cases are shown below.

Case 2.1: BA_Ra104_Iso_FS Case 2.1 uses the following setup, differing only in the boundary conditions:

- Boussinesq Approximation
- Boundary Condition: Free-Slip
- Rayleigh Number = 10^4
- Initial Conditions: $D = 0, O = 4$
- $\eta(T) = 1$

As a consequence of the free slip boundary conditions, any solid body rotation of the entire system satisfies the Stokes equations with their boundary conditions. In other words, the solution of the problem is not unique: given a solution, adding a solid body rotation yields another solution. We select arbitrarily the one that has no net rotation (see Section 5.87). The section in the input file that is relevant is then as follows (the full input file resides at benchmark/davies_et_al/case-2.1.prm):

```
subsection Model settings
  set Remove nullspace                          = net rotation

  set Fixed temperature boundary indicators     = 0,1
  set Prescribed velocity boundary indicators =
  set Tangential velocity boundary indicators = 0,1
  set Zero velocity boundary indicators         =
  set Include adiabatic heating                 = false
  set Include shear heating                     = false
end
```

Again, results are shown below.

Case 2.2: BA_Ra105_Iso_FS Case 2.2 is described as follows:

- Boussinesq Approximation
- Boundary Condition: Free-Slip
- Rayleigh Number = 10^5

- Initial Conditions: Final conditions of case 2.1 (BA_Ra104_Iso_FS)
- $\eta(T) = 1$

In other words, we have an increased Rayleigh number and begin with the final steady state of case 2.1. To start the model where case 2.1 left off, the input file of case 2.1, benchmark/davies_et_al/case-2.1.prm, instructs ASPECT to checkpoint itself every few time steps (see Section 4.5). If case 2.2 uses the same output directory, we can then resume the computations from this checkpoint with an input file that prescribes a different Rayleigh number and a later input time:

```
############### Global parameters
# Case 2.2 begins with the final steady state solution of Case 2.1
# "Resume computation" must be set to true, and "Output directory" must
# point to the folder that contains the results of Case 2.1.

set CFL number                              = 10

set End time                                = 3
set Output directory                        = output

set Resume computation                      = true
```

We increase the Rayleigh number to 10^5 by increasing the magnitude of gravity in the input file. The full script for case 2.2 is located in benchmark/davies_et_al/case-2.2.prm

Case 2.3: BA_Ra103_vv_FS Case 2.3 is a variation on the previous one:

- Boussinesq Approximation
- Boundary Condition: Free-Slip
- Rayleigh Number = 10^3
- Initial Conditions: Final conditions of case 2.1 (BA_Ra104_Iso_FS)
- $\eta(T) = 1000^{-T}$

The Rayleigh number is smaller here (and is selected using the gravity parameter in the input file, as before), but the more important change is that the viscosity is now a function of temperature. At the time of writing, there is no material model that would implement such a viscosity, so we create a plugin that does so for us (see Sections 7 and 7.2 in general, and Section 7.3.1 for material models in particular). The code for it is located in benchmarks/davies_et_al/case-2.3-plugin/VoT.cc (where "VoT" is short for "viscosity as a function of temperature") and is essentially a copy of the simpler material model. The primary change compared to the simpler material model is the line about the viscosity in the following function:

```
template <int dim>
void
VoT<dim>::
evaluate(const typename Interface<dim>::MaterialModelInputs &in,
         typename Interface<dim>::MaterialModelOutputs &out) const
{
  for (unsigned int i=0; i<in.position.size(); ++i)
    {
      out.viscosities[i] = eta*std::pow(1000,(-in.temperature[i]));
      out.densities[i] = reference_rho * (1.0 - thermal_alpha * (in.temperature[i] - reference_T));
      out.thermal_expansion_coefficients[i] = thermal_alpha;
      out.specific_heat[i] = reference_specific_heat;
```

```
      out.thermal_conductivities[i] = k_value;
      out.compressibilities[i] = 0.0;
    }
}
```

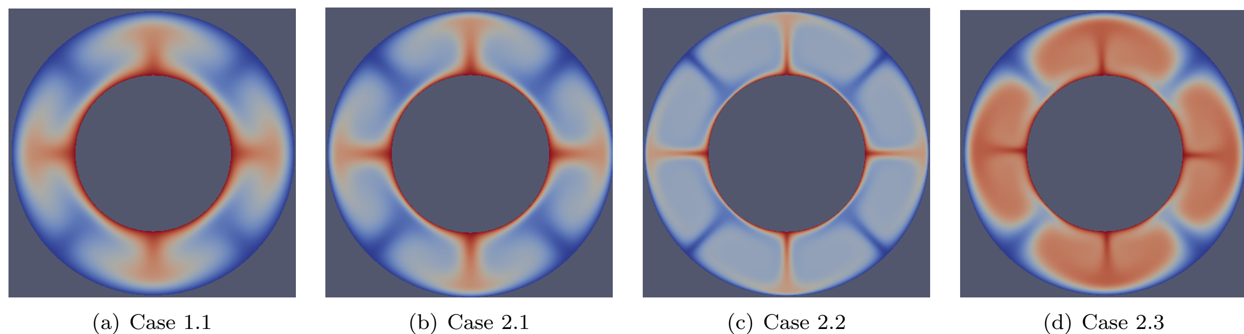
Using the method described in Sections 6.4.1 and 7.2, and the files in the benchmarks/davies_et_al/case-2.3-plugin, we can compile our new material model into a shared library that we can then reference from the input file. The complete input file for case 2.3 is located in benchmark/davies_et_al/case-2.3.prm and contains among others the following parts:

```
set Additional shared libraries            = ./case-2.3-plugin/libVoT.so

subsection Material model
  set Model name = VoT

  subsection VoT model
    set Reference density              = 1
    set Reference specific heat        = 1.
    set Reference temperature          = 0
    set Thermal conductivity           = 1
    set Thermal expansion coefficient = 1e-5
    set Viscosity                      = 1
  end
end
```

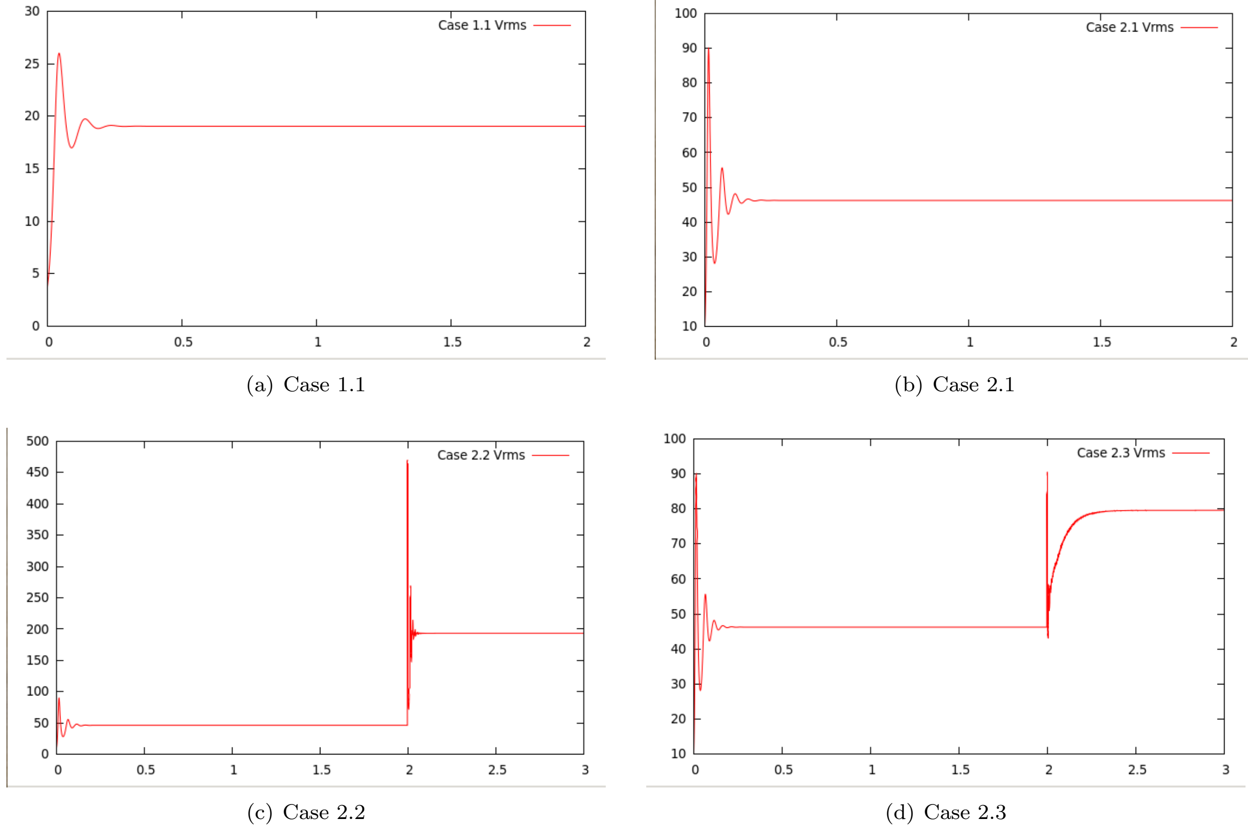
Results In the following, let us discuss some of the results of the benchmark setups discussed above. First, the final steady state temperature fields are shown in Fig. 61. It is immediately obvious how the different Rayleigh numbers affect the width of the plumes. If one imagines a setup with constant gravity, constant inner and outer temperatures and constant thermal expension coefficient (this is not how we describe it in the input files, but we could have done so and it is closer to how we intuit about fluids than adjusting the gravity), then the Rayleigh number is inversely proportional to the viscosity – and it is immediately clear that larger Rayleigh numbers (corresponding to lower viscosities) then lead to thinner plumes. This is nicely reflected in the visualizations.



(a) Case 1.1 (b) Case 2.1 (c) Case 2.2 (d) Case 2.3

Figure 61: Davies et al. benchmarks: Final steady state temperature fields for the 2D cylindrical benchmark cases.

Secondly, Fig. 62 shows the root mean square velocity as a function of time for the various cases. It is obvious that they all converge to steady state solutions. However, there is an initial transient stage and, in cases 2.2 and 2.3, a sudden jolt to the system at the time where we switch from the model used to compute up to time $t = 2$ to the different models used after that.



(a) Case 1.1

(b) Case 2.1

(c) Case 2.2

(d) Case 2.3

Figure 62: Davies et al. benchmarks: V_{rms} for 2D Cylindrical Cases. Large jumps occur when transitioning from case 2.1 to cases 2.2 and 2.3 due to the instantaneous change of parameter settings.

These runs also produce quantitative data that will be published along with the concise descriptions of the benchmarks and a comparison with other codes. In particular, some of the criteria listed above to judge the accuracy of results are listed in Table 6.[33]

6.4.10 The Crameri et al. benchmarks

This section was contributed by Ian Rose.

[33]The input files available in the `benchmark/davies_et_al` directory use 5 global refinements in order to provide cases that can be run without excessive trouble on a normal computer. However, this is not enough to achieve reasonable accuracy and both the data shown below and the data submitted to the benchmarking effort uses 7 global refinement steps, corresponding to a mesh with 1536 cells in tangential and 128 cells in radial direction. Computing on such meshes is not cheap, as it leads to a problem size of more than 2.5 million unknowns. It is best done using a parallel computation.

Case	$\langle T \rangle$	Nu_T	Nu_B	V_{rms}
1.1	0.403	2.464	2.468	19.053
2.1	0.382	4.7000	4.706	46.244
2.2	0.382	9.548	9.584	193.371
2.3	0.582	5.102	5.121	79.632

Table 6: *Davies et al. benchmarks: Numerical results for some of the output quantities required by the benchmarks and the various cases considered.*

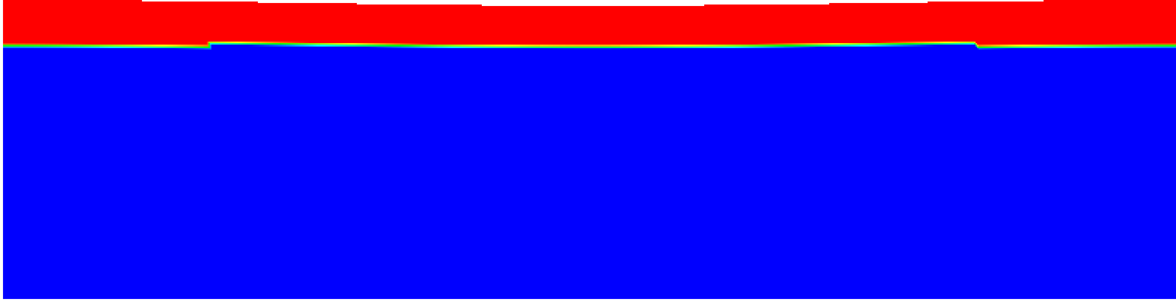


Figure 63: *Setup for the topography relaxation benchmark. The box is* 2800 *km wide and* 700 *km high, with a* 100 *km lid on top. The lid has a viscosity of* 10^{23} $Pa\,s$*, while the mantle has a viscosity of* 10^{21} $Pa\,s$*. The sides are free slip, the bottom is no slip, and the top is a free surface. Both the lid and the mantle have a density of* $3300\,kg/m^3$*, and gravity is* $10\,m/s^2$*. There is a* $7\,km$ *sinusoidal initial topography on the free surface.*

This section follows the two free surface benchmarks described by Crameri et al. [CSG+12].

Case 1: Relaxation of topography The first benchmark involves a high viscosity lid sitting on top of a lower viscosity mantle. There is an initial sinusoidal topography which is then allowed to relax. This benchmark has a semi-analytical solution (which is exact for infinitesimally small topography). Details for the benchmark setup are in Figure 63.

The complete parameter file for this benchmark can be found in benchmarks/crameri_et_al/case_1/crameri_benchmark_1.prm, the most relevant parts of which are excerpted here:

```
set CFL number                              = 0.01

set Additional shared libraries = ./libcrameri_benchmark_1.so

subsection Geometry model
  set Model name = rebound box
  subsection Rebound Box
    set Order = 1
    set Amplitude = 7.e3
  end
  subsection Box
    set X extent = 28.e5
    set Y extent = 7.e5
    set X repetitions = 300
    set Y repetitions = 75
  end
end
```

In particular, this benchmark uses a custom geometry model to set the initial geometry. This geometry model, called "ReboundBox", is based on the Box geometry model. It generates a domain in using the same parameters as Box, but then displaces all the nodes vertically with a sinusoidal perturbation, where the magnitude and order of that perturbation are specified in the ReboundBox subsection.

The characteristic timescales of topography relaxation are significantly smaller than those of mantle convection. Taking timesteps larger than this relaxation timescale tends to cause sloshing instabilities, which are described further in Section 2.11. Some sort of stabilization is required to take large timesteps. In this benchmark, however, we are interested in the relaxation timescale, so we are free to take very small

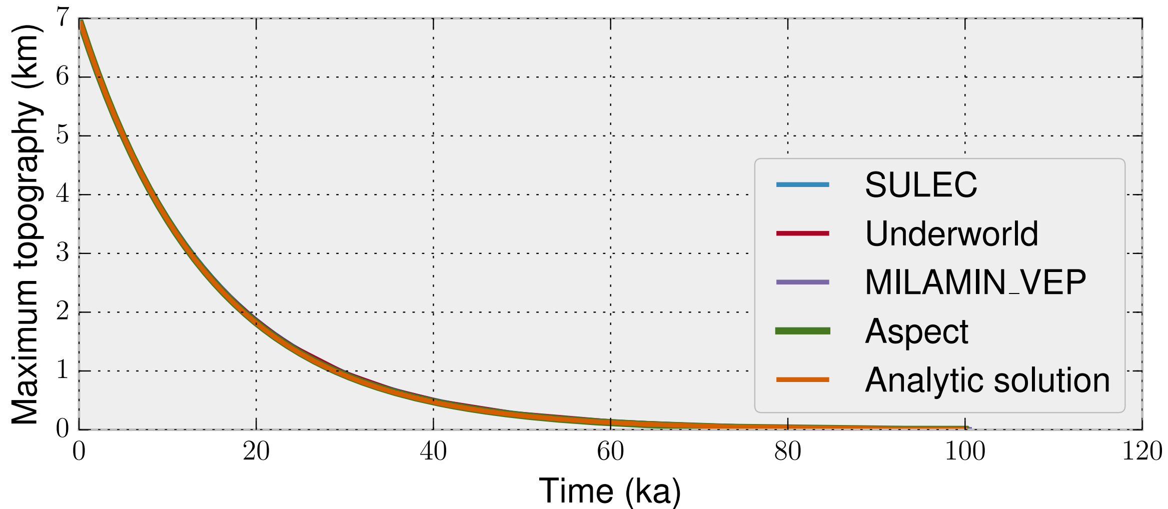


Figure 64: *Results for the topography relaxation benchmark, showing maximum topography versus time. Over about* 100 *ka the topography completely disappears. The results of four free surface codes, as well as the semi-analytic solution, are nearly identical.*

timesteps (in this case, 0.01 times the CFL number). As can be seen in Figure 64, the results of all the codes which are included in this comparison are basically indistinguishable.

