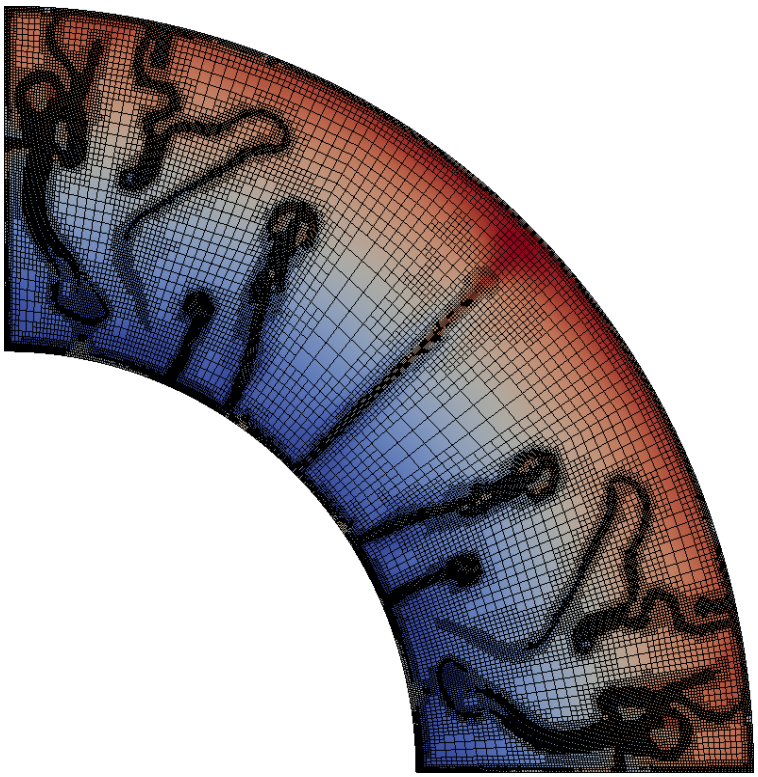


Aspect

Advanced Solver for Problems in Earth's ConvecTion

Preview release, version 0.8
(generated from subversion: *Revision* : 1135)

Wolfgang Bangerth
Thomas Geenen
Timo Heister
Martin Kronbichler

Contents

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 lengthscales is dominated by polar oblateness, but deviations from spherical shape relevant to convection patterns may go down to the lengthscales 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 http://www.dealii.org/) for everything that has to do with finite elements, geometries, meshes, etc.; and, through DEAL.II on Trilinos (see http://trilinos.sandia.gov/) for parallel linear algebra and on P4EST (see http://www.p4est.org/) for parallel mesh handling.

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 7.1 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.

Find wel
addresses t
mailing lists

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 CIG!

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 = {submitted},
  year =    {2011}
}
```

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.

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:

To be finished
Wouldn't th
last term nee
to have a
minus sign
drho/dT i
negative...

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

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

$$\rho C_p\left(\frac{\partial T}{\partial t}+\mathbf{u}\cdot\nabla T\right)-\nabla\cdot k\nabla T=\rho H + 2\eta\left(\varepsilon(\mathbf{u})-\frac{1}{3}(\nabla\cdot\mathbf{u})\mathbf{1}\right):\left(\varepsilon(\mathbf{u})-\frac{1}{3}(\nabla\cdot\mathbf{u})\mathbf{1}\right) + \frac{\partial\rho}{\partial T}T\mathbf{u}\cdot\mathbf{g} \qquad \text{in } \Omega, \quad (3)$$

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; as written, this term assumes that the the overall pressure is dominated by the hydrostatic pressure, in which case the variation of the total pressure can be expressed by gravity and density.

The equations ASPECT currently solves do not include phase change terms, see Section 8.

These equations are augmented by boundary conditions that can either be of Dirichlet-, Neumann, or tangential type on subsets of the boundary $\Gamma=\partial\Omega$:

$$\mathbf{u}=0 \qquad \text{on } \Gamma_{0,\mathbf{u}}, \quad (4)$$

$$\mathbf{n}\cdot\mathbf{u}=0 \qquad \text{on } \Gamma_{\parallel,\mathbf{u}}, \quad (5)$$

$$T=T_{\text{prescribed}} \qquad \text{on } \Gamma_{D,T}, \quad (6)$$

$$\mathbf{n}\cdot k\nabla T=0 \qquad \text{on } \Gamma_{N,T}. \quad (7)$$

Here, $\Gamma_{0,\mathbf{u}}$ corresponds to parts of the boundary on which the velocity is fixed to be zero, $\Gamma_{\parallel,\mathbf{u}}$ to parts of the boundary on which the velocity may be nonzero but must be parallel to the boundary, $\Gamma_{D,T}$ to places where the temperature is prescribed (for example at the inner and outer boundaries of the earth mantle), and finally $\Gamma_{N,T}$ 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., $\Gamma_{0,\mathbf{u}}\cup\Gamma_{\parallel,\mathbf{u}}=\Gamma$ and $\Gamma_{D,T}\cup\Gamma_{N,T}=\Gamma$.

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). 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 $\varepsilon(\mathbf{u})$. 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 Section 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* $\eta = \eta(p, T, \varepsilon(\mathbf{u}), \mathbf{x})$*:* 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* $\rho = \rho(p, T, \mathbf{x})$*:* 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 homogenous 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* $C_p = C_p(p, T, \mathbf{x})$*:* 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* $k = k(p, T, \mathbf{x})$*:* 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

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

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]).

As a rule of thumb for its order of magnitude, wikipedia quotes values of 1.83–$2.90\frac{\mathrm{W}}{\mathrm{m}\cdot\mathrm{K}}$ for sandstone and 1.73–$3.98\frac{\mathrm{W}}{\mathrm{m}\cdot\mathrm{K}}$ 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 intrinsic specific heat production $H = H(\mathbf{x})$:* Units $\frac{\mathrm{W}}{\mathrm{kg}} = \frac{\mathrm{m}^2}{\mathrm{s}^3}$.

  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 $\gamma = 7.4 \cdot 10^{-12}\frac{\mathrm{W}}{\mathrm{kg}}$.

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., is 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 uses this non-dimensional form; for example, the SolCx, SolKz and inclusion benchmarks in Section 6.3.1, 6.3.2 and 6.3.3 clearly do so. In such cases, 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:

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

- 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 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, using the correct (but distinct) conversion factors for each of these quantities, is a challenge and a 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 consequency 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 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

[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.

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:

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

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 inadvertant 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 where 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.

Rest to be written; we have a scheme in mind but haven't implemented it yet.

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 7.1.2) 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 5.17. 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 $p_{\text{ad}}(z), T_{\text{ad}}(z)$ that satisfy adiabatic conditions:

$$\rho C_p \frac{\mathrm{d}}{\mathrm{d}z} T_{\text{ad}}(z) = \frac{\partial \rho}{\partial T} T_{\text{ad}}(z) g_z, \tag{8}$$

$$\frac{\mathrm{d}}{\mathrm{d}z} p_{\text{ad}}(z) = \rho g_z, \tag{9}$$

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 $p_{\text{ad}}(0)$ equal to the average surface pressure (often chosen to be zero, see Section 2.5), and an adiabatic surface temperature $T_{\text{ad}}(0)$ 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 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.

2.8 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.

2.8.1 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):

$$-\nabla \cdot [2\eta\varepsilon(\mathbf{u})] + \nabla p = \rho\mathbf{g} \qquad \text{in } \Omega, \tag{10}$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega, \tag{11}$$

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T\right) - \nabla \cdot k\nabla T = \rho H + 2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u}) \qquad \text{in } \Omega, \tag{12}$$

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 5.19 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.

2.8.2 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]

$$-\nabla \cdot [2\eta\varepsilon(\mathbf{u})] + \nabla p_d = -\beta\rho_0 T\mathbf{g} \qquad \text{in } \Omega, \tag{13}$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in } \Omega, \tag{14}$$

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T\right) - \nabla \cdot k\nabla T = \rho H + 2\eta\varepsilon(\mathbf{u}) : \varepsilon(\mathbf{u}) \qquad \text{in } \Omega, \tag{15}$$

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 hydrastatic pressure $p_s = -\rho_0(1 + \beta T_0)\varphi$. Note that the right hand side forcing term in (13) is now only the deviation of the gravitational force from the force that would act if the material were at temperature T_0.

[4] Note, however, that ASPECT does not solve the equations in the form given in (13)–(15). Rather, it takes the original form with the real density, not the variation of the density. That said, you can use the formulation (13)–(15) by implementing a material model (see Section 7.1.1) in which the density in fact has the form $\rho(T) = \beta\rho_0 T$ even though this is not physical.

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 (15). 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].

Note: In ASPECT 0.1, using the IMPES scheme is the only available option. However, in later versions we will implement a fully nonlinear scheme that treats the equations as coupled, and one will be able to choose between the two variants using a run-time parameter.

3 Installation

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

3.1 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.sandia.gov/. At the current time we recommend Trilinos Version 10.4.x.[5] For installation instructions see the deal.II README file on installing Trilinos and PETSc. 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 features of DEAL.II that have been developed after the 7.1 release. Since at the time of writing, DEAL.II 7.2 has not yet appears, we currently require the development version of DEAL.II, which can be obtained by running

   ```
   svn checkout http://www.dealii.org/svn/dealii/trunk/deal.II
   ```

   Once DEAL.II 7.2 is available, this will suffice as well.

   You may want to set the environment variable[6] `DEAL_DIR` to the directory where you checked out DEAL.II.

4. *Configuring and compiling* DEAL.II*:* Now it is time to configure DEAL.II. To this end, follow the DEAL.II installation instructions. Remember to point the `./configure` script to the paths where you installed p4est and Trilinos and make sure you use MPI compilers. A typical command line would look like this:

[5] There are newer versions of Trilinos, but at least Trilinos 10.6.x and 10.8.x have bugs that make these versions unusable for our purpose.

[6] For bash this would be adding the line `export DEAL_DIR=/path/to/deal.ii/` to the file `~/.bashrc`.