Case 2: Dynamic topography Case two is more complicated. Unlike the case one, it occurs over mantle convection timescales. In this benchmark there is the same high viscosity lid over a lower viscosity mantle. However, now there is a blob of buoyant material rising in the center of the domain, causing dynamic topography at the surface. The details for the setup are in the caption of Figure 65.

Case two requires higher resolution and longer time integrations than case one. The benchmark is over 20 million years and builds dynamic topography of ~ 800 meters.

Again, we excerpt the most relevant parts of the parameter file for this benchmark, with the full thing available in benchmarks/crameri_et_al/case_2/crameri_benchmark_2.prm. Here we use the "Multicomponent" material model, which allows us to easily set up a number of compositional fields with different material properties. The first compositional field corresponds to background mantle, the second corresponds to the rising blob, and the third corresponds to the viscous lid.

Furthermore, the results of this benchmark are sensitive to the mesh refenement and timestepping parameters. Here we have nine refinement levels, and refine according to density and the compositional fields.

```
set CFL number                   = 0.1

subsection Material model
  set Model name = multicomponent
  subsection Multicomponent
    set Densities = 3300, 3200, 3300
    set Viscosities = 1.e21, 1.e20, 1.e23
    set Viscosity averaging scheme = harmonic
  end
end

subsection Mesh refinement
```

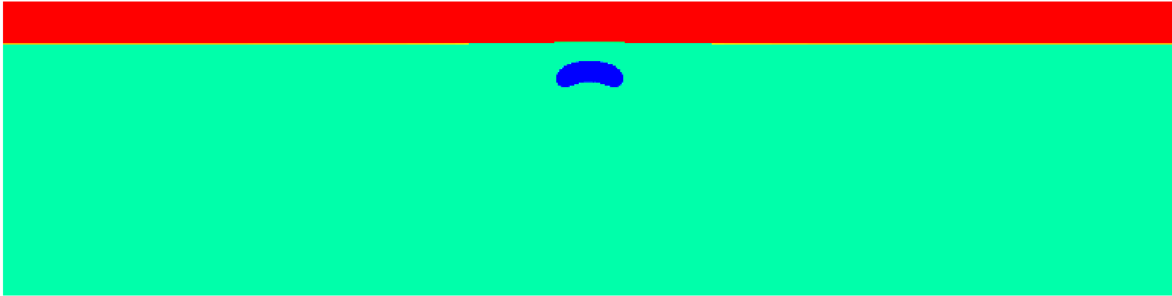


Figure 65: *Setup for the dynamic topography benchmark. Again, the domain is* 2800 *km wide and* 700 *km high. A* 100 *km thick lid with viscosity* 10^{23} *overlies a mantle with viscosity* 10^{21}*. Both the lid and the mantle have a density of* $3300\,kg/m^3$*. A blob with diameter* 100 *km lies* 300 *km from the bottom of the domain. The blob has a density of* $3200kg/m^3$ *and a viscosity of* 10^{20} *Pa s.*

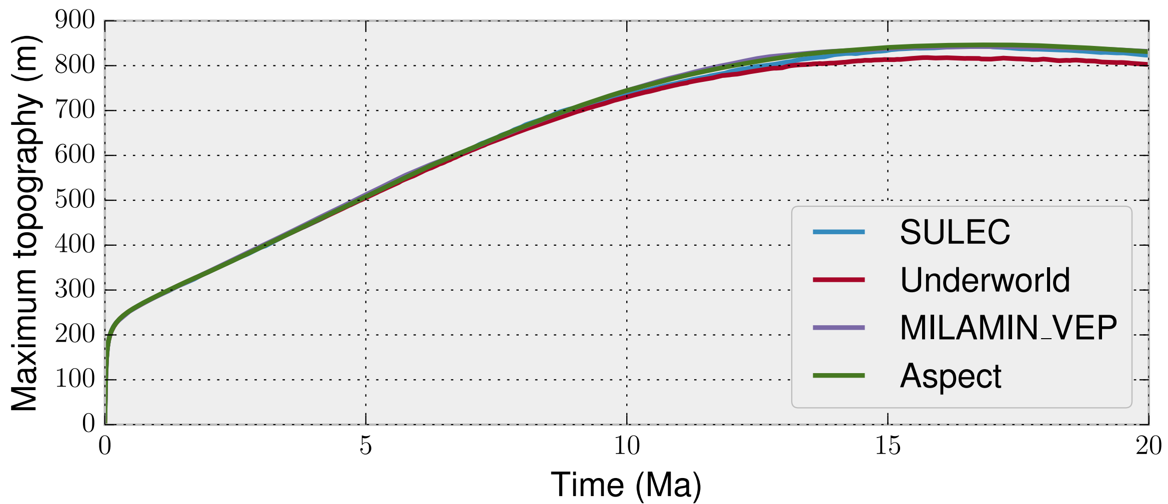


Figure 66: *Evolution of topography for the dynamic topography benchmark. The maximum topography is shown as a function of time, for* ASPECT *as well as for several other codes participating in the benchmark. This benchmark shows considerably more scatter between the codes.*

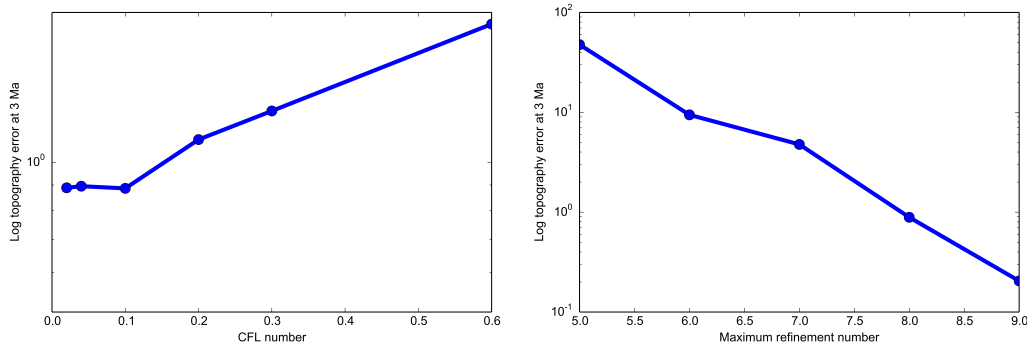


Figure 67: *Convergence for case two. Left: Logarithm of the error with decreasing CFL number. As the CFL number decreases, the error gets smaller. However, once it reaches a value of* ~ 0.1*, there stops being much improvement in accuracy. Right: Logarithm of the error with increasing maximum mesh resolution. As the resolution increases, so does the accuracy.*

```
  set Additional refinement times        =
  set Initial adaptive refinement        = 4
  set Initial global refinement          = 5
  set Refinement fraction                = 0.3
  set Coarsening fraction                = 0.0
  set Strategy                           = density,composition
  set Refinement criteria merge operation = plus
  set Time steps between mesh refinement = 5
end
```

Unlike the first benchmark, for case two there is no (semi) analytical solution to compare against. Furthermore, the time integration for this benchmark is much longer, allowing for errors to accumulate. As such, there is considerably more scatter between the participating codes. ASPECT does, however, fall within the range of the other results, and the curve is somewhat less wiggly. The results for maximum topography versus time are shown in 66

The precise values for topography at a given time are quite dependent on the resolution and timestepping parameters. Following [CSG+12] we investigate the convergence of the maximum topography at 3 Ma as a function of CFL number and mesh resolution. The results are shown in figure 67.

We find that at 3 Ma ASPECT converges to a maximum topography of ∼396 meters. This is slightly different from what MILAMIN_VEP reported as its convergent value in [CSG+12], but still well within the range of variation of the codes. Additionally, we note that ASPECT is able to achieve good results with relatively less mesh resolution due to the ability to adaptively refine in the regions of interest (namely, the blob and the high viscosity lid).

Accuracy improves roughly linearly with decreasing CFL number, though stops improving at CFL ~ 0.1. Accuracy also improves with increasing mesh resolution, though its convergence order does not seem to be excellent. It is possible that other mesh refinement parameters than we tried in this benchmark could improve the convergence. The primary challenge in accuracy is limiting numerical diffusion of the rising blob. If the blob becomes too diffuse, its ability to lift topography is diminished. It would be instructive to compare the results of this benchmark using tracer particles with the results using compositional fields.

7 Extending ASPECT

ASPECT is designed to be an extensible code. In particular, it uses both a plugin architecture and a set of signals through which it is trivial to replace or extend certain components of the program. Examples of things that are simple to extend are:

- the material description,
- the geometry,
- the gravity description,
- the initial conditions,
- the boundary conditions,
- the functions that postprocess the solution, i.e., that can compute derived quantities such as heat fluxes over part of the boundary, mean velocities, etc.,
- the functions that generate derived quantities that can be put into graphical output files for visualization such as fields that depict the strength of the friction heating term, spatially dependent actual viscosities, and so on,
- the computation of refinement indicators,
- the determination of how long a computation should run.

This list may also have grown since this section was written. We will discuss the way this is achieved in Sections 7.1 and 7.3. Changing the core functionality, i.e., the basic equations (1)–(3), and how they are solved is arguably more involved. We will discuss this in Section 7.5.

> **Note:** The purpose of coming up with ways to make extensibility simple is that if you want to extend ASPECT for your own purposes, you can do this in a separate set of files that describe your situation, rather than by modifying the ASPECT source files themselves. This is important, because (i) it makes it possible for you to update ASPECT itself to a newer version without losing the functionality you added (because you did not make any changes to the ASPECT files themselves), (ii) because it makes it possible to keep unrelated changes separate in your own set of files, in a place where they are simple to find, and (iii) because it makes it much easier for you to share your modifications and additions with others.

Since ASPECT is written in C++ using the deal.II library, you will have to be proficient in C++. You will also likely have to familiarize yourself with this library for which there is an extensive amount of documentation:

- The manual at https://www.dealii.org/developer/doxygen/deal.II/index.html that describes in detail what every class, function and variable in deal.II does.
- A collection of modules at https://www.dealii.org/developer/doxygen/deal.II/modules.html that give an overview of whole groups of classes and functions and how they work together to achieve their goal.
- The deal.II tutorial at https://www.dealii.org/developer/doxygen/tutorial/index.html that provides a step-by-step introduction to the library using a sequence of several dozen programs that introduce gradually more complex topics. In particular, you will learn deal.II's way of *dimension independent programming* that allows you to write the program once, test it in 2d, and run the exact same code in 3d without having to debug it a second time.

- The step-31 and step-32 tutorial programs at https://www.dealii.org/developer/doxygen/deal.II/step_31.html and https://www.dealii.org/developer/doxygen/deal.II/step_32.html from which ASPECT directly descends.

- An overview of many general approaches to numerical methods, but also a discussion of deal.II and tools we use in programming, debugging and visualizing data are given in Wolfgang Bangerth's video lectures. These are linked from the deal.II website at https://www.dealii.org/ and directly available at http://www.math.tamu.edu/~bangerth/videos.html.

- The deal.II Frequently Asked Questions at https://github.com/dealii/dealii/wiki/Frequently-Asked-Questions that also have extensive sections on developing code with deal.II as well as on debugging. It also answers a number of questions we frequently get about the use of C++ in deal.II.

- Several other parts of the deal.II website at https://www.dealii.org/ also have information that may be relevant if you dive deeper into developing code. If you have questions, the mailing lists at https://www.dealii.org/mail.html are also of general help.

- A general overview of deal.II is also provided in the paper [BHK07].

As a general note, by default ASPECT utilizes a deal.II feature called *debug mode*, see also the introduction to this topic in Section 4.3. If you develop code, you will definitely want this feature to be on, as it will capture the vast majority of bugs you will invariably introduce in your code.

When you write new functionality and run the code for the first time, you will almost invariably first have to deal with a number of these assertions that point out problems in your code. While this may be annoying at first, remember that these are actual bugs in your code that have to be fixed anyway and that are much easier to find if the program aborts than if you have to go by their more indirect results such as wrong answers. The Frequently Asked Questions at https://github.com/dealii/dealii/wiki/Frequently-Asked-Questions contain a section on how to debug deal.II programs.

The downside of debug mode, as mentioned before, is that it makes the program much slower. Consequently, once you are confident that your program actually does what it is intended to do – **but no earlier!** –, you may want to switch to optimized mode that links ASPECT with a version of the deal.II libraries that uses compiler optimizations and that does not contain the `assert` statements discussed above. This switch can be facilitated by editing the top of the ASPECT `Makefile` and recompiling the program.

In addition to these general comments, ASPECT is itself extensively documented. You can find documentation on all classes, functions and namespaces starting from the `doc/doxygen/index.html` page.

7.1 The idea of plugins and the `SimulatorAccess` and `Introspection` classes

The most common modification you will probably want to do to ASPECT are to switch to a different material model (i.e., have different values of functional dependencies for the coefficients $\eta, \rho, C_p, \ldots$ discussed in Section 2.2); change the geometry; change the direction and magnitude of the gravity vector $\mathbf{g}$; or change the initial and boundary conditions.

To make this as simple as possible, all of these parts of the program (and some more) have been separated into modules that can be replaced quickly and where it is simple to add a new implementation and make it available to the rest of the program and the input parameter file. The way this is achieved is through the following two steps:

- The core of ASPECT really only communicates with material models, geometry descriptions, etc., through a simple and very basic interface. These interfaces are declared in the `include/aspect/material_model/interface.h`, `include/aspect/geometry_model/interface.h`, etc., header files. These classes are always called `Interface`, are located in namespaces that identify their purpose, and their documentation can be found from the general class overview in `doc/doxygen/classes.html`.

  To show an example of a rather minimal case, here is the declaration of the aspect::GravityModel::Interface class (documentation comments have been removed):

```
class Interface
{
  public:
    virtual ~Interface();

    virtual
    Tensor<1,dim>
    gravity_vector (const Point<dim> &position) const = 0;

    static void declare_parameters (ParameterHandler &prm);

    virtual void parse_parameters (ParameterHandler &prm);
};
```

If you want to implement a new model for gravity, you just need to write a class that derives from this base class and implements the `gravity_vector` function. If your model wants to read parameters from the input file, you also need to have functions called `declare_parameters` and `parse_parameters` in your class with the same signatures as the ones above. On the other hand, if the new model does not need any run-time parameters, you do not need to overload these functions.[34]

Each of the categories above that allow plugins have several implementations of their respective interfaces that you can use to get an idea of how to implement a new model.

- At the end of the file where you implement your new model, you need to have a call to the macro `ASPECT_REGISTER_GRAVITY_MODEL` (or the equivalent for the other kinds of plugins). For example, let us say that you had implemented a gravity model that takes actual gravimetric readings from the GRACE satellites into account, and had put everything that is necessary into a class `aspect::GravityModel::GRACE`. Then you need a statement like this at the bottom of the file:

```
ASPECT_REGISTER_GRAVITY_MODEL
(GRACE,
 "grace",
 "A␣gravity␣model␣derived␣from␣GRACE␣"
 "data.␣Run-time␣parameters␣are␣read␣from␣the␣parameter␣"
 "file␣in␣subsection␣'Radial␣constant'.");
```

Here, the first argument to the macro is the name of the class. The second is the name by which this model can be selected in the parameter file. And the third one is a documentation string that describes the purpose of the class (see, for example, Section 5.39 for an example of how existing models describe themselves).

This little piece of code ensures several things: (i) That the parameters this class declares are known when reading the parameter file. (ii) That you can select this model (by the name "grace") via the run-time parameter `Gravity model/Model name`. (iii) That ASPECT can create an object of this kind when selected in the parameter file.

Note that you need not announce the existence of this class in any other part of the code: Everything

[34]At first glance one may think that only the `parse_parameters` function can be overloaded since `declare_parameters` is not virtual. However, while the latter is called by the class that manages plugins through pointers to the interface class, the former function is called essentially at the time of registering a plugin, from code that knows the actual type and name of the class you are implementing. Thus, it can call the function – if it exists in your class, or the default implementation in the base class if it doesn't – even without it being declared as virtual.

should just work automatically.[35] This has the advantage that things are neatly separated: You do not need to understand the core of ASPECT to be able to add a new gravity model that can then be selected in an input file. In fact, this is true for all of the plugins we have: by and large, they just receive some data from the simulator and do something with it (e.g., postprocessors), or they just provide information (e.g., initial meshes, gravity models), but their writing does not require that you have a fundamental understanding of what the core of the program does.

The procedure for the other areas where plugins are supported works essentially the same, with the obvious change in namespace for the interface class and macro name.

In the following, we will discuss the requirements for individual plugins. Before doing so, however, let us discuss ways in which plugins can query other information, in particular about the current state of the simulation. To this end, let us not consider those plugins that by and large just provide information without any context of the simulation, such as gravity models, prescribed boundary velocities, or initial temperatures. Rather, let us consider things like postprocessors that can compute things like boundary heat fluxes. Taking this as an example (see Section 7.3.8), you are required to write a function with the following interface

```
template <int dim>
class MyPostprocessor : public aspect::Postprocess::Interface
{
  public:
    virtual
    std::pair<std::string,std::string>
    execute (TableHandler &statistics);

  // ... more things ...
```

The idea is that in the implementation of the `execute` function you would compute whatever you are interested in (e.g., heat fluxes) and return this information in the statistics object that then gets written to a file (see Sections 4.1 and 4.4.2). A postprocessor may also generate other files if it so likes – e.g., graphical output, a file that stores the locations of tracers, etc. To do so, obviously you need access to the current solution. This is stored in a vector somewhere in the core of ASPECT. However, this vector is, by itself, not sufficient: you also need to know the finite element space it is associated with, and for that the triangulation it is defined on. Furthermore, you may need to know what the current simulation time is. A variety of other pieces of information enters computations in these kinds of plugins.

All of this information is of course part of the core of ASPECT, as part of the aspect::Simulator class. However, this is a rather heavy class: it's got dozens of member variables and functions, and it is the one that does all of the numerical heavy lifting. Furthermore, to access data in this class would require that you need to learn about the internals, the data structures, and the design of this class. It would be poor design if plugins had to access information from this core class directly. Rather, the way this works is that those plugin classes that wish to access information about the state of the simulation inherit from the aspect::SimulatorAccess class. This class has an interface that looks like this:

```
template <int dim>
class SimulatorAccess
{
protected:
  double       get_time () const;

  std::string  get_output_directory () const;

  const LinearAlgebra::BlockVector &
```

[35] The existing implementations of models of the gravity and other interfaces declare the class in a header file and define the member functions in a `.cc` file. This is done so that these classes show up in our doxygen-generated documentation, but it is not necessary: you can put your entire class declaration and implementation into a single file as long as you call the macro discussed above on it. This single file is all you need to touch to add a new model.

```
    get_solution () const;

    const DoFHandler<dim> &
    get_dof_handler () const;

    // ... many more things ...
```

This way, SimulatorAccess makes information available to plugins without the need for them to understand details of the core of ASPECT. Rather, if the core changes, the SimulatorAccess class can still provide exactly the same interface. Thus, it insulates plugins from having to know the core. Equally importantly, since SimulatorAccess only offers its information in a read-only way it insulates the core from plugins since they can not interfere in the workings of the core except through the interface they themselves provide to the core.

Using this class, if a plugin class `MyPostprocess` is then not only derived from the corresponding `Interface` class but *also* from the SimulatorAccess class, then you can write a member function of the following kind (a nonsensical but instructive example; see Section 7.3.8 for more details on what postprocessors do and how they are implemented):[36]

```
template <int dim>
std::pair<std::string,std::string>
MyPostprocessor<dim>::execute (TableHandler &statistics)
{
  // compute the mean value of vector component 'dim' of the solution
  // (which here is the pressure block) using a deal.II function:
  const double
    average_pressure = VectorTools::compute_mean_value (this->get_mapping(),
                                                         this->get_dof_handler(),
                                                         QGauss<dim>(2),
                                                         this->get_solution(),
                                                         dim);
  statistics.add_value ("Average pressure", average_pressure);

  // return that there is nothing to print to screen (a useful
  // plugin would produce something more elaborate here):
  return std::pair<std::string,std::string>();
}
```

The second piece of information that plugins can use is called "introspection". In the code snippet above, we had to use that the pressure variable is at position `dim`. This kind of *implicit knowledge* is usually bad style: it is error prone because one can easily forget where each component is located; and it is an obstacle to the extensibility of a code if this kind of knowledge is scattered all across the code base.

Introspection is a way out of this dilemma. Using the `SimulatorAccess::introspection()` function returns a reference to an object (of type aspect::Introspection) that plugins can use to learn about these sort of conventions. For example, `this->introspection().component_mask.pressure` returns a component mask (a deal.II concept that describes a list of booleans for each component in a finite element that are true if a component is part of a variable we would like to select and false otherwise) that describes which component of the finite element corresponds to the pressure. The variable, `dim`, we need above to indicate that we want the pressure component can be accessed as `this->introspection().component_indices.pressure`. While this is certainly not shorter than just writing `dim`, it may in fact be easier to remember. It is most definitely less prone to errors and makes it simpler to extend the code in the future because we don't litter the sources with "magic constants" like the one above.

[36]For complicated, technical reasons, in the code below we need to access elements of the SimulatorAccess class using the notation `this->get_solution()`, etc. This is due to the fact that both the current class and the base class are templates. A long description of why it is necessary to use `this->` can be found in the DEAL.II Frequently Asked Questions.

This aspect::Introspection class has a significant number of variables that can be used in this way, i.e., they provide symbolic names for things one frequently has to do and that would otherwise require implicit knowledge of things such as the order of variables, etc.

7.2 How to write a plugin

Before discussing what each kind of plugin actually has to implement (see the next subsection), let us briefly go over what you actually have to do when implementing a new plugin. Essentially, the following steps are all you need to do:

- Create a file, say `my_plugin.cc` that contains the declaration of the class you want to implement. This class must be derived from one of the `Interface` classes we will discuss below. The file also needs to contain the implementation of all member functions of your class.

  As discussed above, it is possible – but not necessary – to split this file into two: a header file, say `my_plugin.h`, and the `my_plugin.cc` file (or, if you prefer, into multiple source files). We do this for all the existing plugins in ASPECT so that the documentation of these plugins shows up in the doxygen-generated documentation. However, for your own plugins, there is typically no need for this split. The only occasion where this would be useful is if some plugin actually makes use of a different plugin (e.g., the implementation of a gravity model of your own may want to query some specifics of a geometry model you also implemented); in that case the *using* plugin needs to be able to see the declaration of the class of the *used* plugin, and for this you will need to put the declaration of the latter into a header file.

- At the bottom of the `my_plugin.cc` file, put a statement that instantiates the plugin, documents it, and makes it available to the parameter file handlers by registering it. This is always done using one of the `ASPECT_REGISTER_*` macros that will be discussed in the next subsections; take a look at how they are used in the existing plugins in the ASPECT source files.

- You need to compile the file. There are two ways by which this can be achieved:

  - Put the `my_plugin.cc` into one of the ASPECT source directories and call `cmake .` followed by `make` to ensure that it actually gets compiled. This approach has the advantage that you do not need to worry much about how the file actually gets compiled. On the other hand, every time you modify the file, calling `make` requires not only compiling this one file, but also link ASPECT. Furthermore, when you upgrade from one version of ASPECT to another, you need to remember to copy the `my_plugin.cc` file.
  - Put the `my_plugin.cc` file into a directory of your choice and compile it into a shared library yourself. This may be as easy as calling

    ```
    g++ -I/path/to/aspect/headers -I/path/to/deal.II/headers \backslash
        -fPIC -shared my_plugin.cc -o my_plugin.so
    ```

    on Linux, but the command may be different on other systems. Now you only need to tell ASPECT to load this shared library at startup so that the plugin becomes available at run time and can be selected from the input parameter file. This is done using the `Additional shared libraries` parameter in the input file, see Section 5.2. This approach has the upside that you can keep all files that define new plugins in your own directories where you also run the simulations, also making it easier to keep around your plugins as you upgrade your ASPECT installation. On the other hand, compiling the file into a shared library is a bit more that you need to do yourself. Nevertheless, this is the preferred approach.

    In practice, the compiler line above can become tedious because it includes paths to the ASPECT and DEAL.II header files, but possibly also other things such as Trilinos headers, etc. Having to

remember all of these pieces is a hassle, and a much easier way is in fact to set up a mini-CMake project for this. To this end, simply copy the file `doc/plugin-CMakeLists.txt` to the directory where you have your plugin source files and rename it to `CMakeLists.txt`.

You can then just run the commands

```
cmake -DASPECT_DIR=/path/to/aspect .
make
```

and it should compile your plugin files into a shared library `my_plugin.so`. A concrete example of this process is discussed in Section 6.4.1. Of course, you may want to choose different names for the source files `source_1.cc`, `source_2.cc` or the name of the plugin `my_plugin`.

In essence, what these few lines do is that they find an ASPECT installation (i.e., the directory where you configured and compiled it, which may be the same directory as where you keep your sources, or a different one, as discussed in Section 3) in either the directory explicitly specified in the `ASPECT_DIR` variable passed to `cmake`, the shell environment, or just one directory up. It then sets up compiler paths and similar, and the following lines simply define the name of a plugin, list the source files for it, and define everything that's necessary to compile them into a shared library. Calling `make` on the command line then simply compiles everything.

Note: Complex projects built on ASPECT often require plugins of more than just one kind. For example, they may have plugins for the geometry, the material model, and for postprocessing. In such cases, you can either define multiple shared libraries by repeating the calls to `PROJECT`, `ADD_LIBRARY` and `ASPECT_SETUP_PLUGIN` for each shared library in your `CMakeLists.txt` file above, or you can just compile all of your source files into a single shared library. In the latter case, you only need to list a single library in your input file, but each plugin will still be selectable in the various sections of your input file as long as each of your classes has a corresponding `ASPECT_REGISTER_*` statement somewhere in the file where you have its definition. An even simpler approach is to just put everything into a single file – there is no requirement that different plugins are in separate files, though this is often convenient from a code organization point of view.

Note: If you choose to compile your plugins into a shared library yourself, you will need to recompile them every time you upgrade your ASPECT installation since we do not guarantee that the ASPECT application binary interface (ABI) will remain stable, even if it may not be necessary to actually change anything in the *implementation* of your plugin.

7.3 Materials, geometries, gravitation and other plugin types

7.3.1 Material models

The material model is responsible for describing the various coefficients in the equations that ASPECT solves. To implement a new material model, you need to overload the aspect::MaterialModel::Interface class and use the `ASPECT_REGISTER_MATERIAL_MODEL` macro to register your new class. The implementation of the new class should be in namespace `aspect::MaterialModel`. An example of a material model implemented this way is given in Section 6.4.9.

Specifically, your new class needs to implement the following interface:

```
template <int dim>
class aspect::MaterialModel::Interface
{
```

```
    public:
      // Physical parameters used in the basic equations
      virtual void evaluate(const MaterialModelInputs &in, MaterialModelOutputs &out) const=0;

      virtual bool is_compressible () const = 0;


      // Reference quantities
      virtual double reference_viscosity () const = 0;

      virtual double reference_density () const = 0;

      virtual double reference_thermal_expansion_coefficient () const = 0;


      // Auxiliary material properties used for postprocessing
      virtual double
      seismic_Vp (const double      temperature,
                  const double      pressure,
                  const std::vector<double> &compositional_fields,
                  const Point<dim> &position) const;

      virtual double
      seismic_Vs (const double      temperature,
                  const double      pressure,
                  const std::vector<double> &compositional_fields,
                  const Point<dim> &position) const;

      virtual unsigned int
      thermodynamic_phase (const double      temperature,
                           const double      pressure,
                           const std::vector<double> &compositional_fields) const;


      // Functions used in dealing with run-time parameters
      static void
      declare_parameters (ParameterHandler &prm);

      virtual void
      parse_parameters (ParameterHandler &prm);


      // Optional:
      virtual void initialize ();

      virtual void update ();
}
```

The main properties of the material are computed in the function evaluate() that takes a struct of type MaterialModelInputs and is supposed to fill a MaterialModelOutputs structure. For performance reasons this function is handling lookups at an arbitrary number of positions, so for each variable (for example viscosity), a std::vector is returned. The following members of MaterialModelOutputs need to be filled:

```
struct MaterialModelOutputs
{
          std::vector<double> viscosities;
          std::vector<double> densities;
```

```
        std::vector<double> thermal_expansion_coefficients;
        std::vector<double> specific_heat;
        std::vector<double> thermal_conductivities;
        std::vector<double> compressibilities;
}
```

The variables refer to the coefficients η, C_p, k, ρ in equations (1)–(3), each as a function of temperature, pressure, position, compositional fields and, in the case of the viscosity, the strain rate (all handed in by MaterialModelInputs). Implementations of evaluate() may of course choose to ignore dependencies on any of these arguments.

The remaining functions are used in postprocessing as well as handling run-time parameters. The exact meaning of these member functions is documented in the aspect::MaterialModel::Interface class documentation. Note that some of the functions listed above have a default implementation, as discussed on the documentation page just mentioned.

The function `is_compressible` returns whether we should consider the material as compressible or not, see Section 2.10.1 on the Boussinesq model. As discussed there, incompressibility as described by this function does not necessarily imply that the density is constant; rather, it may still depend on temperature or pressure. In the current context, compressibility simply means whether we should solve the continuity equation as $\nabla \cdot (\rho \mathbf{u}) = 0$ (compressible Stokes) or as $\nabla \cdot \mathbf{u} = 0$ (incompressible Stokes).

The purpose of the parameter handling functions has been discussed in the general overview of plugins above.

The functions initialize() and update() can be implemented if desired (the default implementation does nothing) and are useful if the material model has internal state. The function initialize() is called once during the initialization of ASPECT and can be used to allocate memory, initialize state, or read information from an external file. The function update() is called at the beginning of every time step.

Additionally, every material model has a member variable "model_dependence", declared in the Interface class, which can be accessed from the plugin as "this→model_dependence". This structure describes the nonlinear dependence of the various coefficients on pressure, temperature, composition or strain rate. This information will be used in future versions of ASPECT to implement a fully nonlinear solution scheme based on, for example, a Newton iteration. The initialization of this variable is optional, but only plugins that declare correct dependencies can benefit from these solver types. All packaged material models declare their dependencies in the parse_parameters() function and can be used as a starting point for implementations of new material models.

Older versions of ASPECT used to have individual functions like viscosity() instead of the evaluate() function discussed above. They are now a deprecated way of implementing a material model. You can get your old model working by deriving from InterfaceCompatibility instead of Interface.

7.3.2 Heating models

To be written

7.3.3 Geometry models

The geometry model is responsible for describing the domain in which we want to solve the equations. A domain is described in DEAL.II by a coarse mesh and, if necessary, an object that characterizes the boundary. Together, these two suffice to reconstruct any domain by adaptively refining the coarse mesh and placing new nodes generated by refining cells onto the surface described by the boundary object. The geometry model is also responsible for marking different parts of the boundary with different *boundary indicators* for which one can then, in the input file, select whether these boundaries should be Dirichlet-type (fixed temperature) or Neumann-type (no heat flux) boundaries for the temperature, and what kind of velocity conditions should hold there. In DEAL.II, a boundary indicator is a number of type `types::boundary_id`, but since boundaries are hard to remember and get right in input files, geometry models also have a function that provide a map from symbolic names that can be used to describe pieces of the boundary to the corresponding boundary indicators. For example, the simple `box` geometry model in 2d provides the map {`"left"`→`0`, `"right"`→`1`,

"bottom"→2,"top"→3}, and we have consistently used these symbolic names in the input files used in this manual.

To implement a new geometry model, you need to overload the aspect::GeometryModel::Interface class and use the ASPECT_REGISTER_GEOMETRY_MODEL macro to register your new class. The implementation of the new class should be in namespace aspect::GeometryModel.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::GeometryModel::Interface
{
  public:
    virtual
    void
    create_coarse_mesh (parallel::distributed::Triangulation<dim> &coarse_grid) const = 0;

    virtual
    double
    length_scale () const = 0;

    virtual
    double depth(const Point<dim> &position) const = 0;

    virtual
    Point<dim> representative_point(const double depth) const = 0;

    virtual
    double maximal_depth() const = 0;

    virtual
    std::set<types::boundary_id_t>
    get_used_boundary_indicators () const = 0;

    virtual
    std::map<std::string,types::boundary_id>
    get_symbolic_boundary_names_map () const;

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The kind of information these functions need to provide is extensively discussed in the documentation of this interface class at aspect::GeometryModel::Interface. The purpose of the last two functions has been discussed in the general overview of plugins above.

The create_coarse_mesh function does not only create the actual mesh (i.e., the locations of the vertices of the coarse mesh and how they connect to cells) but it must also set the boundary indicators for all parts of the boundary of the mesh. The DEAL.II glossary describes the purpose of boundary indicators as follows:

> In a Triangulation object, every part of the boundary is associated with a unique number (of type types::boundary_id) that is used to identify which boundary geometry object is responsible to generate new points when the mesh is refined. By convention, this boundary indicator is also often used to determine what kinds of boundary conditions are to be applied to a particular part

of a boundary. The boundary is composed of the faces of the cells and, in 3d, the edges of these faces.