```
./configure CXX=mpicxx --enable-mpi --disable-threads \
   --with-trilinos=/u/username/bin/trilinos-10.4.2 \
   --with-p4est=/u/username/bin/p4est-0.3.3.8
```

if the Trilinos and P4EST packages have been installed in the subdirectory `/u/username/bin`. 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 all`. If you have multiple processor cores, feel free to do `make all -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_DIR/examples/step-32`. Compile by running `make` and run with `mpirun -n 2 ./step-32`.

3.2 Obtaining Aspect and initial configuration

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

```
svn checkout http://dealii.org/svn/aspect/trunk/aspect
```

If `$DEAL_DIR` points to your DEAL.II installation, there is no further configuration that needs to be done, otherwise you need to edit the `Makefile` accordingly.

3.3 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 `lib/` subdirectory and will be named `lib/aspect-2d` or `lib/aspect-3d`, depending on the space dimension you compile for (see Section 4.2 for more information on this). 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

```
make doc
```

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.

4 Running Aspect

4.1 Overview

After compiling ASPECT as described above, you should have an executable file in the `lib/` subdirectory. It can be called as follows:

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

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

```
mpirun -np 32 ./lib/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. When you download ASPECT, you should already have a sample input file in the top-level directory that gives you an idea of the parameters that can be set. A full description of all parameters is given in Section 5.