By default, all boundary indicators of a mesh are zero, unless you are reading from a mesh file that specifically sets them to something different, or unless you use one of the mesh generation functions in namespace `GridGenerator` that have a 'colorize' option. A typical piece of code that sets the boundary indicator on part of the boundary to something else would look like this, here setting the boundary indicator to 42 for all faces located at $x = -1$:

```
for (typename Triangulation<dim>::active_cell_iterator
       cell = triangulation.begin_active();
     cell != triangulation.end();
     ++cell)
  for (unsigned int f=0; f<GeometryInfo<dim>::faces_per_cell; ++f)
    if (cell->face(f)->at_boundary())
      if (cell->face(f)->center()[0] == -1)
        cell->face(f)->set_boundary_indicator (42);
```

This calls functions `TriaAccessor::set_boundary_indicator`. In 3d, it may also be appropriate to call `TriaAccessor::set_all_boundary_indicators` instead on each of the selected faces. To query the boundary indicator of a particular face or edge, use `TriaAccessor::boundary_indicator`.

The code above only sets the boundary indicators of a particular part of the boundary, but it does not by itself change the way the Triangulation class treats this boundary for the purposes of mesh refinement. For this, you need to call `Triangulation::set_boundary` to associate a boundary object with a particular boundary indicator. This allows the Triangulation object to use a different method of finding new points on faces and edges to be refined; the default is to use a `StraightBoundary` object for all faces and edges. The results section of step-49 has a worked example that shows all of this in action.

The second use of boundary indicators is to describe not only which geometry object to use on a particular boundary but to select a part of the boundary for particular boundary conditions. *[...]*

Note: Boundary indicators are inherited from mother faces and edges to their children upon mesh refinement. Some more information about boundary indicators is also presented in a section of the documentation of the Triangulation class.

Two comments are in order here. First, if a coarse triangulation's faces already accurately represent where you want to pose which boundary condition (for example to set temperature values or determine which are no-flow and which are tangential flow boundary conditions), then it is sufficient to set these boundary indicators only once at the beginning of the program since they will be inherited upon mesh refinement to the child faces. Here, *at the beginning of the program* is equivalent to inside the `create_coarse_mesh())` function of the geometry module shown above that generates the coarse mesh.

Secondly, however, if you can only accurately determine which boundary indicator should hold where on a refined mesh – for example because the coarse mesh is the cube $[0, L]^3$ and you want to have a fixed velocity boundary describing an extending slab only for those faces for which $z > L - L_{\text{slab}}$ – then you need a way to set the boundary indicator for all boundary faces either to the value representing the slab or the fluid underneath *after every mesh refinement step*. By doing so, child faces can obtain boundary indicators different from that of their parents. DEAL.II triangulations support this kind of operations using a so-called *post-refinement signal*. In essence, what this means is that you can provide a function that will be called by the triangulation immediately after every mesh refinement step.

The way to do this is by writing a function that sets boundary indicators and that will be called by the `Triangulation` class. The triangulation does not provide a pointer to itself to the function being called,

nor any other information, so the trick is to get this information into the function. C++ provides a nice mechanism for this that is best explained using an example:

```
#include <deal.II/base/std_cxx1x/bind.h>

template <int dim>
void set_boundary_indicators (parallel::distributed::Triangulation<dim> &triangulation)
{
  ... set boundary indicators on the triangulation object ...
}

template <int dim>
void
MyGeometry<dim>::
create_coarse_mesh (parallel::distributed::Triangulation<dim> &coarse_grid) const
{
  ... create the coarse mesh ...

  coarse_grid.signals.post_refinement.connect
    (std_cxx1x::bind (&set_boundary_indicators<dim>,
                      std_cxx1x::ref(coarse_grid)));

}
```

What the call to `std_cxx1x::bind` does is to produce an object that can be called like a function with no arguments. It does so by taking the address of a function that does, in fact, take an argument but permanently fix this one argument to a reference to the coarse grid triangulation. After each refinement step, the triangulation will then call the object so created which will in turn call `set_boundary_indicators<dim>` with the reference to the coarse grid as argument.

This approach can be generalized. In the example above, we have used a global function that will be called. However, sometimes it is necessary that this function is in fact a member function of the class that generates the mesh, for example because it needs to access run-time parameters. This can be achieved as follows: assuming the `set_boundary_indicators()` function has been declared as a (non-static, but possibly private) member function of the `MyGeometry` class, then the following will work:

```
#include <deal.II/base/std_cxx1x/bind.h>

template <int dim>
void
MyGeometry<dim>::
set_boundary_indicators (parallel::distributed::Triangulation<dim> &triangulation) const
{
  ... set boundary indicators on the triangulation object ...
}

template <int dim>
void
MyGeometry<dim>::
create_coarse_mesh (parallel::distributed::Triangulation<dim> &coarse_grid) const
{
  ... create the coarse mesh ...

  coarse_grid.signals.post_refinement.connect
    (std_cxx1x::bind (&MyGeometry<dim>::set_boundary_indicators,
                      std_cxx1x::cref(*this),
                      std_cxx1x::ref(coarse_grid)));
```

```
}
```

Here, like any other member function, `set_boundary_indicators` implicitly takes a pointer or reference to the object it belongs to as first argument. `std::bind` again creates an object that can be called like a global function with no arguments, and this object in turn calls `set_boundary_indicators` with a pointer to the current object and a reference to the triangulation to work on. Note that because the `create_coarse_mesh` function is declared as `const`, it is necessary that the `set_boundary_indicators` function is also declared `const`.

> **Note:** For reasons that have to do with the way the `parallel::distributed::Triangulation` is implemented, functions that have been attached to the post-refinement signal of the triangulation are called more than once, sometimes several times, every time the triangulation is actually refined.

7.3.4 Gravity models

The gravity model is responsible for describing the magnitude and direction of the gravity vector at each point inside the domain. To implement a new gravity model, you need to overload the aspect::GravityModel::Interface class and use the `ASPECT_REGISTER_GRAVITY_MODEL` macro to register your new class. The implementation of the new class should be in namespace `aspect::GravityModel`.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::GravityModel::Interface
{
  public:
    virtual
    Tensor<1,dim>
    gravity_vector (const Point<dim> &position) const = 0;

    virtual
    void
    update ();

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The kind of information these functions need to provide is discussed in the documentation of this interface class at aspect::GravityModel::Interface. The first needs to return a gravity vector at a given position, whereas the second is called at the beginning of each time step, for example to allow a model to update itself based on the current time or the solution of the previous time step. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.3.5 Initial conditions

The initial conditions model is responsible for describing the initial temperature distribution throughout the domain. It essentially has to provide a function that for each point can return the initial temperature. Note that the model (1)–(3) does not require initial values for the pressure or velocity. However, if coefficients are nonlinear, one can significantly reduce the number of initial nonlinear iterations if a good guess for them

is available; consequently, ASPECT initializes the pressure with the adiabatically computed hydrostatic pressure, and a zero velocity. Neither of these two has to be provided by the objects considered in this section.

To implement a new initial conditions model, you need to overload the aspect::InitialConditions::Interface class and use the `ASPECT_REGISTER_INITIAL_CONDITIONS` macro to register your new class. The implementation of the new class should be in namespace `aspect::InitialConditions`.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::InitialConditions::Interface
{
  public:
    void
    initialize (const GeometryModel::Interface<dim>       &geometry_model,
                const BoundaryTemperature::Interface<dim> &boundary_temperature,
                const AdiabaticConditions<dim>            &adiabatic_conditions);

    virtual
    double
    initial_temperature (const Point<dim> &position) const = 0;

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The meaning of the first class should be clear. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.3.6 Prescribed velocity boundary conditions

Most of the time, one chooses relatively simple boundary values for the velocity: either a zero boundary velocity, a tangential flow model in which the tangential velocity is unspecified but the normal velocity is zero at the boundary, or one in which all components of the velocity are unspecified (i.e., for example, an outflow or inflow condition where the total stress in the fluid is assumed to be zero). However, sometimes we want to choose a velocity model in which the velocity on the boundary equals some prescribed value. A typical example is one in which plate velocities are known, for example their current values or historical reconstructions. In that case, one needs a model in which one needs to be able to evaluate the velocity at individual points at the boundary. This can be implemented via plugins.

To implement a new boundary velocity model, you need to overload the aspect::VelocityBoundaryConditions::Interface class and use the `ASPECT_REGISTER_VELOCITY_BOUNDARY_CONDITIONS` macro to register your new class. The implementation of the new class should be in namespace `aspect::VelocityBoundaryConditions`.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::VelocityBoundaryConditions::Interface
{
  public:
    virtual
    Tensor<1,dim>
    boundary_velocity (const Point<dim> &position) const = 0;
```

```
    virtual
    void
    initialize (const GeometryModel::Interface<dim> &geometry_model);

    virtual
    void
    update ();

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The first of these functions needs to provide the velocity at the given point. The next two are other member functions that can (but need not) be overloaded if a model wants to do initialization steps at the beginning of the program or at the beginning of each time step. Examples are models that need to call an external program to obtain plate velocities for the current time, or from historical records, in which case it is far cheaper to do so only once at the beginning of the time step than for every boundary point separately. See, for example, the aspect::VelocityBoundaryConditions::GPlates class.

The remaining functions are obvious, and are also discussed in the documentation of this interface class at aspect::VelocityBoundaryConditions::Interface. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.3.7 Temperature boundary conditions

The boundary conditions are responsible for describing the temperature values at those parts of the boundary at which the temperature is fixed (see Section 7.3.3 for how it is determined which parts of the boundary this applies to).

To implement a new boundary conditions model, you need to overload the aspect::BoundaryTemperature::Interface class and use the `ASPECT_REGISTER_BOUNDARY_TEMPERATURE_MODEL` macro to register your new class. The implementation of the new class should be in namespace `aspect::BoundaryTemperature`.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::BoundaryTemperature::Interface
{
  public:
    virtual
    double
    temperature (const GeometryModel::Interface<dim> &geometry_model,
                 const unsigned int                   boundary_indicator,
                 const Point<dim>                    &location) const = 0;

    virtual
    double minimal_temperature () const = 0;

    virtual
    double maximal_temperature () const = 0;

    static
```

```
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The first of these functions needs to provide the fixed temperature at the given point. The geometry model and the boundary indicator of the particular piece of boundary on which the point is located is also given as a hint in determining where this point may be located; this may, for example, be used to determine if a point is on the inner or outer boundary of a spherical shell. The remaining functions are obvious, and are also discussed in the documentation of this interface class at aspect::BoundaryTemperature::Interface. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.3.8 Postprocessors: Evaluating the solution after each time step

Postprocessors are arguably the most complex and powerful of the plugins available in ASPECT since they do not only passively provide any information but can actually compute quantities derived from the solution. They are executed once at the end of each time step and, unlike all the other plugins discussed above, there can be an arbitrary number of active postprocessors in the same program (for the plugins discussed in previous sections it was clear that there is always exactly one material model, geometry model, etc.).

Motivation. The original motivation for postprocessors is that the goal of a simulation is of course not the simulation itself, but that we want to do something with the solution. Examples for already existing postprocessors are:

- Generating output in file formats that are understood by visualization programs. This is facilitated by the aspect::Postprocess::Visualization class and a separate class of visualization postprocessors, see Section 7.3.9.
- Computing statistics about the velocity field (e.g., computing minimal, maximal, and average velocities), temperature field (minimal, maximal, and average temperatures), or about the heat fluxes across boundaries of the domain. This is provided by the aspect::Postprocess::VelocityStatistics, aspect::Postprocess::TemperatureStatistics, aspect::Postprocess::HeatFluxStatistics classes, respectively.

Since writing this text, there may have been other additions as well.

However, postprocessors can be more powerful than this. For example, while the ones listed above are by and large stateless, i.e., they do not carry information from one invocation at one timestep to the next invocation,[37] there is nothing that prohibits postprocessors from doing so. For example, the following ideas would fit nicely into the postprocessor framework:

- *Passive tracers:* If one would like to follow the trajectory of material as it is advected along with the flow field, one technique is to use tracer particles. To implement this, one would start with an initial population of particles distributed in a certain way, for example close to the core-mantle boundary. At the end of each time step, one would then need to move them forward with the flow field by one time increment. As long as these particles do not affect the flow field (i.e., they do not carry any information that feeds into material properties; in other words, they are *passive*), their location could well be stored in a postprocessor object and then be output in periodic intervals for visualization. In fact, such a passive tracer postprocessor is already available.

[37]This is not entirely true. The visualization plugin keeps track of how many output files it has already generated, so that they can be numbered consecutively.

- *Surface or crustal processes:* Another possibility would be to keep track of surface or crustal processes induced by mantle flow. An example would be to keep track of the thermal history of a piece of crust by updating it every time step with the heat flux from the mantle below. One could also imagine integrating changes in the surface topography by considering the surface divergence of the surface velocity computed in the previous time step: if the surface divergence is positive, the topography is lowered, eventually forming a trench; if the divergence is negative, a mountain belt eventually forms.

In all of these cases, the essential limitation is that postprocessors are *passive*, i.e., that they do not affect the simulation but only observe it.

The statistics file. Postprocessors fall into two categories: ones that produce lots of output every time they run (e.g., the visualization postprocessor), and ones that only produce one, two, or in any case a small and fixed number of often numerical results (e.g., the postprocessors computing velocity, temperature, or heat flux statistics). While the former are on their own in implementing how they want to store their data to disk, there is a mechanism in place that allows the latter class of postprocessors to store their data into a central file that is updated at the end of each time step, after all postprocessors are run.

To this end, the function that executes each of the postprocessors is given a reference to a `dealii::TableHandler` object that allows to store data in named columns, with one row for each time step. This table is then stored in the `statistics` file in the directory designated for output in the input parameter file. It allows for easy visualization of trends over all time steps. To see how to put data into this statistics object, take a look at the existing postprocessor objects.

Note that the data deposited into the statistics object need not be numeric in type, though it often is. An example of text-based entries in this table is the visualization class that stores the name of the graphical output file written in a particular time step.

Implementing a postprocessor. Ultimately, implementing a new postprocessor is no different than any of the other plugins. Specifically, you'll have to write a class that overloads the aspect::Postprocess::Interface base class and use the `ASPECT_REGISTER_POSTPROCESSOR` macro to register your new class. The implementation of the new class should be in namespace `aspect::Postprocess`.

In reality, however, implementing new postprocessors is often more difficult. Primarily, this difficulty results from two facts:

- Postprocessors are not self-contained (only providing information) but in fact need to access the solution of the model at each time step. That is, of course, the purpose of postprocessors, but it requires that the writer of a plugin has a certain amount of knowledge of how the solution is computed by the main `Simulator` class, and how it is represented in data structures. To alleviate this somewhat, and to insulate the two worlds from each other, postprocessors do not directly access the data structures of the simulator class. Rather, in addition to deriving from the aspect::Postprocess::Interface base class, postprocessors also derive from the SimulatorAccess class that has a number of member functions postprocessors can call to obtain read-only access to some of the information stored in the main class of ASPECT. See the documentation of this class to see what kind of information is available to postprocessors. See also Section 7.1 for more information about the `SimulatorAccess` class.

- Writing a new postprocessor typically requires a fair amount of knowledge how to leverage the DEAL.II library to extract information from the solution. The existing postprocessors are certainly good examples to start from in trying to understand how to do this.

Given these comments, the interface a postprocessor class has to implement is rather basic:

```
template <int dim>
class aspect::Postprocess::Interface
{
  public:
    virtual
```

```
    std::pair<std::string,std::string>
    execute (TableHandler &statistics) = 0;

    virtual
    void
    save (std::map<std::string, std::string> &status_strings) const;

    virtual
    void
    load (const std::map<std::string, std::string> &status_strings);

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The purpose of these functions is described in detail in the documentation of the aspect::Postprocess::Interface class. While the first one is responsible for evaluating the solution at the end of a time step, the `save/load` functions are used in checkpointing the program and restarting it at a previously saved point during the simulation. The first of these functions therefore needs to store the status of the object as a string under a unique key in the database described by the argument, while the latter function restores the same state as before by looking up the status string under the same key. The default implementation of these functions is to do nothing; postprocessors that do have non-static member variables that contain a state need to overload these functions.

There are numerous postprocessors already implemented. If you want to implement a new one, it would be helpful to look at the existing ones to see how they implement their functionality.

Postprocessors and checkpoint/restart. Postprocessors have `save()` and `load()` functions that are used to write the data a postprocessor has into a checkpoint file, and to load it again upon restart. This is important since many postprocessors store some state – say, a temporal average over all the time steps seen so far, or the number of the last graphical output file generated so that we know how the next one needs to be numbered.

The typical case is that this state is the same across all processors of a parallel computation. Consequently, what ASPECT writes into the checkpoint file is only the state obtained from the postprocessors on processor 0 of a parallel computation. On restart, all processors read from the same file and the postprocessors on *all* processors will be initialized by what the same postprocessor on processor 0 wrote.

There are situations where postprocessors do in fact store complementary information on different processors. At the time of writing this, one example is the postprocessor that supports advecting passive particles along the velocity field: on every processor, it handles only those particles that lie inside the part of the domain that is owned by this MPI rank. The serialization approach outlined above can not work in this case, for obvious reasons. In cases like this, one needs to implement the `save()` and `load()` differently than usual: one needs to put all variables that are common across processors into the maps of string as usual, but one then also needs to save all state that is different across processors, from all processors. There are two ways: If the amount of data is small, you can use MPI communications to send the state of all processors to processor zero, and have processor zero store it in the result so that it gets written into the checkpoint file; in the `load()` function, you will then have to identify which part of the text written by processor 0 is relevant to the current processor. Or, if your postprocessor stores a large amount of data, you may want to open a restart file specifically for this postprocessor, use MPI I/O or other ways to write into it, and do the reverse operation in `load()`.

Note that this approach requires that ASPECT actually calls the `save()` function on all processors. This in fact happens – though ASPECT also discards the result on all but processor zero.

7.3.9 Visualization postprocessors

As mentioned in the previous section, one of the postprocessors that are already implemented in ASPECT is the aspect::Postprocess::Visualization class that takes the solution and outputs it as a collection of files that can then be visualized graphically, see Section 4.4. The question is which variables to output: the solution of the basic equations we solve here is characterized by the velocity, pressure and temperature; on the other hand, we are frequently interested in derived, spatially and temporally variable quantities such as the viscosity for the actual pressure, temperature and strain rate at a given location, or seismic wave speeds.

ASPECT already implements a good number of such derived quantities that one may want to visualize. On the other hand, always outputting *all* of them would yield very large output files, and would furthermore not scale very well as the list continues to grow. Consequently, as with the postprocessors described in the previous section, what *can* be computed is implemented in a number of plugins and what *is* computed is selected in the input parameter file (see Section 5.100).

Defining visualization postprocessors works in much the same way as for the other plugins discussed in this section. Specifically, an implementation of such a plugin needs to be a class that derives from interface classes, should by convention be in namespace `aspect::Postprocess::VisualizationPostprocessors`, and is registered using a macro, here called `ASPECT_REGISTER_VISUALIZATION_POSTPROCESSOR`. Like the postprocessor plugins, visualization postprocessors can derive from class aspect::Postprocess::SimulatorAccess if they need to know specifics of the simulation such as access to the material models and to get access to the introspection facility outlined in Section 7.1. A typical example is the plugin that produces the viscosity as a spatially variable field by evaluating the viscosity function of the material model using the pressure, temperature and location of each visualization point (implemented in the `aspect::Postprocess::VisualizationPostprocessors::Viscosity` class). On the other hand, a hypothetical plugin that simply outputs the norm of the strain rate $\sqrt{\varepsilon(\mathbf{u}):\varepsilon(\mathbf{u})}$ would not need access to anything but the solution vector (which the plugin's main function is given as an argument) and consequently is not derived from the aspect::Postprocess::SimulatorAccess class.[38]

Visualization plugins can come in two flavors:

- *Plugins that compute things from the solution in a pointwise way:* The classes in this group are derived not only from the respective interface class (and possibly the SimulatorAccess class) but also from the deal.II class `DataPostprocessor` or any of the classes like `DataPostprocessorScalar` or `DataPostprocessorVector`. These classes can be thought of as filters: DataOut will call a function in them for every cell and this function will transform the values or gradients of the solution and other information such as the location of quadrature points into the desired quantity to output. A typical case would be if the quantity $g(x)$ you want to output can be written as a function $g(x) = G(u(x), \nabla u(x), x, ...)$ in a pointwise sense where $u(x)$ is the value of the solution vector (i.e., the velocities, pressure, temperature, etc) at an evaluation point. In the context of this program an example would be to output the density of the medium as a spatially variable function since this is a quantity that for realistic media depends pointwise on the values of the solution.

  To sum this, slightly confusing multiple inheritance up, visualization postprocessors do the following:

  - If necessary, they derive from aspect::Postprocess::SimulatorAccess.
  - They derive from aspect::Postprocess::VisualizationPostprocessors::Interface. The functions of this interface class are all already implemented as doing nothing in the base class but can be overridden in a plugin. Specifically, the following functions exist:

[38] The actual plugin `aspect::Postprocess::VisualizationPostprocessors::StrainRate` only computes $\sqrt{\varepsilon(\mathbf{u}):\varepsilon(\mathbf{u})}$ in the incompressible case. In the compressible case, it computes $\sqrt{[\varepsilon(\mathbf{u})-\frac{1}{3}(\textrm{tr}\,\varepsilon(\mathbf{u}))\mathbf{I}]:[\varepsilon(\mathbf{u})-\frac{1}{3}(\textrm{tr}\,\varepsilon(\mathbf{u}))\mathbf{I}]}$ instead. To test whether the model is compressible or not, the plugin needs access to the material model object, which the class gains by deriving from aspect::Postprocess::SimulatorAccess and then calling `this->get_material_model().is_compressible()`.

```
class Interface
{
  public:
    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);

    virtual
    void save (std::map<std::string, std::string> &status_strings) const;

    virtual
    void load (const std::map<std::string, std::string> &status_strings);
};
```

– They derive from either the `dealii::DataPostprocessor` class, or the simpler to use `dealii::DataPostprocesso` or `dealii::DataPostprocessorVector` classes. For example, to derive from the second of these classes, the following interface functions has to be implemented:

```
class dealii::DataPostprocessorScalar
{
  public:
    virtual
    void
    compute_derived_quantities_vector
      (const std::vector<Vector<double> >              &uh,
       const std::vector<std::vector<Tensor<1,dim> > > &duh,
       const std::vector<std::vector<Tensor<2,dim> > > &dduh,
       const std::vector<Point<dim> >                  &normals,
       const std::vector<Point<dim> >                  &evaluation_points,
       std::vector<Vector<double> >                    &computed_quantities) const;
};
```

What this function does is described in detail in the deal.II documentation. In addition, one has to write a suitable constructor to call `dealii::DataPostprocessorScalar::DataPostprocessorScalar`.

- *Plugins that compute things from the solution in a cellwise way:* The second possibility is for a class to not derive from `dealii::DataPostprocessor` but instead from the aspect::Postprocess::VisualizationPostprocessors::Ce class. In this case, a visualization postprocessor would generate and return a vector that consists of one element per cell. The intent of this option is to output quantities that are not pointwise functions of the solution but instead can only be computed as integrals or other functionals on a per-cell basis. A typical case would be error estimators that do depend on the solution but not in a pointwise sense; rather, they yield one value per cell of the mesh. See the documentation of the `CellDataVectorCreator` class for more information.

If all of this sounds confusing, we recommend consulting the implementation of the various visualization plugins that already exist in the ASPECT sources, and using them as a template.

7.3.10 Mesh refinement criteria

Despite research since the mid-1980s, it isn't completely clear how to refine meshes for complex situations like the ones modeled by ASPECT. The basic problem is that mesh refinement criteria either can refine based on some variable such as the temperature, the pressure, the velocity, or a compositional field, but that oftentimes this by itself is not quite what one wants. For example, we know that Earth has discontinuities, e.g., at 440km and 610km depth. In these places, densities and other material properties suddenly change. Their resolution in computation models is important as we know that they affect convection patterns. At the same time, there is only a small effect on the primary variables in a computation – maybe a jump in the second or third derivative, for example, but not a discontinuity that would be clear to see. As a consequence, automatic refinement criteria do not always refine these interfaces as well as necessary.

To alleviate this, ASPECT has plugins for mesh refinement. Through the parameters in Section 5.82, one can select when to refine but also which refinement criteria should be used and how they should be combined if multiple refinement criteria are selected. Furthermore, through the usual plugin mechanism, one can extend the list of available mesh refinement criteria (see the parameter "Strategy" in Section 5.82). Each such plugin is responsible for producing a vector of values (one per active cell on the current processor, though only those values for cells that the current processor owns are used) with an indicator of how badly this cell needs to be refined: large values mean that the cell should be refined, small values that the cell may be coarsened away.

To implement a new mesh refinement criterion, you need to overload the aspect::MeshRefinement::Interface class and use the `ASPECT_REGISTER_MESH_REFINEMENT_CRITERION` macro to register your new class. The implementation of the new class should be in namespace `aspect::MeshRefinement`.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::MeshRefinement::Interface
{
  public:
    virtual
    void
    execute (Vector<float> &error_indicators) const = 0;

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The first of these functions computes the set of refinement criteria (one per cell) and returns it in the given argument. Typical examples can be found in the existing implementations in the `source/mesh_refinement` directory. As usual, your termination criterion implementation will likely need to be derived from the `SimulatorAccess` to get access to the current state of the simulation.

The remaining functions are obvious, and are also discussed in the documentation of this interface class at aspect::MeshRefinement::Interface. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.3.11 Criteria for terminating a simulation

ASPECT allows for different ways of terminating a simulation. For example, the simulation may have reached a final time specified in the input file. However, it also allows for ways to terminate a simulation when it has reached a steady state (or, rather, some criterion determines that it is close enough to steady

state), or by an external action such as placing a specially named file in the output directory. The criteria determining termination of a simulation are all implemented in plugins. The parameters describing these criteria are listed in Section 5.109.

To implement a termination criterion, you need to overload the aspect::TerminationCriteria::Interface class and use the `ASPECT_REGISTER_TERMINATION_CRITERION` macro to register your new class. The implementation of the new class should be in namespace `aspect::TerminationCriteria`.

Specifically, your new class needs to implement the following basic interface:

```
template <int dim>
class aspect::TerminationCriteria::Interface
{
  public:
    virtual
    bool
    execute () const = 0;

    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

The first of these functions returns a value that indicates whether the simulation should be terminated. Typical examples can be found in the existing implementations in the `source/termination_criteria` directory. As usual, your termination criterion implementation will likely need to be derived from the `SimulatorAccess` to get access to the current state of the simulation.

The remaining functions are obvious, and are also discussed in the documentation of this interface class at aspect::TerminationCriteria::Interface. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.4 Extending ASPECT through the signals mechanism

Not all things you may want to do fit neatly into the list of plugins of the previous sections. Rather, there are cases where you may want to change things that are more of the one-off kind and that require code that is at a lower level and requires more knowledge about ASPECT's internal workings. For such changes, we still want to stick with the general principle outlined at the beginning of Section 7: You should be able to make all of your changes and extensions in your own files, without having to modify ASPECT's own sources.

To support this, ASPECT uses a "signals" mechanism. Signals are, in essence, objects that represent *events*, for example the fact that the solver has finished a time step. The core of ASPECT defines a number of such signals, and *triggers* them at the appropriate points. The idea of signals is now that you can *connect* to them: you can tell the signal that it should call a particular function every time the signal is triggered. The functions that are connected to a signal are called "slots" in common diction. One, several, or no slots may be connected to each signal.

There are two kinds of signals that ASPECT provides:

- Signals that are triggered at startup of the program: These are, in essence, signals that live in some kind of global scope. Examples are signals that declare additional parameters for use in input files, or that read the values of these parameters from a `ParameterHandler` object. These signals are static member variables of the structure that contains them and consequently exist only once for the entire program.

- Signals that reference specific events that happen inside a simulator object. These are regular member variables of the structure that contains them, and because each simulator object has such a structure, the signals exist once per simulator object. (Which in practice is only once per program, of course.)

For both of these kinds, a user-written plugin file can (but does not need) to register functions that connect functions in this file (i.e., slots) to their respective signals.

In the first case, code that registers slots with global signals would look like this:

```
// A function that will be called at the time when parameters are declared.
// It receives the dimension in which ASPECT will be run as the first argument,
// and the ParameterHandler object that holds the runtime parameter
// declarations as second argument.
void declare_parameters(const unsigned int dim,
                        ParameterHandler &prm)
{
  prm.declare_entry("My␣parameter", ...);
}


// The same for parsing parameters. 'my_parameter' is a parameter
// that stores something we want to read from the input file
// and use in other functions in this file (which we don't show here).
// For simplicity, we assume that it is an integer.
//
// The function also receives a first argument that contains all
// of the other (already parsed) arguments of the simulation, in
// case what you want to do here wants to refer to other parameters.
int my_parameter;

template <int dim>
void parse_parameters(const Parameters<dim> &parameters,
                      ParameterHandler &prm)
{
  my_parameter = prm.get_integer ("My␣parameter");
}


// Now have a function that connects slots (i.e., the two functions
// above) to the static signals. Do this for both the 2d and 3d
// case for generality.
void parameter_connector ()
{
  SimulatorSignals<2>::declare_additional_parameters.connect (&declare_parameters);
  SimulatorSignals<3>::declare_additional_parameters.connect (&declare_parameters);

  SimulatorSignals<2>::parse_additional_parameters.connect (&parse_parameters<2>);
  SimulatorSignals<3>::parse_additional_parameters.connect (&parse_parameters<3>);
}


// Finally register the connector function above to make sure it gets run
// whenever we load a user plugin that is mentioned among the additional
// shared libraries in the input file:
ASPECT_REGISTER_SIGNALS_PARAMETER_CONNECTOR(parameter_connector)
```

The second kind of signal can be connected to once a simulator object has been created. As above, one needs to define the slots, define a connector function, and register the connector function. The following gives an example:

```
// A function that is called at the end of creating the current constraints
// on degrees of freedom (i.e., the constraints that describe, for example,
// hanging nodes, boundary conditions, etc).
template <int dim>
void post_constraints_creation (const SimulatorAccess<dim> &simulator_access,
                                ConstraintMatrix &current_constraints)
{
  ...; // do whatever you want to do here
}


// A function that is called from the simulator object and that can connect
// a slot (such as the function above) to any of the signals declared in the
// structure passed as argument:
template <int dim>
void signal_connector (SimulatorSignals<dim> &signals)
{
  signals.post_constraints_creation.connect (&post_constraints_creation<dim>);
}


// Finally register the connector function so that it is called whenever
// a simulator object has been set up. For technical reasons, we need to
// register both 2d and 3d versions of this function:
ASPECT_REGISTER_SIGNALS_CONNECTOR(signal_connector<2>,
                                  signal_connector<3>)
```

As mentioned above, each signal may be connected to zero, one, or many slots. Consequently, you could have multiple plugins each of which connect to the same slot, or the connector function above may just connect multiple slots (i.e., functions in your program) to the same signal.

So what could one do in a place like this? One option would be to just monitor what is going on, e.g., in code like this that simply outputs into the statistics file (see Section 4.4.2):

```
template <int dim>
void post_constraints_creation (const SimulatorAccess<dim> &simulator_access,
                                ConstraintMatrix &current_constraints)
{
  simulator_access.get_statistics_object()
    .add_value ("number␣of␣constraints",
                current_constraints.n_constraints());
}
```

This will produce, for every time step (because this is how often the signal is called) an entry in a new column in the statistics file that records the number of constraints. On the other hand, it is equally possible to also modify the constraints object at this point. An application would be if you wanted to run a simulation where you prescribe the velocity in a part of the domain, e.g., for a subducting slab (see Section 6.2.9).

Signals exist for various waypoints in a simulation and you can consequently monitor and change what is happening inside a simulation by connecting your own functions to these signals. It would be pointless to list here what signals actually exist – simply refer to the documentation of the SimulatorSignals class for a complete list of signals you can connect to.

As a final note, it is generally true that writing functions that can connect to signals require significantly more internal knowledge of the workings of ASPECT than writing plugins through the mechanisms outlined

above. It also allows to affect the course of a simulation by working on the internal data structures of ASPECT in ways that are not available to the largely passive and reactive plugins discussed in previous sections. With this obviously also comes the potential for trouble. On the other hand, it also allows to do things with ASPECT that were not initially intended by the authors, and that would be hard or impossible to implement through plugins. An example would be to couple different codes by exchanging details of the internal data structures, or even update the solution vectors using information received from another code.

> **Note:** Chances are that if you think about using the signal mechanism, there is not yet a signal that is triggered at exactly the point where you need it. Consequently, you will be tempted to just put your code into the place where it fits inside ASPECT where it fits best. This is poor practice: it prevents you from upgrading to a newer version of ASPECT at a later time because this would overwrite the code you inserted.
> Rather, a more productive approach would be to either define a new signal that is triggered where you need it, and connect a function (slot) in your own plugin file to this signal using the mechanisms outlined above. Then send the code that defines and triggers the signal to the developers of ASPECT to make sure that it is also included in the next release. Alternatively, you can also simply ask on the mailing lists for someone to add such a signal in the place where you want it. Either way, adding signals is something that is easy to do, and we would much rather add signals than have people who modify the ASPECT source files for their own needs and are then stuck on a particular version.

7.5 Extending the basic solver

The core functionality of the code, i.e., that part of the code that implements the time stepping, assembles matrices, solves linear and nonlinear systems, etc., is in the `aspect::Simulator` class (see the doxygen documentation of this class). Since the implementation of this class has more than 3,000 lines of code, it is split into several files that are all located in the `source/simulator` directory. Specifically, functionality is split into the following files:

- `source/simulator/core.cc`: This file contains the functions that drive the overall algorithm (in particular `Simulator::run`) through the main time stepping loop and the functions immediately called by `Simulator::run`.
- `source/simulator/assembly.cc`: This is where all the functions are located that are related to assembling linear systems.
- `source/simulator/solver.cc`: This file provides everything that has to do with solving and preconditioning the linear systems.
- `source/simulator/initial_conditions.cc`: The functions in this file deal with setting initial conditions for all variables.
- `source/simulator/checkpoint_restart.cc`: The location of functionality related to saving the current state of the program to a set of files and restoring it from these files again.
- `source/simulator/helper_functions.cc`: This file contains a set of functions that do the odd thing in support of the rest of the simulator class.
- `source/simulator/parameters.cc`: This is where we define and read run-time parameters that pertain to the top-level functionality of the program.