Running the program should 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

*** 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:

```
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 5.11 and 5.13 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 5.32). 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, it's 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------- 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[7] as that is widely

[7] 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.

used by a number of excellent visualization packages and also supports parallel visualization.[8] 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 file of this kind, `solution.pvd` is a file that describes to Paraview which `solution-xxxx.pvtu` jointly form a complete simulation by listing the `.pvtu` files of all timesteps together with the simulation time to which they correspond. To visualize an entire simulation, not just a single time step, it is therefore simplest to just load this `solution.pvd` file.

The final file of this kind, `solution.pvd` is a file that describes to Paraview which `solution-xxxx.pvtu` jointly form a complete simulation by listing the `.pvtu` files of all timesteps together with the simulation time to which they correspond. To visualize an entire simulation, not just a single time step, it is therefore simplest to just load this `solution.pvd` file.

For more on visualization, see Section 4.4.

- *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.

[8] 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 5.35.

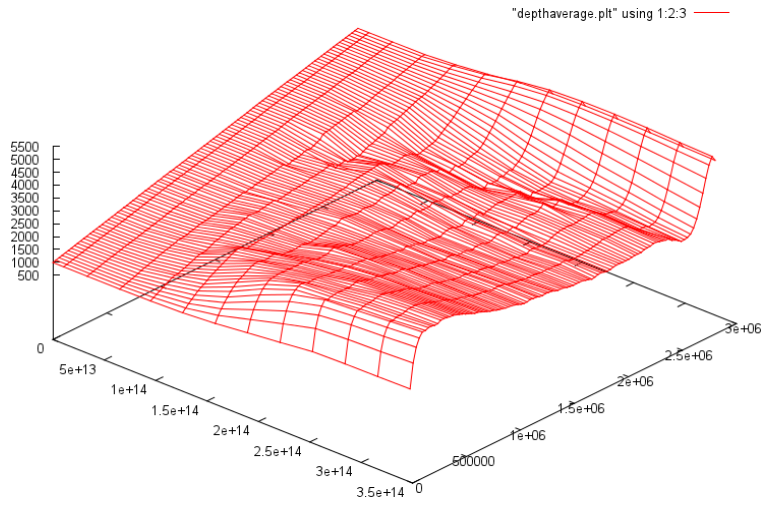


Figure 1: *Example output for depth average statistics. On the left axis are 13 time steps, on the right is the depth (from the top at 0 to the bottom of the mantle on the far right), and the upwards pointing axis is the average temperature. This plot is created by calling* `splot "depthaverage.plt" using 1:2:3 with lines` *in gnuplot.*

> **Note:** As noted in Section 2.1, ASPECT can be thought to compute in the meter-kilogram-second (MKS, or SI) system. Unless otherwise noted, the quantities in the output file are therefore also in MKS units.

- *Depth average statistics:* Similar to the `output/statistics` file, Aspect can generate depth-average statistics into `output/depthaverage.plt`. This is done by the "depth average" postprocessor and the user can control how often this file is updated.

  The data is written in text format that can be easily displayed by e.g. gnuplot. For an example, see Figure 1. The plot shows how an initially linear temperatur 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 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.

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 editing the top of the `Makefile` and following the steps to build and run in Section 3.3; alternatively, you can just build the entire application using the command `make debug-mode=off`.

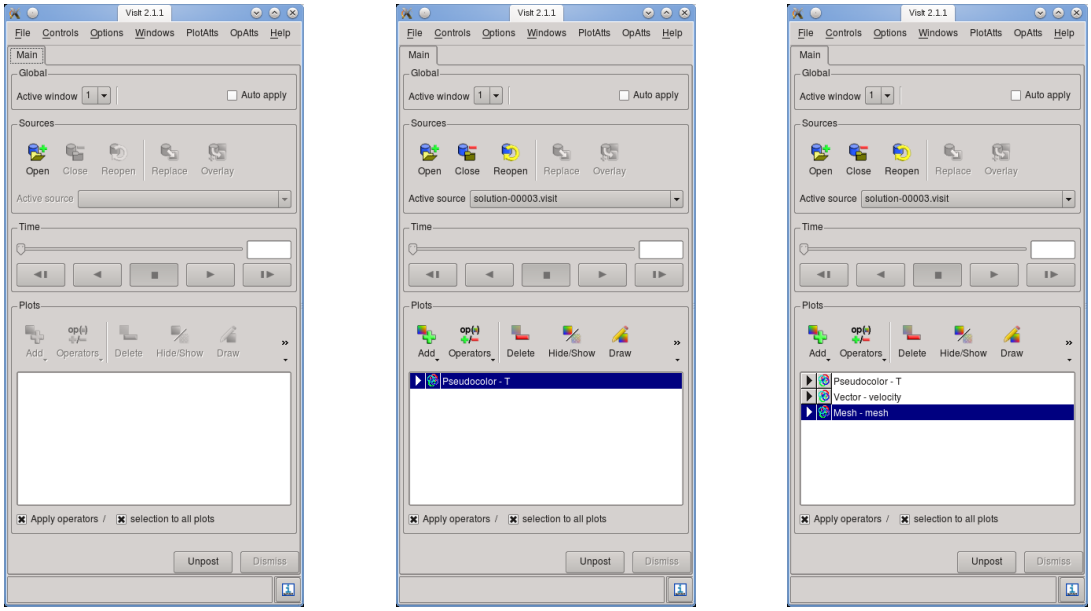
> **Note:** It goes without saying that if you make significant modifications to the program, you should do the first runs in debug mode to verify that your program still works as expected.

4.4 Visualizing results

Among the postprocessors that can be selected in the input parameter file (see Sections 4.1 and 5.35) 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 7.1.8).

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 `http://www.llnl.gov/visit/`) and ParaView (see `http://www.paraview.org/HTML/Index.html`). 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



(a) (b) (c)

Figure 2: *Main window of Visit, illustrating the different steps of adding content to a visualization.*

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 5.35. 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.

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 6.1.1 for the input file for this particular model). We can run the program with 4 processors using

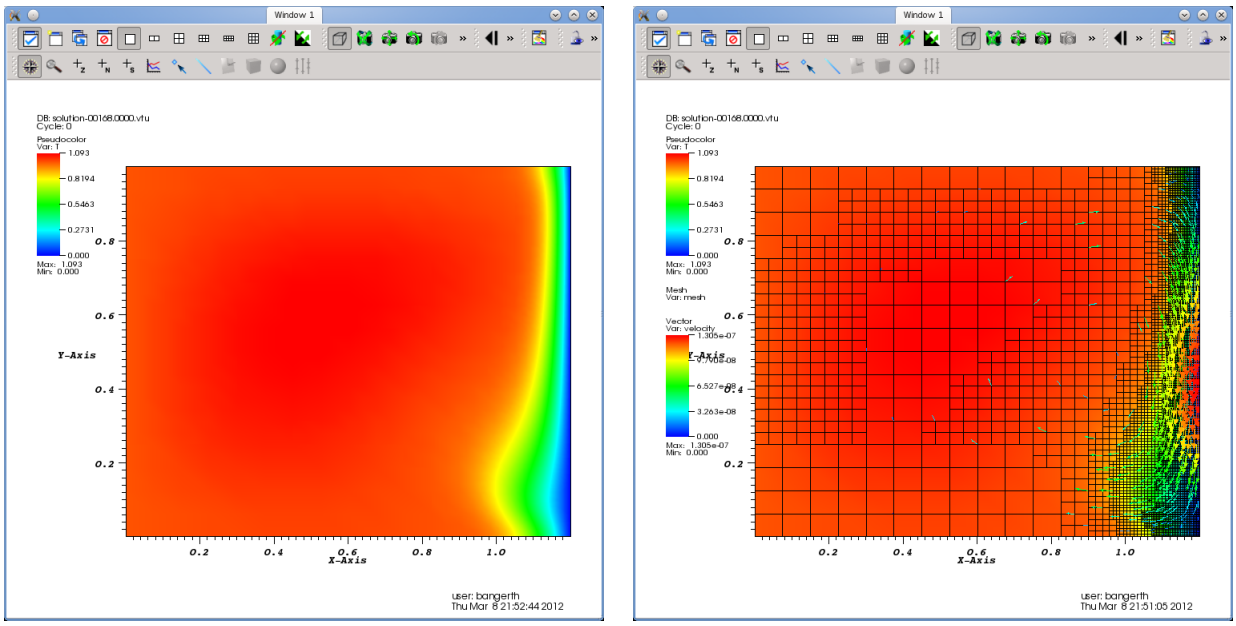
```
mpirun -np 4 ./lib/aspect-2d box.prm
```

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:[9]

- *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).

[9]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 3: *Display window of Visit, showing a single plot and one where different data is overlaid.*

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. 2(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 5.35). 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) "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.

  Screenshot would be nice

  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. 2(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. 3(a) in the display window of Visit.

  Create screen shot

- *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. 2(c) and the main view as in Fig. 3(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. 3, 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 topis can be found in the Visit documentation, see http://www.llnl.gov/visit/.

4.4.2 Visualizing statistical data

To be written

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 5.35. 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 50 time steps 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 non-sensical results after some 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 74.

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 with entries that are unknown simply being ignored. Rather, if your input file contains a line setting a parameter that is unknown to something, 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.

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 5.9), some details about the stabilization (Section 5.10), and how adaptive mesh refinement is supposed to work (Section 5.30).

- 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 5.31 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. These are, together with the sections in which their parameters are declared:

- The material model: Sections 5.19, 5.21.
- The geometry: Sections 5.11, 5.13.
- The gravity description: Sections 5.14, 5.15.
- Initial conditions for the temperature: Sections 5.17, 5.18.
- Temperature boundary conditions: Sections 5.3, 5.5.
- Postprocessors: Sections 5.32, 5.35.

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.

5.2 Global parameters

- *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:* `Dimension`

  Value: 2

  Default: 2

  Description: The number of space dimensions you want to run this program in.

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

- *Parameter name:* `End time`

  Value: 20

  Default: 1e8

  Description: The end 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:* `Linear solver tolerance`

  Value: 1e-7

  Default: 1e-7

  Description: A relative tolerance up to which linear systems in each time or nonlinear step should be solved. The absolute tolerance will then be the norm of the right hand side of the equation times this tolerance. 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. However, for cases where the static pressure is much larger than the dynamic one, it may be necessary to choose a smaller value.

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

- *Parameter name:* `Nonlinear iteration`

  Value: false

  Default: false

  Description: A flag indicating whether the Stokes+Advection equation should be solved once per time step (false) or resolved using a fixed-point iteration (true).

  Possible values: [Bool]

- *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).

  Possible values: [Bool]

- *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.

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

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

  Value: false

  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.

  Possible values: [Bool]

5.3 Parameters in section `Boundary temperature model`

- *Parameter name:* `Model name`

  Value: box

  Default:

  Description: Select one of the following models:

  'ConvBox2D': A model in which the temperature is chosen constant on the top and bottom sides of a box.

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

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

  'Tan Gurnis': A model for the Tan/Gurnis benchmark.

  Possible values: [Selection ConvBox2D—box—spherical constant—Tan Gurnis]

5.4 Parameters in section `Boundary temperature model/Convection Box 2D`

- *Parameter name:* `Bottom temperature`

  Value: 6000

  Default: 6000

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

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

5.5 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)]

5.6 Parameters in section `Boundary velocity model`

5.7 Parameters in section `Boundary velocity model/Function`

- *Parameter name:* `Function constants`

  Value: pi=3.1415926

  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: if(x¿1+sin(0.5*pi*t) , 1 , -1); 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 fparser library.

  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]

5.8 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)]

5.9 Parameters in section `Discretization`

- *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 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]

5.10 Parameters in section `Discretization/Stabilization parameters`

- *Parameter name:* `alpha`

  Value: 2

  Default: 2

  Description: The exponent α in the entropy viscosity stabilization. Units: None.

  Possible values: [Double 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 0.052 and 0.078 for 3d. Units: None.

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

- *Parameter name:* `cR`

  Value: 0.11

  Default: 0.11

  Description: The c_R factor in the entropy viscosity stabilization. Units: None.

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

5.11 Parameters in section `Geometry model`

- *Parameter name:* `Model name`

  Value: box

  Default:

  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".

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

  Possible values: [Selection box—spherical shell]

5.12 Parameters in section `Geometry model/Box`

- *Parameter name:* `X extent`
  Value: 2
  Default: 1
  Description: Extent of the box in x-direction. Units: m.
  Possible values: [Double 0...1.79769e+308 (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:* `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)]

5.13 Parameters in section `Geometry model/Spherical shell`

- *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)]

5.14 Parameters in section `Gravity model`

- *Parameter name:* `Model name`

  Value: vertical

  Default:

  Description: Select one of the following models:

  '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.

  '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 radial constant—radial earth-like—vertical]

5.15 Parameters in section `Gravity model/Radial constant`

- *Parameter name:* `Magnitude`

  Value: 30

  Default: 30

  Description: Magnitude of the gravity vector in m/s^2. The direction is always radially outward from the center of the earth.

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

5.16 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)]

5.17 Parameters in section `Initial conditions`

- *Parameter name:* `Model name`

  Value: perturbed box

  Default:

  Description: Select one of the following models:

  '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.

  'spherical hexagonal perturbation': An initial temperature field in which the temperature is perturbed following a six-fold pattern in angular direction from an otherwise spherically symmetric state.

  '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'.

Possible values: [Selection perturbed box—spherical hexagonal perturbation—spherical gaussian perturbation]

5.18 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)]

5.19 Parameters in section `Material model`

- *Parameter name:* `Model name`

  Value: simple

  Default:

  Description: Select one of the following models:

  'SolCx': A material model that corresponds to the 'SolCx' benchmark defined in Duretz et al., G-Cubed, 2011.

  'SolKz': A material model that corresponds to the 'SolKz' benchmark defined in Duretz et al., G-Cubed, 2011.

  'Inclusion': A material model that corresponds to the 'Inclusion' benchmark defined in Duretz et al., G-Cubed, 2011.

  'simple': A simple material model that has constant values for all coefficients but the density. This model uses the formulation that assumes an incompressible medium despite the fact that the density follows the law $\rho(T) = \rho_0(1-\beta(T-T_{\text{ref}})$. The value for the components of this formula and additional parameters are read from the parameter file in subsection 'Simple model'.

  'Steinberger': lookup from the paper of Steinberger/Calderwood

  'table': A material model that reads tables of pressure and temperature dependent material coefficients from files. The default values for this model's runtime parameters use a material description taken from the paper *Complex phase distribution and seismic velocity structure of the transition zone: Convection model predictions for a magnesium-endmember olivinepyroxene mantle* by Michael H.G. Jacobs and Arie P. van den Berg, Physics of the Earth and Planetary Interiors, Volume 186, Issues 12, May 2011, Pages 3648. See http://www.sciencedirect.com/science/article/pii/S0031920111000422.

  'Tan Gurnis': A simple compressible material model based on a benchmark from the paper of Tan/Gurnis (2007). This does not use the temperature equation, but has a hardcoded temperature.

  Possible values: [Selection SolCx—SolKz—Inclusion—simple—Steinberger—table—Tan Gurnis]

5.20 Parameters in section `Material model/Inclusion`

- *Parameter name:* `Viscosity jump`

  Value: 1e3

  Default: 1e3

  Description: Viscosity in the Inclusion.

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

5.21 Parameters in section `Material model/Simple 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. Units: K.

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

- *Parameter name:* `Thermal conductivity`

  Value: 1e-6

  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: 1

  Default: 5e24

  Description: The value of the constant viscosity. Units: $kg/m/s$.

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

5.22 Parameters in section `Material model/SolCx`

- *Parameter name:* `Background density`

  Value: 0

  Default: 0

  Description: Density value upon which the variation of this testcase is overlaid. Since this background density is constant it does not affect the flow pattern but it adds to the total pressure since it produces a nonzero adiabatic pressure if set to a nonzero value.

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

- *Parameter name:* `Viscosity jump`

  Value: 1e6

  Default: 1e6

  Description: Viscosity in the right half of the domain.

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

5.23 Parameters in section `Material model/Table model`

- *Parameter name:* `Composition`
  Value: olixene
  Default: olixene
  Description: The Composition of the model.
  Possible values: [Anything]

- *Parameter name:* `Compressible`
  Value: true
  Default: true
  Description: whether the model is compressible.
  Possible values: [Bool]

- *Parameter name:* `ComputePhases`
  Value: false
  Default: false
  Description: whether to compute phases.
  Possible values: [Bool]

- *Parameter name:* `Gravity`
  Value: 30
  Default: 30
  Description: The value of the gravity constant.Units: m/s^2.
  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `Path to model data`
  Value: datadir
  Default: datadir
  Description: The path to the model data.
  Possible values: [DirectoryName]

- *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)]

5.24 Parameters in section `Material model/Table model/Viscosity`

- *Parameter name:* `ReferenceViscosity`
  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:* `ViscosityModel`
  Value: Exponential
  Default: Exponential
  Description: Viscosity Model
  Possible values: [Anything]
- *Parameter name:* `Viscosity increase lower mantle`
  Value: 1e0
  Default: 1e0
  Description: The Viscosity increase (jump) in the lower mantle.
  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.25 Parameters in section `Material model/Table model/Viscosity/Composite`

- *Parameter name:* `activation_energy_diffusion`
  Value: 335e3
  Default: 335e3
  Description: activation energy for diffusion creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `activation_energy_dislocation`
  Value: 540e3
  Default: 540e3
  Description: activation energy for dislocation creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `activation_volume_diffusion`
  Value: 4.0e-6
  Default: 4.0e-6
  Description: activation volume for diffusion creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `activation_volume_dislocation`
  Value: 14.0e-6
  Default: 14.0e-6
  Description: activation volume for dislocation creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `prefactor_diffusion`
  Value: 1.92e-11
  Default: 1.92e-11
  Description: prefactor for diffusion creep (1e0/prefactor)*exp((activation_energy+activation_volume*pressure)/(R*temp
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `prefactor_dislocation`
  Value: 2.42e-10
  Default: 2.42e-10
  Description: prefactor for dislocation creep (1e0/prefactor)*exp((activation_energy+activation_volume*pressure)/(R*te
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `stress_exponent`
  Value: 3.5
  Default: 3.5
  Description: stress exponent for dislocation creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.26 Parameters in section `Material model/Table model/Viscosity/Diffusion`

- *Parameter name:* `activation_energy_diffusion`
  Value: 335e3
  Default: 335e3
  Description: activation energy for diffusion creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `activation_volume_diffusion`
  Value: 4.0e-6
  Default: 4.0e-6
  Description: activation volume for diffusion creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `prefactor_diffusion`
  Value: 1.92e-11
  Default: 1.92e-11
  Description: prefactor for diffusion creep (1e0/prefactor)*exp((activation_energy+activation_volume*pressure)/(R*temp
  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.27 Parameters in section `Material model/Table model/Viscosity/Dislocation`

- *Parameter name:* `activation_energy_dislocation`
  Value: 335e3
  Default: 335e3
  Description: activation energy for dislocation creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `activation_volume_dislocation`
  Value: 4.0e-6
  Default: 4.0e-6
  Description: activation volume for dislocation creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `prefactor_dislocation`
  Value: 1.92e-11
  Default: 1.92e-11