Obviously, if you want to extend this core functionality, it is useful to first understand the numerical methods this class implements. To this end, take a look at the paper that describes these methods, see [KHB12]. Further, there are two predecessor programs whose extensive documentation is at a much higher

level than the one typically found inside ASPECT itself, since they are meant to teach the basic components of convection simulators as part of the DEAL.II tutorial:

- The step-31 program at https://www.dealii.org/developer/doxygen/deal.II/step_31.html: This program is the first version of a convection solver. It does not run in parallel, but it introduces many of the concepts relating to the time discretization, the linear solvers, etc.
- The step-32 program at https://www.dealii.org/developer/doxygen/deal.II/step_32.html: This is a parallel version of the step-31 program that already solves on a spherical shell geometry. The focus of the documentation in this program is on the techniques necessary to make the program run in parallel, as well as some of the consequences of making things run with realistic geometries, material models, etc.

Neither of these two programs is nearly as modular as ASPECT, but that was also not the goal in creating them. They will, however, serve as good introductions to the general approach for solving thermal convection problems.

Note: Neither this manual, nor the documentation in ASPECT makes much of an attempt at teaching how to use the DEAL.II library upon which ASPECT is built. Nevertheless, you will likely have to know at least the basics of DEAL.II to successfully work on the ASPECT code. We refer to the resources listed at the beginning of this section as well as references [BHK07, BHK12].

8 Future plans for ASPECT

We have a number of near-term plans for ASPECT that we hope to implement soon:

- *Iterating out the nonlinearity:* In the current version of ASPECT, we use the velocity, pressure and temperature of the previous time step to evaluate the coefficients that appear in the flow equations (1)–(2); and the velocity and pressure of the current time step as well as the previous time step's temperature to evaluate the coefficients in the temperature equation (3). This is an appropriate strategy if the model is not too nonlinear; however, it introduces inaccuracies and limits the size of the time step if coefficients strongly depend on the solution variables.

  To avoid this, one can iterate out the equations using either a fixed point or Newton scheme. Both approaches ensure that at the end of a time step, the values of coefficients and solution variables are consistent. On the other hand, one may have to solve the linear systems that describe a time step more than once, increasing the computational effort.

  We have started implementing such methods using a testbase code, based on earlier experiments by Jennifer Worthen [Wor12]. We hope to implement this feature in ASPECT early in 2012.
- *Faster 3d computations:* Whichever way you look at it, 3d computations are expensive. In parallel computations, the Stokes solve currently takes upward of 90% of the overall wallclock time, suggesting an obvious target for improvements based on better algorithms as well as from profiling the code to find hot spots. In particular, playing with better solver and/or preconditioner options would seem to be a useful goal.
- *Particle-based methods:* It is often useful to employ particle tracers to visualize where material is being transported. While conceptually simple, their implementation is made difficult in parallel computations if particles cross the boundary between parts of the regions owned by individual processors, as well as during re-partitioning the mesh between processors following mesh refinement. Eric Heien is working on an implementation of such passive tracers.

- *More realistic material models:* The number of material models available in ASPECT is currently relatively small. Obviously, how realistic a simulation is depends on how realistic a material model is. We hope to obtain descriptions of more realistic material descriptions over time, either given analytically or based on table-lookup of material properties.
- *Melting:* An important part of mantle behavior is melting. Melting not only affects the properties of the material such as density or viscosity, but it also leads to chemical segregation and, in fact, to the flow of two different fluids (the melt and the rock matrix) relative to each other. Modeling this additional process would yield significant insight.
- *Converting output into seismic velocities:* The predictions of mantle convection codes are often difficult to verify experimentally. On the other hand, simulations can be used to predict a seismic signature of the earth mantle – for example the location of transition zones that can be observed using seismic imaging. To facilitate such comparisons, it is of interest to output not only the primary solution variables but also convert them into the primary quantity visible in seismic imaging: compressive and shear wave velocities. Implementing this should be relatively straightforward if given a formula or table that expresses velocities in terms of the variables computed by ASPECT.

To end this section, let us repeat something already stated in the introduction:

Note: ASPECT is a community project. As such, we encourage contributions from the community to improve this code over time. Obvious candidates for such contributions are implementations of new plugins as discussed in Section 7.3 since they are typically self-contained and do not require much knowledge of the details of the remaining code. Obviously, however, we also encourage contributions to the core functionality in any form!

9 Finding answers to more questions

If you have questions that go beyond this manual, there are a number of resources:

- For questions on the source code of ASPECT, portability, installation, etc., use the ASPECT development mailing list at http://lists.geodynamics.org/cgi-bin/mailman/listinfo/aspect-devel. This mailing list is where the ASPECT developers all hang out.
- ASPECT is primarily based on the deal.II library (the dependency on Trilinos and p4est is primarily through deal.II, and not directly visible in the ASPECT source code). If you have particular questions about deal.II, contact the mailing lists described at https://www.dealii.org/mail.html.
- In case of more general questions about mantle convection, you can contact the CIG mantle convection mailing lists at http://lists.geodynamics.org/cgi-bin/mailman/listinfo/cig-MC.
- If you have specific questions about ASPECT that are not suitable for public and archived mailing lists, you can contact the primary developers:
  - Wolfgang Bangerth: bangerth@math.tamu.edu.
  - Timo Heister: heister@clemson.edu.

References

[AHFT13] V. Allken, R.S. Huismans, H. Fossen, and C. Thieulot. 3D numerical modelling of graben interaction and linkage: a case study of the Canyonlands grabens, Utah. *Basin Research*, 25:1–14, 2013.

[AHT11] V. Allken, R. Huismans, and C. Thieulot. Three dimensional numerical modelling of upper crustal extensional systems. *J. Geophys. Res.*, 116:B10409, 2011.

[AHT12] V. Allken, R. Huismans, and C. Thieulot. Factors controlling the mode of rift interaction in brittle-ductile coupled systems: a 3d numerical study. *Geochem. Geophys. Geosyst.*, 13(5):Q05010, 2012.

[BBC+89] B. Blankenbach, F. Busse, U. Christensen, L. Cserepes, D. Gunkel, U. Hansen, H. Harder, G. Jarvis, M. Koch, G. Marquart, D. Moore, P. Olson, H. Schmeling, and T. Schnaubelt. A benchmark comparison for mantle convection codes. *Geophys. J. Int.*, 98:23–38, 1989.

[BHK07] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24, 2007.

[BHK12] W. Bangerth, T. Heister, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*, 2012. `http://www.dealii.org/`.

[BRV+04] J. Badro, J.-P. Rueff, G. Vankó, G. Monaco, G. Fiquet, and F. Guyot. Electronic transitions in perovskite: Possible nonconvecting layers in the lower mantle. *Science*, 305:383–386, 2004.

[BSA+13] C. Burstedde, G. Stadler, L. Alisic, L. C. Wilcox, E. Tan, M. Gurnis, and O. Ghattas. Large-scale adaptive mantle convection simulation. *Geophysical Journal International*, 192.3:889–906, 2013.

[Bui12] S. J. H. Buiter. A review of brittle compressional wedge models. *Tectonophysics*, 530:1–17, 2012.

[BWG11] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Sci. Comput.*, 33(3):1103–1133, 2011.

[CSG+12] F. Crameri, H. Schmeling, G. J. Golabek, T. Duretz, R. Orendt, S. J. H. Buiter, D. A. May, B. J. P. Kaus, T. V. Gerya, and P. J. Tackley. A comparison of numerical surface topography calculations in geodynamic modelling: An evaluation of the 'sticky air' method. *Geophysical Journal International*, 189(1):38–54, 2012.

[DB14] C. R. Dohrmann and P. B. Bochev. A stabilized finite element method for the stokes problem based on polynomial pressure projections. *International Journal for Numerical Methods in Fluids*, 46:183–201, 20014.

[DHPRF04] J. Donea, A. Huerta, J.-Ph. Ponthot, and A. Rodríguez-Ferran. *Arbitrary Lagrangian-Eulerian Methods.* John Wiley & Sons, Ltd, 2004.

[DK08] Y. Deubelbeiss and B. J. P. Kaus. Comparison of eulerian and lagrangian numerical techniques for the stokes equations in the presence of strongly varying viscosity. *Physics of the Earth and Planetary Interiors*, 171:92–111, 2008.

[DMGT11] T. Duretz, D. A. May, T. V. Garya, and P. J. Tackley. Discretization errors and free surface stabilization in the finite difference and marker-in-cell method for applied geodynamics: A numerical study. *Geoch. Geoph. Geosystems*, 12:Q07004/1–26, 2011.

[GTZ+12] M. Gurnis, M. Turner, S. Zahirovic, L. DiCaprio, S. Spasojevic, R. D. Müller, J. Boyden, M. Seton, V. C. Manea, and D. J. Bower. Plate tectonic reconstructions with continuously closing plates. *Computers & Geosciences*, 38:35–42, 2012.

[H+11] M. A. Heroux et al. Trilinos web page, 2011. http://trilinos.sandia.gov.

[HBH+05] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31:397–423, 2005.

[Jea11] Jean-Luc Guermond and Richard Pasquetti and Bojan Popov. Entropy viscosity method for nonlinear conservation laws. *Journal of Computational Physics*, 230:4248–4267, 2011.

[Kau10] B.J.P. Kaus. Factors that control the angle of shear bands in geodynamic numerical models of brittle deformation. *Tectonophysics*, 484:36–47, 2010.

[KHB12] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation through modern numerical methods. *Geophysics Journal International*, 191:12–29, 2012.

[KMM10] B. J. P. Kaus, H. Mühlhaus, and D. A. May. A stabilization algorithm for geodynamic numerical simulations with a free surface. *Physics of the Earth and Planetary Interiors*, 181(1):12–20, 2010.

[MD04] C. Morency and M.-P. Doin. Numerical simulations of the mantle lithosphere delamination. *Journal of Geophysical Research: Solid Earth (1978–2012)*, 109(B3), 2004.

[MQL+07] L. Moresi, S. Quenette, V. Lemiale, C. Meriaux, B. Appelbe, and H. B. Mühlhaus. Computational approaches to studying non-linear dynamics of the crust and mantle. *Phys. Earth Planet. Interiors*, 163:69–82, 2007.

[RDvHW11] J. Ritsema, A. Deuss, H. J. van Heijst, and J. H. Woodhouse. S40rts: a degree-40 shear-velocity model for the mantle from new rayleigh wave dispersion, teleseismic traveltime and normal-mode splitting function measurements. *Geophysical Journal International*, 184:1223–1236, 2011.

[RvH00] J. Ritsema and H. J. van Heijst. Seismic imaging of structural heterogeneity in earth's mantle: Evidence for large-scale mantle flow. *Sci. Progr.*, 83:243–259, 2000.

[SBE+08] H. Schmeling, A. Y. Babeyko, A. Enns, C. Faccenna, F. Funiciello, T. Gerya, G. J. Golabek, S. Grigull, B. J. P. Kaus, G. Morra, S. M. Schmalholz, and J. van Hunen. A benchmark comparison of spontaneous subduction models—towards a free surface. *Physics of the Earth and Planetary Interiors*, 171:198–223, 2008.

[SP03] D. W. Schmid and Y. Y. Podladchikov. Analytical solutions for deformable elliptical inclusions in general shear. *Geophysical Journal International*, 155(1):269–288, 2003.

[STO01] G. Schubert, D. L. Turcotte, and P. Olson. *Mantle Convection in the Earth and Planets, Part 1*. Cambridge, 2001.

[Thi15] C. Thieulot. ELEFANT: a user-friendly multipurpose geodynamics code. Technical report, Utrecht University, 2015.

[TMK14] M. Thielmann, D. A. May, and B. J. P. Kaus. Discretization errors in the hybrid finite element particle-in-cell method. *Pure and Applied Geophysics*, 171:2165–2184, 2014.

[vKKS+97] P. E. van Keken, S. D. King, H. Schmeling, U. R. Christensen, D. Neumeister, and M.-P. Doin. A comparison of methods for the modeling of thermochemical convection. *J. Geoph. Res.*, 102:22477–22495, 1997.

[Wil99] S. D. Willett. Rheological dependence of extension in wedge models of convergent orogens. *Tectonophysics*, 305:419–435, 1999.

[Wor12] J. Worthen. *Inverse Problems in Mantle Convection: Models, Algorithms, and Applications.* PhD thesis, University of Texas at Austin, in preparation, 2012.

[Zho96] S. Zhong. Analytic solution for Stokes' flow with lateral variations in viscosity. *Geophys. J. Int.*, 124:18–28, 1996.

Index of run-time parameter entries

The following is a listing of all run-time parameters that can be set in the input parameter file. They are all described in Section 5 and the listed page numbers are where their detailed documentation can be found. A listing of all parameters sorted by the section name in which they are declared is given in the index on page 314 below.

A1, 119, 160
A2, 119, 161
A3, 119, 161
Activation energies, 124, 234
Activation energies for diffusion creep, 110
Activation energies for dislocation creep, 110
Activation volume, 124, 234
Activation volumes for diffusion creep, 110
Activation volumes for dislocation creep, 111
Additional refinement times, 133, 178, 182
Additional shared libraries, 45, 200, 210, 250, 253, 255, 273, 284
Additional tangential mesh velocity boundary indicators, 71
Adiabatic surface temperature, 16, 45
Adiabatic temperature gradient, 90
Age bottom boundary layer, 88
Age top boundary layer, 89
alpha, 70
Ambient temperature, 93
Amplitude, 89, 99
Angle, 99
Angle of internal friction, 115
Angular mode, 100, 270
Averaging operation, 106

B1, 119, 161
B2, 119, 161
B3, 120, 161
Base model, 106, 108
Bell shape limit, 106
beta, 70, 123, 163
Bilinear interpolation, 131
Bottom composition, 52, 53
Bottom temperature, 57, 58, 171, 181, 188
Boundary indicator to temperature mappings, 59
Boundary refinement indicators, 137
Box origin X coordinate, 73, 75
Box origin Y coordinate, 73, 75
Box origin Z coordinate, 73, 75

C1, 120, 161
C2, 120, 161
C3, 120, 162
Cells along circumference, 80
Center X, 94
Center x, 154
Center Y, 94
Center y, 154
Center Z, 94
Center z, 154
CFL number, 43, 45, 198, 272
Checkpoint on termination, 166
Chunk inner radius, 77
Chunk maximum latitude, 77
Chunk maximum longitude, 77
Chunk minimum latitude, 78
Chunk minimum longitude, 78
Chunk outer radius, 78
Coarsening fraction, 133, 179
Coefficient of yield stress increase with depth, 124, 234
Cohesion, 115
Cohesive strength of rocks at the surface, 124, 234
Command, 146
Composition polynomial degree, 41, 69
Composition solver tolerance, 41, 45
Composition viscosity prefactor, 115, 120, 128, 202, 268
Composition viscosity prefactor 1, 106
Composition viscosity prefactor 2, 106
Compositional field scaling factors, 137
Compressibility, 115, 120, 268
Compressible, 131
Coordinate system, 138, 139
Corresponding phase for density jump, 116, 267
cR, 71
Crust composition number, 85
Crust defined by composition, 86
Crust depth, 86

D1, 120, 162
D2, 121, 162
D3, 121, 162
Data directory, 51, 56, 62, 64, 67, 91, 95, 96, 108, 131, 152, 164, 227, 230
Data file name, 51, 56, 62, 67, 91, 152, 164

Data file time step, 51, 56, 62, 64
Data output format, 147, 196
Decreasing file order, 52, 57, 62, 64
Define transition by depth instead of pressure, 116
Densities, 111, 124, 126, 234
Density differential for compositional field 1, 107, 116, 121, 128, 193, 194, 202, 242, 263
Density differential for compositional field 2, 107, 194
Depth, 78
Depth dependence method, 109
Depth list, 109
Depth subdivisions, 79
Dimension, 29, 31, 43, 46, 168, 170, 181, 187, 202, 216, 222, 233, 251, 253, 256, 261
Discontinuous penalty, 70

E1, 121, 162
E2, 121, 162
East-West subdivisions, 79
Eccentricity, 79
Effective viscosity coefficient, 111
End step, 166, 235
End time, 29, 43, 46, 170, 187, 202, 216, 222, 251, 254, 256, 261, 272
Evaluation points, 147

File name, 167
Filename for initial geotherm table, 100
First data file model time, 52, 57, 62, 64
First data file number, 52, 57, 63, 65
Fixed composition boundary indicators, 140
Fixed temperature boundary indicators, 140, 172, 181, 186, 188, 198, 217, 223, 230, 233, 271, 293
Free surface boundary indicators, 140, 198
Free surface stabilization theta, 71, 198
Function constants, 44, 59, 61, 63, 68, 82, 85, 90, 92, 109, 138, 139, 151, 152, 164, 165, 171, 182, 186, 188, 234, 243, 244, 263
Function expression, 44, 59, 61, 63, 68, 82, 85, 90, 92, 110, 138, 139, 151, 152, 165, 171, 182, 186, 188, 190, 194, 196, 198, 203, 210, 212, 224, 234, 235, 243, 244, 263