  Description: prefactor for dislocation creep (1e0/prefactor)*exp((activation_energy+activation_volume*pressure)/(R*te
  Possible values: [Double 0...1.79769e+308 (inclusive)]
- *Parameter name:* `stress_exponent`
  Value: 3.5
  Default: 3.5
  Description: stress exponent for dislocation creep
  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.28 Parameters in section `Material model/Table model/Viscosity/Exponential`

- *Parameter name:* `exponential_P`
  Value: 1
  Default: 1
  Description: multiplication factor or Pressure exponent
  Possible values: [Double 0...1.79769e+308 (inclusive)]

- *Parameter name:* `exponential_T`
  Value: 1
  Default: 1
  Description: multiplication factor or Temperature exponent
  Possible values: [Double 0...1.79769e+308 (inclusive)]

5.29 Parameters in section `Material model/Tan Gurnis model`

- *Parameter name:* `Di`
  Value: 0.5
  Default: 0.5
  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:* `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:* `a`

  Value: 0

  Default: 0

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

- *Parameter name:* `gamma`

  Value: 1

  Default: 1

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

- *Parameter name:* `wavenumber`

  Value: 1

  Default: 1

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

5.30 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: 0

  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: 5

  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:* `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)] ¡¡¡¡¡¡¡ .working

- *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: Density c_p temperature

  Default: Density c_p temperature

  Description: The method used to determine which cells to refine and which to coarsen.

  Possible values: [Selection Temperature—Velocity—Normalized density and temperature—Weighted density and temperature—Density c_p temperature]

- *Parameter name:* `Time steps between mesh refinement`

  Value: 0

  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)]

5.31 Parameters in section `Model settings`

- *Parameter name:* `Fixed temperature boundary indicators`

  Value: 0, 1

  Default:

  Description: A comma separated list of integers 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.

  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 ¡[Integer range 0...2147483647 (inclusive)]¿ 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.

  Possible values: [Bool]

- *Parameter name:* `Include shear heating`

  Value: true

  Default: true

  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.

  Possible values: [Bool]

- *Parameter name:* `Prescribed velocity boundary indicators`

  Value: 3: function

  Default:

  Description: A comma separated list denoting those boundaries on which the velocity is tangential but prescribed, i.e., where 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: value1, key2: value2, key3: value3, ..." where each key must be a valid boundary indicator and each value must be one of the currently implemented boundary velocity models.

  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.

  Possible values: [Map map of ¡[Integer range 0...255 (inclusive)]:[Selection inclusion—function—zero velocity]¿ of length 0...4294967295 (inclusive)]

- *Parameter name:* `Radiogenic heating rate`

  Value: 0e0

  Default: 0e0

  Description: H0

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

- *Parameter name:* `Tangential velocity boundary indicators`

  Value:

  Default:

Description: A comma separated list of integers 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).

Possible values: [List list of ¡[Integer range 0...255 (inclusive)]¿ of length 0...4294967295 (inclusive)]

- *Parameter name:* `Zero velocity boundary indicators`

  Value: 0, 1, 2

  Default:

  Description: A comma separated list of integers denoting those boundaries on which the velocity is zero.

  Possible values: [List list of ¡[Integer range 0...255 (inclusive)]¿ of length 0...4294967295 (inclusive)]

5.32 Parameters in section `Postprocess`

- *Parameter name:* `List of postprocessors`

  Value: visualization, velocity statistics, temperature statistics, heat flux statistics

  Default: all

  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:

  'depth average': A postprocessor that computes depth averaged quantities and writes them out.

  'DuretzEtAl error': A postprocessor that compares the solution of the benchmarks from the Duretz et al., G-Cubed, 2011, paper with the one computed by ASPECT and reports the error. Specifically, it can compute the errors for the SolCx, SolKz and inclusion benchmarks. The postprocessor inquires which material model is currently being used and adjusts which exact solution to use accordingly.

  'heat flux statistics': A postprocessor that computes some statistics about the heat flux across boundaries.

  'heat flux statistics for the table model': A postprocessor that computes some statistics about the heat flux across boundaries.

  'velocity statistics for the table model': A postprocessor that computes some statistics about the velocity field.

  'Tan Gurnis error': A postprocessor that compares the solution of the benchmarks from the Tan/Gurnis (2007) paper with the one computed by ASPECT by outputing data that is compared using a matlab script.

  'temperature statistics': A postprocessor that computes some statistics about the temperature field.

  'tracers': Postprocessor that propagates tracer particles based on the velocity field.

  'velocity statistics': A postprocessor that computes some statistics about the velocity field.

  '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 depth average—DuretzEtAl error—heat flux statistics—heat flux statistics for the table model—velocity statistics for the table model—Tan Gurnis error—temperature statistics—tracers—velocity statistics—visualization—all]

5.33 Parameters in section `Postprocess/Depth average`

- *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)]

5.34 Parameters in section `Postprocess/Tracers`

- *Parameter name:* `Data output format`

  Value: none

  Default: none

  Description: File format to output raw particle data in.

  Possible values: [Selection none—ascii—vtu—hdf5]

- *Parameter name:* `Number of tracers`

  Value: 1e3

  Default: 1e3

  Description: Total number of tracers to create (not per processor or per element).

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

- *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)]

5.35 Parameters in section `Postprocess/Visualization`

- *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:

  'density': A visualization output object that generates output for the density.

'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 $\eta[\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{tr } \varepsilon(\mathbf{u}))\mathbf{I}] : [\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{tr } \varepsilon(\mathbf{u}))\mathbf{I}]$.

'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.

'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.

'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 $\sqrt{[\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{tr } \varepsilon(\mathbf{u}))\mathbf{I}] : [\varepsilon(\mathbf{u}) - \frac{1}{3}(\text{tr } \varepsilon(\mathbf{u}))\mathbf{I}]}$ in the compressible case.

'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.

'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 density—friction heating—nonadiabatic pressure—nonadiabatic temperature—partition—seismic vp—seismic vs—specific heat—strain rate—thermodynamic phase—viscosity—viscosit ratio—all]

- *Parameter name:* `Number of grouped files`

  Value: 0

  Default: 0

  Description: VTU file output supports grouping files from several CPUs into one file 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 in a background thread. A value of 1 will generate one big file containing the whole solution.

  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:* `Time between graphical output`

  Value: 0.1

  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)]

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. 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 6.1), setups that try to model geophysical situations (Section 6.2) and setups that are used to benchmark ASPECT to ensure correctness or to test accuracy of our solvers (Section 6.3).

Note: The input files discussed in the following sections can generally be found in the `cookbooks/` directory of your ASPECT installation.

6.1 Simple setups

6.1.1 Convection in a box

In this first example, let us consider a simple situation: a 2d box of dimensions $[0, 1.2] \times [0, 1]$ where we keep the velocity at the left, bottom and right boundaries fixed at zero, but allow free slip along the right boundary (in other words, along the right boundary we have $\mathbf{u} \cdot \mathbf{n} = 0$, but the tangential component of $\mathbf{u}$ is unconstrained). Furthermore, we assume that the temperature is one along the left, bottom and top boundaries, but zero along the right boundary. This results in a situation where the cold material along the right wall has negative buoyancy, drops down and creates a convection roll in the box. The fact that the domain has different extents in the x- and y-directions complicates the resulting flow slightly.

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):

```
################ Global parameters
```

6.1.2 Convection in a box with prescribed, variable velocity boundary conditions

A similarly simple setup is to equip the model we had in the previous section 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 to use boundary conditions at the top that describe experimentally determined velocities of plates. This cookbook shows a simple version of something like this.

Like for many other things, ASPECT has a set of plugins for prescribed velocity boundary values (see Sections 5.6 and 7.1.5). 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.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   = 0, 1
  set Zero velocity boundary indicators       = 0, 1, 2
  set Tangential velocity boundary indicators =
  set Prescribed velocity boundary indicators = 3: function
end
```

Following the convention described above, this means that we prescribe a fixed temperature at the left and right sides of the box (boundary numbers zero and one). We fix the velocity to zero at the left, right, and bottom boundaries. No boundary will use tangential flow. 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.

6.2 Geophysical setups

To be written

6.3 Benchmarks

Benchmarks are used to verify that a solver solves the problem correctly, i.e., to *verify* correctness of a code.[10] 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 these benchmarks are also presented in [KHB12] in much more detail than shown here.

6.3.1 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

$$\eta(\mathbf{x}) = \begin{cases} 1 & \text{for } x_1 \le 0.5, \\ 10^6 & \text{for } x_1 > 0.5. \end{cases}$$

[10] 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.

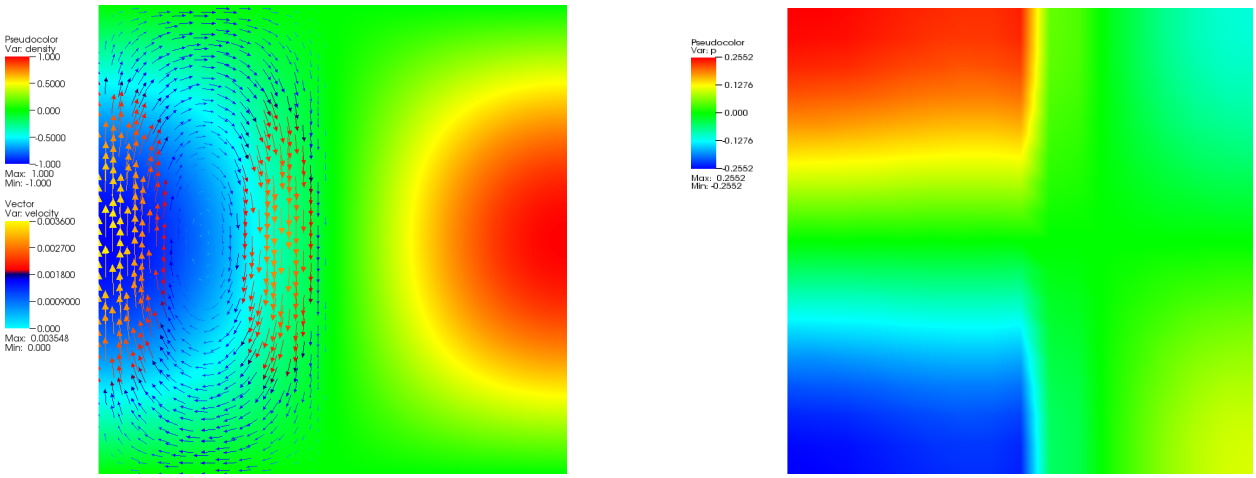


Figure 4: SolCx Stokes benchmark. Left: The density perturbation field and overlaid to it some velocity vectors. The viscosity is very large in the right hand, leading to a stagnant flow in this region. Right: The pressure on a relatively coarse mesh, showing the internal layer along the line where the viscosity jumps.