Grain size, 111
Grain size exponents for diffusion creep, 111

Half decay times, 86
Heat capacity, 111, 125, 234
Heating rates, 86

Include adiabatic heating, 141, 172, 198, 202, 271
Include latent heat, 141
Include shear heating, 141, 172, 198, 202, 217, 221, 223, 230, 271
Inclusion gradient, 94
Inclusion shape, 94
Inclusion temperature, 94
Initial adaptive refinement, 29, 43, 133, 173, 178, 182, 189, 203, 217, 224, 235, 252, 255, 257, 263
Initial concentrations crust, 86
Initial concentrations mantle, 86
Initial condition file name, 95, 96, 227
Initial global refinement, 29, 43, 133, 173, 178, 182, 189, 203, 217, 224, 235, 252, 255, 257, 263
Inner composition, 55
Inner radius, 81, 216, 223, 270
Inner temperature, 60, 217, 224
Integration scheme, 148
Interpolate output, 155
Interpolation scheme, 148
Interpolation width, 230
Isotherm depth filename, 91
Isotherm temperature, 91

Jump height, 200

Latent heat, 131
Lateral viscosity file name, 131
Lateral wave number one, 93, 99
Lateral wave number two, 93, 99
Latitude repetitions, 78
Left composition, 53
Left composition lithosphere, 54
Left temperature, 57, 58, 171
Left temperature lithosphere, 58
Linear solver A block tolerance, 41, 46
Linear solver S block tolerance, 41, 46
Linear solver tolerance, 41, 46, 170, 202, 258
List of material properties, 160
List of model names, 83
List of normalized fields, 66
List of output variables, 146, 156, 193, 203, 264, 297
List of postprocessors, 29, 41, 143, 173, 182, 189, 192, 193, 195, 198, 203, 217, 224, 235, 243, 252, 255, 257, 259, 264, 294
List of tracer properties, 148, 196
Lithosphere thickness, 65, 98
Lithosphere thickness amplitude, 99
Lithospheric thickness, 75

Load balancing strategy, 149
Longitude repetitions, 78
Lower viscosity, 200

Magnitude, 82, 83, 93, 172, 177, 182, 202, 262, 271
Magnitude at surface, 83
Mass fraction cpx, 121, 163
Material averaging, 42, 101, 202, 203
Material file names, 131
Max nonlinear iterations, 47
Max nonlinear iterations in pre-refinement, 47
Maximal composition, 54
Maximal temperature, 60
Maximum lateral viscosity variation, 132
Maximum latitude, 154
Maximum longitude, 154
Maximum order, 95, 97
Maximum pyroxenite melt fraction, 122
Maximum radius, 154
Maximum relative deviation, 167
Maximum strain rate ratio iterations, 111
Maximum time step, 47, 233
Maximum tracers per cell, 149
Maximum viscosity, 112, 115, 132
Maximum x, 153
Maximum y, 153
Maximum z, 153
Mesh refinement, 299
Minimal composition, 54
Minimal temperature, 60
Minimum latitude, 155
Minimum longitude, 155
Minimum radius, 155
Minimum refinement level, 133
Minimum strain rate, 112, 125, 234
Minimum tracers per cell, 149
Minimum viscosity, 112, 115, 132
Minimum x, 153
Minimum y, 153
Minimum z, 154
Model name, 29, 43, 44, 50, 55, 67, 72, 81, 84, 87, 101, 163, 168, 171, 172, 181, 182, 187, 188, 190, 192, 194, 198, 200, 202, 203, 212, 216, 217, 220, 222–224, 227, 233–235, 242–244, 251, 252, 254, 256, 257, 259, 262, 263, 267, 270, 271, 273, 285, 287, 291

Names of fields, 66, 159, 234
Names of vectors, 160
NE corner, 79
Non-dimensional depth, 100
Nonlinear solver scheme, 47, 233
Nonlinear solver tolerance, 47
Normalize individual refinement criteria, 133
North-South subdivisions, 79
Number of cheap Stokes solver steps, 48
Number of elements, 86
Number of fields, 66, 190, 203, 212, 234, 263
Number of grouped files, 39, 158, 217, 225
Number of tracers, 150, 195, 196
Number of zones, 146, 235
NW corner, 79

Opening angle, 81, 216, 270
Outer composition, 55
Outer radius, 81, 216, 223, 270
Outer temperature, 61, 217, 224
Output directory, 29, 39, 43, 48, 170, 187, 202, 216, 222, 251, 254, 256, 261, 272
Output format, 33, 146, 159, 203, 217, 225, 235
Output mesh velocity, 159

Particle generator name, 150
Peridotite melting entropy change, 122
Phase transition Clapeyron slopes, 116, 267
Phase transition density jumps, 116, 267
Phase transition depths, 117, 267
Phase transition pressure widths, 117
Phase transition pressures, 117
Phase transition temperatures, 117, 267
Phase transition widths, 117, 267
Point one, 65, 230, 231
Point two, 65, 230, 231
Position, 89
Preexponential constant for viscous rheology law, 125, 234
Prefactors for diffusion creep, 112
Prefactors for dislocation creep, 112
Prescribe internal velocities, 210
Prescribed traction boundary indicators, 141
Prescribed velocity boundary indicators, 142, 172, 181, 186, 188, 198, 217, 223, 230, 251, 254, 256, 259, 271, 292
Pressure normalization, 15, 42, 48, 170, 202, 251, 252, 254, 256
Pyroxenite melting entropy change, 122

r1, 123, 163
r2, 124, 163
Radial layers, 155
Radial viscosity file name, 132
Radiogenic heating rate, 84

Radius, 80, 89
Radius repetitions, 78
Reaction depth, 107, 195
Reference compressibility, 127
Reference density, 43, 107, 114, 118, 122, 127, 129, 130, 172, 193, 200, 202, 262, 267, 270, 273
Reference specific heat, 107, 114, 118, 122, 128–130, 172, 200, 267, 270, 273
Reference strain rate, 115, 125, 234
Reference temperature, 43, 93, 95, 97, 107, 112, 114, 118, 122, 125, 126, 129, 130, 172, 193, 200, 227, 234, 267, 270, 273
Reference viscosity, 112, 114, 125, 132
Refinement criteria merge operation, 134
Refinement criteria scaling factors, 134
Refinement fraction, 134, 178, 263
Relative density of melt, 122
Remove degree 0 from perturbation, 95, 97, 227
Remove nullspace, 142, 271
Remove temperature heterogeneity down to specified depth, 95, 97, 227
Resume computation, 39, 43, 48, 183, 272
Right composition, 53, 54
Right composition lithosphere, 54
Right temperature, 58, 171
Right temperature lithosphere, 58
Rotation offset, 100, 270
Run on all processes, 146
Run postprocessors on initial refinement, 135

Scale factor, 52, 57, 63, 65, 67, 92, 164
SE corner, 79
Semi-major axis, 80
Shape radius, 94
Sigma, 100
Sign, 100
Solidus filename, 98
Specific heats, 126
Specify a lower maximum order, 96, 97
Spline knots depth file name, 96, 97, 227
Start time, 48, 187, 202, 251, 253, 256, 261
Steps between checkpoint, 66, 183, 225
Stokes velocity polynomial degree, 69, 180, 252, 254, 257
Strain rate residual tolerance, 113
Strategy, 135, 217, 224, 235, 263, 299
Stress exponents for diffusion creep, 113
Stress exponents for dislocation creep, 113
Stress exponents for plastic rheology, 125, 234
Stress exponents for viscous rheology, 126, 234
Subadiabaticity, 89
Subtract mean of dynamic topography, 147, 160
Supersolidus, 98
Surface pressure, 15, 49, 171
Surface temperature, 91
Surface velocity projection, 71
SW corner, 80

Tangential velocity boundary indicators, 143, 172, 181, 186, 188, 198, 202, 217, 223, 230, 233, 251, 254, 256, 259, 262, 271
Temperature amplitude, 99
Temperature polynomial degree, 41, 69, 180, 211
Temperature solver tolerance, 41, 49, 170
Temporary output location, 159
Terminate on failure, 146
Termination criteria, 166, 235, 299
Thermal conductivities, 127
Thermal conductivity, 108, 114, 118, 123, 128–130, 172, 188, 192, 194, 200, 268, 270, 273
Thermal diffusivity, 113, 126, 234
Thermal expansion coefficient, 108, 114, 118, 123, 128–130, 173, 189, 193, 194, 200, 202, 216, 222, 242, 268, 270, 273
Thermal expansion coefficient in initial temperature scaling, 96, 98, 227
Thermal expansion coefficient of melt, 123
Thermal expansivities, 113, 126, 127, 234
Thermal viscosity exponent, 108, 118, 123, 129, 268
Time between checkpoint, 66
Time between data output, 150, 196
Time between graphical output, 147, 159, 173, 182, 189, 193, 195, 198, 203, 217, 225
Time in steady state, 167
Time step, 230
Time steps between mesh refinement, 137, 173, 178, 182, 189, 217, 224
Timing output frequency, 49
Top composition, 53, 54
Top temperature, 58, 59, 172, 181, 188
Tracer weight, 151

Upper viscosity, 200
Use artificial viscosity smoothing, 70, 212
Use conduction timestep, 49
Use direct solver for Stokes system, 49
Use discontinuous composition discretization, 69
Use discontinuous temperature discretization, 69
Use lateral average temperature for viscosity, 132
Use locally conservative discretization, 69, 205, 252, 254, 257

Use simplified adiabatic heating, 84
Use years in output instead of seconds, 12, 50, 170, 187, 216, 222, 261

Variable names, 44, 60, 61, 63, 68, 82, 85, 90, 92, 110, 138, 139, 151, 153, 165, 166, 171, 182, 186, 188, 190, 194, 196, 198, 203, 210, 212, 234, 235, 243, 244, 263
Velocity file name, 65, 230
Vertical wave number, 93
Viscosities, 127
Viscosity, 43, 108, 118, 123, 128–130, 173, 189, 193, 194, 202, 216, 222, 242, 262, 268, 270, 273
Viscosity averaging scheme, 113, 127
Viscosity depth file, 109
Viscosity jump, 251, 257
Viscosity list, 109
Viscosity parameter, 259
Viscosity prefactors, 119, 268
Vs to density scaling, 96, 98, 227

Write in background thread, 159

X extent, 73, 75, 168, 171, 181, 187, 202, 233, 251, 254, 256, 262
X periodic, 73, 75
X periodic lithosphere, 75
X repetitions, 74, 76, 233

Y extent, 74, 76, 168, 171, 181, 187, 202, 233, 251, 254, 256, 262
Y periodic, 74, 76
Y periodic lithosphere, 76
Y repetitions, 74, 76
Y repetitions lithosphere, 76

Z extent, 74, 76, 181, 262
Z periodic, 74, 77
Z repetitions, 74, 77
Z repetitions lithosphere, 77
Zero velocity boundary indicators, 143, 172, 181, 186, 188, 198, 202, 217, 223, 251, 254, 256, 271

Index of run-time parameters with section names

The following is a listing of all run-time parameters, sorted by the section in which they appear. To find entries sorted by their name, rather than their section, see the index on page 309 above.

Additional shared libraries, 45, 200, 210, 250, 253, 255, 273, 284
Adiabatic conditions model
  Model name, 50
Adiabatic surface temperature, 16, 45

Boundary composition model
  Ascii data model
    Data directory, 51
    Data file name, 51
    Data file time step, 51
    Decreasing file order, 52
    First data file model time, 52
    First data file number, 52
    Scale factor, 52
  Box
    Bottom composition, 52
    Left composition, 53
    Right composition, 53
    Top composition, 53
  Box with lithosphere boundary indicators
    Bottom composition, 53
    Left composition, 53
    Left composition lithosphere, 54
    Right composition, 54
    Right composition lithosphere, 54
    Top composition, 54
  Initial composition
    Maximal composition, 54
    Minimal composition, 54
  Model name, 50, 235
  Spherical constant
    Inner composition, 55
    Outer composition, 55
Boundary temperature model
  Ascii data model
    Data directory, 56
    Data file name, 56
    Data file time step, 56
    Decreasing file order, 57
    First data file model time, 57
    First data file number, 57
    Scale factor, 57
  Box
    Bottom temperature, 57, 171, 181, 188
    Left temperature, 57, 171
    Right temperature, 58, 171
    Top temperature, 58, 172, 181, 188
  Box with lithosphere boundary indicators
    Bottom temperature, 58
    Left temperature, 58
    Left temperature lithosphere, 58
    Right temperature, 58
    Right temperature lithosphere, 58
    Top temperature, 59
  Constant
    Boundary indicator to temperature mappings, 59
  Function
    Function constants, 59
    Function expression, 59
    Maximal temperature, 60
    Minimal temperature, 60
    Variable names, 60
  Initial temperature
    Maximal temperature, 60
    Minimal temperature, 60
  Model name, 55, 171, 181, 188, 202, 217, 224, 234, 251, 254, 257, 262
  Spherical constant
    Inner temperature, 60, 217, 224
    Outer temperature, 61, 217, 224
Boundary traction model
  Function
    Function constants, 61
    Function expression, 61
    Variable names, 61
Boundary velocity model
  Ascii data model
    Data directory, 62
    Data file name, 62
    Data file time step, 62
    Decreasing file order, 62
    First data file model time, 62
    First data file number, 63
    Scale factor, 63
  Function
    Function constants, 63, 186, 188
    Function expression, 63, 186, 188
    Variable names, 63, 186, 188
  GPlates model
    Data directory, 64, 230

Data file time step, 64
Decreasing file order, 64
First data file model time, 64
First data file number, 65
Interpolation width, 230
Lithosphere thickness, 65
Point one, 65, 230
Point two, 65, 230
Scale factor, 65
Time step, 230
Velocity file name, 65, 230
Burstedde benchmark
Viscosity parameter, 259

CFL number, 43, 45, 198, 272
Checkpointing
Steps between checkpoint, 66, 183, 225
Time between checkpoint, 66
Composition solver tolerance, 41, 45
Compositional fields
List of normalized fields, 66
Names of fields, 66, 234
Number of fields, 66, 190, 203, 212, 234, 263
Compositional initial conditions
Ascii data model
Data directory, 67
Data file name, 67
Scale factor, 67
Function
Function constants, 68, 243, 244, 263
Function expression, 68, 190, 194, 203, 212, 234, 243, 244, 263
Variable names, 68, 190, 194, 203, 212, 234, 243, 244, 263
Model name, 67, 190, 194, 203, 212, 234, 243, 244, 263

Dimension, 29, 31, 43, 46, 168, 170, 181, 187, 202, 216, 222, 233, 251, 253, 256, 261
Discretization
Composition polynomial degree, 41, 69
Stabilization parameters
alpha, 70
beta, 70
cR, 71
Discontinuous penalty, 70
Use artificial viscosity smoothing, 70, 212
Stokes velocity polynomial degree, 69, 180, 252, 254, 257
Temperature polynomial degree, 41, 69, 180, 211
Use discontinuous composition discretization, 69
Use discontinuous temperature discretization, 69
Use locally conservative discretization, 69, 205, 252, 254, 257

End time, 29, 43, 46, 170, 187, 202, 216, 222, 251, 254, 256, 261, 272

Free surface
Additional tangential mesh velocity boundary indicators, 71
Free surface stabilization theta, 71, 198
Surface velocity projection, 71
Function
Function expression, 196
Variable names, 196

Geometry model
Box
Box origin X coordinate, 73
Box origin Y coordinate, 73
Box origin Z coordinate, 73
X extent, 73, 168, 171, 181, 187, 202, 233, 251, 254, 256, 262
X periodic, 73
X repetitions, 74, 233
Y extent, 74, 168, 171, 181, 187, 202, 233, 251, 254, 256, 262
Y periodic, 74
Y repetitions, 74
Z extent, 74, 181, 262
Z periodic, 74
Z repetitions, 74
Box with lithosphere boundary indicators
Box origin X coordinate, 75
Box origin Y coordinate, 75
Box origin Z coordinate, 75
Lithospheric thickness, 75
X extent, 75
X periodic, 75
X periodic lithosphere, 75
X repetitions, 76
Y extent, 76
Y periodic, 76
Y periodic lithosphere, 76
Y repetitions, 76
Y repetitions lithosphere, 76
Z extent, 76
Z periodic, 77
Z repetitions, 77

Z repetitions lithosphere, 77
Chunk
Chunk inner radius, 77
Chunk maximum latitude, 77
Chunk maximum longitude, 77
Chunk minimum latitude, 78
Chunk minimum longitude, 78
Chunk outer radius, 78
Latitude repetitions, 78
Longitude repetitions, 78
Radius repetitions, 78
Ellipsoidal chunk
Depth, 78
Depth subdivisions, 79
East-West subdivisions, 79
Eccentricity, 79
NE corner, 79
North-South subdivisions, 79
NW corner, 79
SE corner, 79
Semi-major axis, 80
SW corner, 80
Model name, 29, 72, 168, 171, 181, 187, 202, 216, 223, 233, 251, 252, 254, 256, 262, 270, 287
Sphere
Radius, 80
Spherical shell
Cells along circumference, 80
Inner radius, 81, 216, 223, 270
Opening angle, 81, 216, 270
Outer radius, 81, 216, 223, 270
Gravity model
Function
Function constants, 82
Function expression, 82
Variable names, 82
Model name, 81, 172, 182, 188, 202, 217, 224, 235, 251, 252, 254, 257, 259, 262, 271, 291
Radial constant
Magnitude, 82, 271
Radial linear
Magnitude at surface, 83
Vertical
Magnitude, 83, 172, 177, 182, 202, 262

Heating model
Adiabatic heating
Use simplified adiabatic heating, 84
Constant heating
Radiogenic heating rate, 84
Function
Function constants, 85
Function expression, 85
Variable names, 85
List of model names, 83
Model name, 84, 220
Radioactive decay
Crust composition number, 85
Crust defined by composition, 86
Crust depth, 86
Half decay times, 86
Heating rates, 86
Initial concentrations crust, 86
Initial concentrations mantle, 86
Number of elements, 86

Initial conditions
Adiabatic
Age bottom boundary layer, 88
Age top boundary layer, 89
Amplitude, 89
Function/Function constants, 90
Function/Function expression, 90
Function/Variable names, 90
Position, 89
Radius, 89
Subadiabaticity, 89
Adiabatic boundary
Adiabatic temperature gradient, 90
Data directory, 91
Isotherm depth filename, 91
Isotherm temperature, 91
Surface temperature, 91
Ascii data model
Data directory, 91
Data file name, 91
Scale factor, 92
Function
Function constants, 44, 92, 171, 182, 234
Function expression, 44, 92, 171, 182, 188, 198, 203, 224, 234, 263
Variable names, 44, 92, 171, 182, 188, 198, 234
Harmonic perturbation
Lateral wave number one, 93
Lateral wave number two, 93
Magnitude, 93
Reference temperature, 93
Vertical wave number, 93
Inclusion shape perturbation
Ambient temperature, 93
Center X, 94

Center Y, 94
Center Z, 94
Inclusion gradient, 94
Inclusion shape, 94
Inclusion temperature, 94
Shape radius, 94
Model name, 44, 87, 171, 182, 188, 198, 203, 217, 224, 227, 234, 251, 254, 257, 262, 270, 291
S40RTS perturbation
Data directory, 95, 227
Initial condition file name, 95, 227
Maximum order, 95
Reference temperature, 95, 227
Remove degree 0 from perturbation, 95, 227
Remove temperature heterogeneity down to specified depth, 95, 227
Specify a lower maximum order, 96
Spline knots depth file name, 96, 227
Thermal expansion coefficient in initial temperature scaling, 96, 227
Vs to density scaling, 96, 227
SAVANI perturbation
Data directory, 96
Initial condition file name, 96
Maximum order, 97
Reference temperature, 97
Remove degree 0 from perturbation, 97
Remove temperature heterogeneity down to specified depth, 97
Specify a lower maximum order, 97
Spline knots depth file name, 97
Thermal expansion coefficient in initial temperature scaling, 98
Vs to density scaling, 98
Solidus
Data/Solidus filename, 98
Lithosphere thickness, 98
Perturbation/Lateral wave number one, 99
Perturbation/Lateral wave number two, 99
Perturbation/Lithosphere thickness amplitude, 99
Perturbation/Temperature amplitude, 99
Supersolidus, 98
Spherical gaussian perturbation
Amplitude, 99
Angle, 99
Filename for initial geotherm table, 100
Non-dimensional depth, 100
Sigma, 100
Sign, 100
Spherical hexagonal perturbation
Angular mode, 100, 270
Rotation offset, 100, 270

Linear solver A block tolerance, 41, 46
Linear solver S block tolerance, 41, 46
Linear solver tolerance, 41, 46, 170, 202, 258
List of tracer properties, 196

Material model
Averaging
Averaging operation, 106
Base model, 106
Bell shape limit, 106
Composition reaction model
Composition viscosity prefactor 1, 106
Composition viscosity prefactor 2, 106
Density differential for compositional field 1, 107, 194
Density differential for compositional field 2, 107, 194
Reaction depth, 107, 195
Reference density, 107
Reference specific heat, 107
Reference temperature, 107
Thermal conductivity, 108, 194
Thermal expansion coefficient, 108, 194
Thermal viscosity exponent, 108
Viscosity, 108, 194
Depth dependent model
Base model, 108
Data directory, 108
Depth dependence method, 109
Depth list, 109
Viscosity depth file, 109
Viscosity depth function/Function constants, 109
Viscosity depth function/Function expression, 110
Viscosity depth function/Variable names, 110
Viscosity list, 109
Diffusion dislocation
Activation energies for diffusion creep, 110
Activation energies for dislocation creep, 110
Activation volumes for diffusion creep, 110
Activation volumes for dislocation creep, 111
Densities, 111
Effective viscosity coefficient, 111

Grain size, 111
Grain size exponents for diffusion creep, 111
Heat capacity, 111
Maximum strain rate ratio iterations, 111
Maximum viscosity, 112
Minimum strain rate, 112
Minimum viscosity, 112
Prefactors for diffusion creep, 112
Prefactors for dislocation creep, 112
Reference temperature, 112
Reference viscosity, 112
Strain rate residual tolerance, 113
Stress exponents for diffusion creep, 113
Stress exponents for dislocation creep, 113
Thermal diffusivity, 113
Thermal expansivities, 113
Viscosity averaging scheme, 113
Drucker Prager
Reference density, 114
Reference specific heat, 114
Reference temperature, 114
Reference viscosity, 114
Thermal conductivity, 114
Thermal expansion coefficient, 114
Viscosity/Angle of internal friction, 115
Viscosity/Cohesion, 115
Viscosity/Maximum viscosity, 115
Viscosity/Minimum viscosity, 115
Viscosity/Reference strain rate, 115
Inclusion
Viscosity jump, 257
Latent heat
Composition viscosity prefactor, 115, 268
Compressibility, 115, 268
Corresponding phase for density jump, 116, 267
Define transition by depth instead of pressure, 116
Density differential for compositional field 1, 116
Phase transition Clapeyron slopes, 116, 267
Phase transition density jumps, 116, 267
Phase transition depths, 117, 267
Phase transition pressure widths, 117
Phase transition pressures, 117
Phase transition temperatures, 117, 267
Phase transition widths, 117, 267
Reference density, 118, 267
Reference specific heat, 118, 267
Reference temperature, 118, 267
Thermal conductivity, 118, 268
Thermal expansion coefficient, 118, 268
Thermal viscosity exponent, 118, 268
Viscosity, 118, 268
Viscosity prefactors, 119, 268
Latent heat melt
A1, 119
A2, 119
A3, 119
B1, 119
B2, 119
B3, 120
beta, 123
C1, 120
C2, 120
C3, 120
Composition viscosity prefactor, 120
Compressibility, 120
D1, 120
D2, 121
D3, 121
Density differential for compositional field 1, 121
E1, 121
E2, 121
Mass fraction cpx, 121
Maximum pyroxenite melt fraction, 122
Peridotite melting entropy change, 122
Pyroxenite melting entropy change, 122
r1, 123
r2, 124
Reference density, 122
Reference specific heat, 122
Reference temperature, 122
Relative density of melt, 122
Thermal conductivity, 123
Thermal expansion coefficient, 123
Thermal expansion coefficient of melt, 123
Thermal viscosity exponent, 123
Viscosity, 123
Material averaging, 42, 101, 202, 203
Model name, 43, 101, 172, 188, 192, 194, 200, 202, 216, 222, 234, 242, 251, 254, 257, 259, 262, 267, 270, 273, 285
Morency and Doin
Activation energies, 124, 234
Activation volume, 124, 234
Coefficient of yield stress increase with depth, 124, 234
Cohesive strength of rocks at the surface,

124, 234
Densities, 124, 234
Heat capacity, 125, 234
Minimum strain rate, 125, 234
Preexponential constant for viscous rheology law, 125, 234
Reference strain rate, 125, 234
Reference temperature, 125, 234
Reference viscosity, 125
Stress exponents for plastic rheology, 125, 234
Stress exponents for viscous rheology, 126, 234
Thermal diffusivity, 126, 234
Thermal expansivities, 126, 234
Multicomponent
Densities, 126
Reference temperature, 126
Specific heats, 126
Thermal conductivities, 127
Thermal expansivities, 127
Viscosities, 127
Viscosity averaging scheme, 127
Simple compressible model
Reference compressibility, 127
Reference density, 127
Reference specific heat, 128
Thermal conductivity, 128
Thermal expansion coefficient, 128
Viscosity, 128
Simple model
Composition viscosity prefactor, 128, 202
Density differential for compositional field 1, 128, 193, 202, 242, 263
Reference density, 43, 129, 172, 193, 202, 262, 270
Reference specific heat, 129, 172, 270
Reference temperature, 43, 129, 172, 193, 270
Thermal conductivity, 129, 172, 188, 192, 270
Thermal expansion coefficient, 129, 173, 189, 193, 202, 216, 222, 242, 270
Thermal viscosity exponent, 129
Viscosity, 43, 129, 173, 189, 193, 202, 216, 222, 242, 262, 270
Simpler model
Reference density, 130
Reference specific heat, 130
Reference temperature, 130
Thermal conductivity, 130
Thermal expansion coefficient, 130
Viscosity, 130
Simpler with crust model
Jump height, 200
Lower viscosity, 200
Reference density, 200
Reference specific heat, 200
Reference temperature, 200
Thermal conductivity, 200
Thermal expansion coefficient, 200
Upper viscosity, 200
SolCx
Viscosity jump, 251
Steinberger model
Bilinear interpolation, 131
Compressible, 131
Data directory, 131
Latent heat, 131
Lateral viscosity file name, 131
Material file names, 131
Maximum lateral viscosity variation, 132
Maximum viscosity, 132
Minimum viscosity, 132
Radial viscosity file name, 132
Reference viscosity, 132
Use lateral average temperature for viscosity, 132
VoT model
Reference density, 273
Reference specific heat, 273
Reference temperature, 273
Thermal conductivity, 273
Thermal expansion coefficient, 273
Viscosity, 273
Max nonlinear iterations, 47
Max nonlinear iterations in pre-refinement, 47
Maximum time step, 47, 233
Mesh refinement, 299
Additional refinement times, 133, 178, 182
Boundary
Boundary refinement indicators, 137
Coarsening fraction, 133, 179
Composition
Compositional field scaling factors, 137
Initial adaptive refinement, 29, 43, 133, 173, 178, 182, 189, 203, 217, 224, 235, 252, 255, 257, 263
Initial global refinement, 29, 43, 133, 173, 178, 182, 189, 203, 217, 224, 235, 252, 255, 257, 263
Maximum refinement function

Coordinate system, 138
Function constants, 138
Function expression, 138
Variable names, 138
Minimum refinement function
Coordinate system, 139
Function constants, 139
Function expression, 139, 235
Variable names, 139, 235
Minimum refinement level, 133
Normalize individual refinement criteria, 133
Refinement criteria merge operation, 134
Refinement criteria scaling factors, 134
Refinement fraction, 134, 178, 263
Run postprocessors on initial refinement, 135
Strategy, 135, 217, 224, 235, 263, 299
Time steps between mesh refinement, 137, 173, 178, 182, 189, 217, 224
Model settings
Fixed composition boundary indicators, 140
Fixed temperature boundary indicators, 140, 172, 181, 186, 188, 198, 217, 223, 230, 233, 271, 293
Free surface boundary indicators, 140, 198
Include adiabatic heating, 141, 172, 198, 202, 271
Include latent heat, 141
Include shear heating, 141, 172, 198, 202, 217, 221, 223, 230, 271
Prescribed traction boundary indicators, 141
Prescribed velocity boundary indicators, 142, 172, 181, 186, 188, 198, 217, 223, 230, 251, 254, 256, 259, 271, 292
Remove nullspace, 142, 271
Tangential velocity boundary indicators, 143, 172, 181, 186, 188, 198, 202, 217, 223, 230, 233, 251, 254, 256, 259, 262, 271
Zero velocity boundary indicators, 143, 172, 181, 186, 188, 198, 202, 217, 223, 251, 254, 256, 271

Nonlinear solver scheme, 47, 233
Nonlinear solver tolerance, 47
Number of cheap Stokes solver steps, 48

Output directory, 29, 39, 43, 48, 170, 187, 202, 216, 222, 251, 254, 256, 261, 272

Point one, 231
Point two, 231
Postprocess
Command
Command, 146
Run on all processes, 146
Terminate on failure, 146
Depth average
List of output variables, 146
Number of zones, 146, 235
Output format, 146, 225, 235
Time between graphical output, 147, 217, 225
Dynamic Topography
Subtract mean of dynamic topography, 147
List of postprocessors, 29, 41, 143, 173, 182, 189, 192, 193, 195, 198, 203, 217, 224, 235, 243, 252, 255, 257, 259, 264, 294
Point values
Evaluation points, 147
Tracers
Data output format, 147, 196
Function/Function constants, 151
Function/Function expression, 151
Function/Variable names, 151
Generator/Ascii file/Data directory, 152
Generator/Ascii file/Data file name, 152
Generator/Probability density function/Function constants, 152
Generator/Probability density function/Function expression, 152
Generator/Probability density function/Variable names, 153
Generator/Uniform box/Maximum x, 153
Generator/Uniform box/Maximum y, 153
Generator/Uniform box/Maximum z, 153
Generator/Uniform box/Minimum x, 153
Generator/Uniform box/Minimum y, 153
Generator/Uniform box/Minimum z, 154
Generator/Uniform radial/Center x, 154
Generator/Uniform radial/Center y, 154
Generator/Uniform radial/Center z, 154
Generator/Uniform radial/Maximum latitude, 154
Generator/Uniform radial/Maximum longitude, 154
Generator/Uniform radial/Maximum radius, 154
Generator/Uniform radial/Minimum latitude, 155
Generator/Uniform radial/Minimum longitude, 155
Generator/Uniform radial/Minimum radius, 155

Generator/Uniform radial/Radial layers, 155
Integration scheme, 148
Interpolation scheme, 148
List of tracer properties, 148
Load balancing strategy, 149
Maximum tracers per cell, 149
Minimum tracers per cell, 149
Number of tracers, 150, 195, 196
Particle generator name, 150
Time between data output, 150, 196
Tracer weight, 151
Visualization
Compositional fields as vectors/Names of fields, 159
Compositional fields as vectors/Names of vectors, 160
Dynamic Topography/Subtract mean of dynamic topography, 160
Interpolate output, 155
List of output variables, 156, 193, 203, 264, 297
Material properties/List of material properties, 160
Melt fraction/A1, 160
Melt fraction/A2, 161
Melt fraction/A3, 161
Melt fraction/B1, 161
Melt fraction/B2, 161
Melt fraction/B3, 161
Melt fraction/beta, 163
Melt fraction/C1, 161
Melt fraction/C2, 161
Melt fraction/C3, 162
Melt fraction/D1, 162
Melt fraction/D2, 162
Melt fraction/D3, 162
Melt fraction/E1, 162
Melt fraction/E2, 162
Melt fraction/Mass fraction cpx, 163
Melt fraction/r1, 163
Melt fraction/r2, 163
Number of grouped files, 39, 158, 217, 225
Output format, 33, 159, 203, 217, 225
Output mesh velocity, 159
Temporary output location, 159
Time between graphical output, 159, 173, 182, 189, 193, 195, 198, 203, 217, 225
Write in background thread, 159
Prescribe internal velocities, 210
Prescribed Stokes solution
Ascii data model
Data directory, 164
Data file name, 164
Scale factor, 164
Model name, 163
Pressure function
Function constants, 164
Function expression, 165
Variable names, 165
Velocity function
Function constants, 165
Function expression, 165
Variable names, 166
Prescribed velocities
Indicator function
Function expression, 210
Variable names, 210
Velocity function
Function expression, 210
Variable names, 210
Pressure normalization, 15, 42, 48, 170, 202, 251, 252, 254, 256

Resume computation, 39, 43, 48, 183, 272

Start time, 48, 187, 202, 251, 253, 256, 261
Surface pressure, 15, 49, 171

Temperature solver tolerance, 41, 49, 170
Termination criteria, 299
Checkpoint on termination, 166
End step, 166, 235
Steady state velocity
Maximum relative deviation, 167
Time in steady state, 167
Termination criteria, 166, 235
User request
File name, 167
Timing output frequency, 49

Use conduction timestep, 49
Use direct solver for Stokes system, 49
Use years in output instead of seconds, 12, 50, 170, 187, 216, 222, 261