This strongly discontinuous viscosity field yields an almost stagnant flow in the right half of the domain and 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. 4.

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:

```
################ 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 = 0,1,2,3
  set Zero velocity boundary indicators       =
end


subsection Material model
  set Model name = SolCx

  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 the what to do with the solution

subsection Postprocess
  set List of postprocessors = DuretzEtAl error, 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 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 5.11). 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 5.14).

- 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 5.9 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> ../lib/aspect sol_cx.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. 4. 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].

6.3.2 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 this benchmark:

```
################ 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 = 0,1,2,3
  set Zero velocity boundary indicators       =
end


subsection Material model
  set Model name = SolKz
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
```

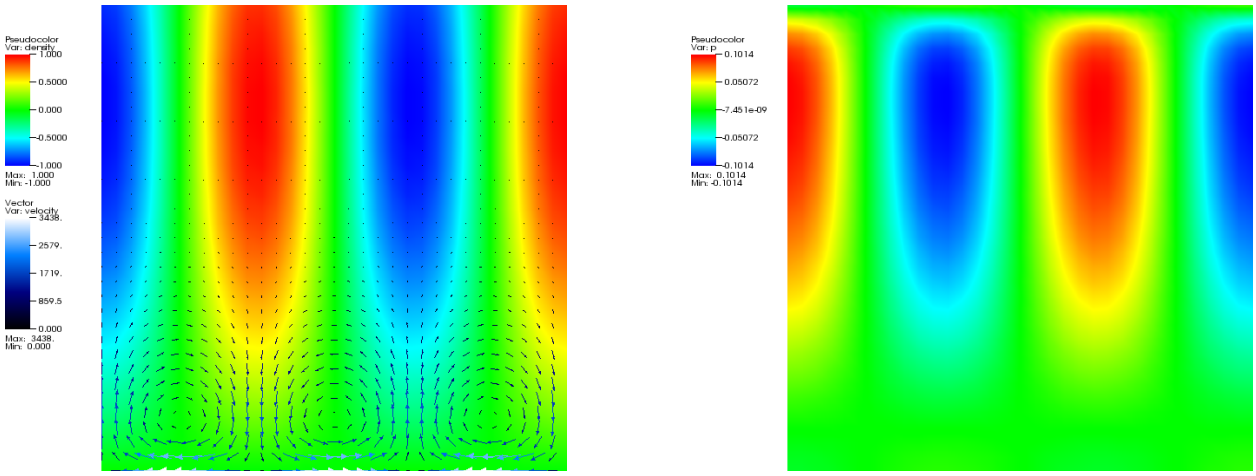


Figure 5: SolKz Stokes benchmark. Left: The density perturbation field and overlaid to it some velocity vectors. The viscosity grows exponentially in the vertical direction, leading to small velocities at the top despite the large density variations. Right: The pressure.

```
subsection 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 the what to do with the solution

subsection Postprocess
  set List of postprocessors = DuretzEtAl error, 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. 5.

6.3.3 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. 6. 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].

As before, the benchmark can be run with a small variation of the input files already discussed above:

```
################ Global parameters
```

Revisit this once we have the machinery in place to choose nonzero boundary conditions in a more elegant way.

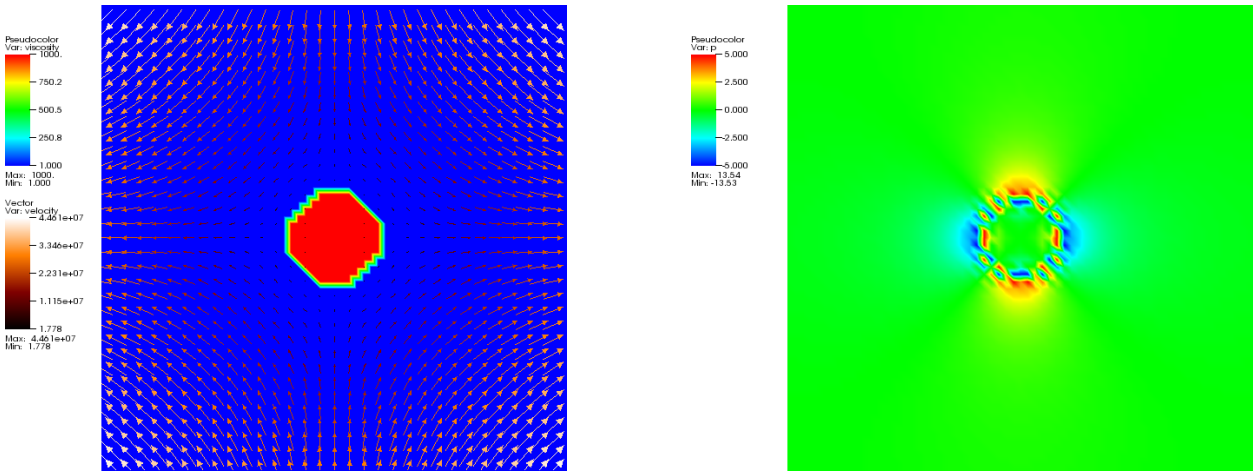


Figure 6: Inclusion Stokes benchmark. Left: The viscosity field when interpolated onto the mesh (internally, the "exact" viscosity field – large inside a circle, small outside – is used), and overlaid to it some velocity vectors. Right: The pressure with its oscillations along the interface. The oscillations 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 = 0,1,2,3
  set Tangential velocity boundary indicators =
  set Zero velocity boundary indicators       =
end


subsection Material model
  set Model name = Inclusion

  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 the what to do with the solution

subsection Postprocess
  set List of postprocessors = DuretzEtAl error, visualization
end
```

7 Extending Aspect

Aspect is designed to be an extensible code. In particular, the program uses a plugin architecture in which it is trivial to replace or extend certain components of the program:

- 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.

We will discuss the way this is achieved in Section 7.1. 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 7.2.

In either of these two cases, you will need to extend the source code of the program. 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 http://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 http://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 http://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 http://www.dealii.org/developer/doxygen/deal.II/step_31.html and http://www.dealii.org/developer/doxygen/deal.II/step_32.html from which ASPECT directly descends.
- The DEAL.II Frequently Asked Questions at http://dealii.sourceforge.net/index.php/Deal.II_Questions_and_Answers 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 http://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 http://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 http://dealii.sourceforge.net/index.php/Deal.II_Questions_and_Answers 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 Materials, geometries, gravitation and other aspects of the model: plugins

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 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 rest of the program only communicates with material models, geometry descriptions, etc., through a simple and very basic interface. These interfaces are declared in the files `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.[11]

  Each of the categories above that allow plugins have several implementations of their respective interfaces that you can use to get an idea 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`. 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

[11] 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.

the purpose of the class (see, for example, Section 5.14 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 should just work automatically. Note also that 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.

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.

7.1.1 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`.

Specifically, your new class needs to implement the basic interface:

```
template <int dim>
class aspect::MaterialModel::Interface
{
  public:
    // Physical parameters used in the basic equations
    virtual double viscosity (const double                   temperature,
                              const double                   pressure,
                              const SymmetricTensor<2,dim> &strain_rate,
                              const Point<dim>             &position) const = 0;

    virtual double density (const double      temperature,
                            const double      pressure,
                            const Point<dim> &position) const = 0;

    virtual double compressibility (const double temperature,
                                    const double pressure,
                                    const Point<dim> &position) const = 0;

    virtual double specific_heat (const double      temperature,
                                  const double      pressure,
                                  const Point<dim> &position) const = 0;

    virtual double thermal_expansion_coefficient (const double      temperature,
                                                  const double      pressure,
                                                  const Point<dim> &position) const;

    virtual double thermal_conductivity (const double temperature,
                                         const double pressure,
                                         const Point<dim> &position) const
                                         = 0;

    // Qualitative properties one can ask a material model
    virtual bool
    viscosity_depends_on (const NonlinearDependence::Dependence dependence) const = 0;
```

```
  virtual bool
  density_depends_on (const NonlinearDependence::Dependence dependence) const = 0;

  virtual bool
  compressibility_depends_on (const NonlinearDependence::Dependence dependence) const = 0;

  virtual bool
  specific_heat_depends_on (const NonlinearDependence::Dependence dependence) const = 0;

  virtual bool
  thermal_conductivity_depends_on (const NonlinearDependence::Dependence dependence) const = 0;

  virtual bool is_compressible () const = 0;

  // Partial derivatives of physical parameters
  virtual double
  viscosity_derivative (const double               temperature,
                        const double               pressure,
                        const Point<dim>          &position,
                        const NonlinearDependence::Dependence dependence) const;

  virtual double
  density_derivative (const double               temperature,
                      const double               pressure,
                      const Point<dim>          &position,
                      const NonlinearDependence::Dependence dependence) const;

  virtual double
  compressibility_derivative (const double               temperature,
                              const double               pressure,
                              const Point<dim>          &position,
                              const NonlinearDependence::Dependence dependence) const;

  virtual double
  specific_heat_derivative (const double               temperature,
                            const double               pressure,
                            const Point<dim>          &position,
                            const NonlinearDependence::Dependence dependence) const;

  virtual double
  thermal_conductivity_derivative (const double               temperature,
                                   const double               pressure,
                                   const Point<dim>          &position,
                                   const NonlinearDependence::Dependence dependence) const;

  // 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;

  virtual
  double
  seismic_Vs (const double      temperature,
              const double      pressure) const;
  virtual
  unsigned int
```

```
    thermodynamic_phase (const double       temperature,
                         const double       pressure) const;

    // Functions used in dealing with run-time parameters
    static
    void
    declare_parameters (ParameterHandler &prm);

    virtual
    void
    parse_parameters (ParameterHandler &prm);
};
```

Here, the first set of functions refer to the coefficients η, C_p, k, ρ in equations (1)–(3), each as a function of temperature, pressure, position and, in the case of the viscosity, the strain rate. Implementations of these methods may of course choose to ignore dependencies on any of these arguments. The second set of functions describes the nonlinear dependence of the various coefficients on pressure, temperature, or strain rate, and the next block then provides the numerical values of these dependencies. 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 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 2.8.1 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 contuity equation as $\nabla \cdot (\rho \mathbf{u}) = 0$ (compressible Stokes) or as $\nabla \cdot \mathbf{u} = 0$ (incompressible Stokes).

The purpose of the last two functions has been discussed in the general overview of plugins above.

7.1.2 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 to describe to the rest of the code which parts of the boundary represent Dirichlet-type (fixed temperature) or Neumann-type (no heat flux) boundaries for the temperature, and where the velocity is considered zero or tangential to the boundary. This information is encoded in functions that return which boundary indicators represent these types of boundaries; in DEAL.II, a boundary indicator is a number attached to each piece of the boundary that can be used to represent the type of boundary a piece belongs to.

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;

        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.

7.1.3 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;

        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 but is likely already obvious. The purpose of the last two functions has been discussed in the general overview of plugins above.

7.1.4 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:
    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.

7.1.5 Prescribed velocity boundary conditions

To be written

7.1.6 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 7.1.2 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.

7.1.7 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 7.1.8.
- 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,[12] 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.
- *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

[12] 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.

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 `aspect::Postprocess::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.

- 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.

Give a simple example, e.g. evaluating a point value.

7.1.8 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 5.35).

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 solution such as access to the material models. 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 and consequently is not derived from the aspect::Postprocess::SimulatorAccess class.[13] In addition to these two classes, visualization plugins do their actual work by implementing the interface of the

[13] 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 $\sqrt{[\varepsilon(\mathbf{u}) - \frac{1}{3}(\mathrm{tr}\ \varepsilon(\mathbf{u}))\mathbf{I}] : [\varepsilon(\mathbf{u}) - \frac{1}{3}(\mathrm{tr}\ \varepsilon(\mathbf{u}))\mathbf{I}]}$ 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.

`dealii::DataPostprocessor` class, possibly through the simpler to use `dealii::DataPostprocessorScalar` or `dealii::DataPostprocessorVector` classes.

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 overriden in a plugin. Specifically, the following functions exist:

```
class Interface
{
  public:
    virtual ~Interface();

    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::DataPostprocessorScal` 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`.

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.

7.2 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 `http://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 `http://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.

- *Incorporating latent heat effects:* Real materials undergo phase transitions at certain pressures and temperatures, and these phase transitions release or take up energy (i.e., heat). The terms that need to be added to the temperature equation (3) are not very difficult but one needs a description of the latent heat based on the Clapeyron slope as a function of temperature and pressure [CY85, STO01], which we currently don't have. If someone contributes such a description we'll be happy to add the relevant terms into the model.

- *Prescribing boundary velocities:* One might want to prescribe the velocity at the outer boundary, for example using GPS-inferred plate velocities.

- *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 7.1 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 resouces:

- For questions on the source code of Aspect, portability, installation, etc., use the ASPECTdevelopment mailing list at aspect-devel@geodynamics.org. Information about this mailing list is provided at http://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 http://www.dealii.org/mail.html.
- In case of more general questions about mantle convection, you can contact the CIG mantle convection mailing lists at cig-mc@geodynamics.org. Information about this mailing list is provided at http://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@math.tamu.edu.

References

[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.

[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.

[CY85] U. R. Christensen and D. A. Yuen. Layered convection induced by phase transitions. *J. Geoph. Res.*, 90:10291–10300, 1985.

[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.

[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.

[KHB12] Martin Kronbichler, Timo Heister, and Wolfgang Bangerth. High accuracy mantle convection simulation through modern numerical methods. *Geophysics Journal International*, 191:12–29, 2012.

[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.

[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.

[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 76 below.

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 74 above.