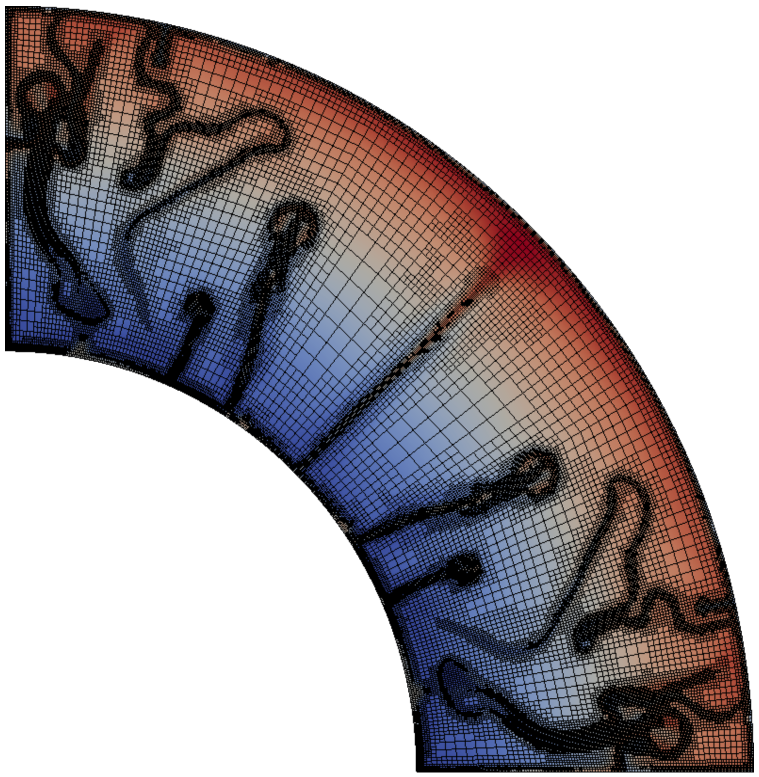

Aspect

Advanced Solver for Problems in Earth's ConvecTion

Preview release, version 0.8

Wolfgang Bangerth
Timo Heister
Martin Kronbichler

Contents

1 Introduction

To be written

discuss deal.II, Trilinos, p4est briefly

> **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!

> **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!

2 Equations, models, coefficients

2.1 Basic equations

ASPECT solves the 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

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

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

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

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 we 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 $\Gamma = \partial\Omega$:

$$\mathbf{u} = 0 \qquad \text{on } \Gamma_{0,\mathbf{u}}, \tag{4}$$

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

$$T = T_{\text{prescribed}} \qquad \text{on } \Gamma_{D,T}, \tag{6}$$

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

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.[1] Consequently, it allows the direct use of this pressure when looking up pressure dependent material parameters.

Note: 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. However, all existing implementations of models uses the SI system, i.e., they express everything in meters, kilograms and seconds – the MKS system –, as well as degrees Kelvin. It is therefore convenient to think that ASPECT actually requires the use of these units throughout and that all material parameters, geometries, etc., must also be expressed in these units.
That said, for convenience, output quantities are sometimes provided in units *centimeters per year* instead of *meters per second* (velocities) or in *years* instead of *seconds* (the current time, the time step size); however, this conversion happens at the time output is generated, and is not part of the solution process.

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.

[1]Other codes often replace this equation by $-\nabla\cdot 2\eta\nabla\mathbf{u} + \nabla p_d = (\rho-\rho_0)\mathbf{g}$ where $p_d = (p+\rho_0\varphi)$, and ϕ is the gravitational potential so that $\mathbf{g} = -\nabla\varphi$ and chosen in such a way that $\varphi = 0$ at that part of the boundary where we want the pressure to be zero (e.g., on the earth surface). Furthermore, ρ_0 is a reference density. 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. $p_d = p + \rho_0\varphi$ is then the *dynamic* pressure, i.e., the difference between total pressure and hydrostatic pressure $p_s = -\rho_0\varphi$.

While this formulation has a number of numerical advantages, it also has significant disadvantages: (i) The pressure we compute is not immediately comparable to quantities that we need to look up pressure-dependent quantities such as the density. (ii) The definition of a reference density is only simple if we have incompressible models for which the density only depends on the temperature; for more complicated models, it is not a priori clear which density ρ_0 to chose so that $p+\rho_0\varphi$ really only contains the dynamic part of the pressure. (iii) To compute the total pressure p from p_d one needs to know a gravitational potential φ that is consistent with the gravity vector $\mathbf{g}$. This is not always trivial because many simple models just prescribe a $\mathbf{g}$ for which no such potential needs to exist; for example, this is the case when using a radially inward gravitational vector of constant magnitude.

Note: The next version 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 $\mathrm{Pa} \cdot \mathrm{s} = \mathrm{kg}\frac{1}{\mathrm{m}\cdot\mathrm{s}}$.

  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 $\frac{\mathrm{kg}}{\mathrm{m}^3}$.

  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 $\rho_{\text{ref}} = 3300\frac{\mathrm{kg}}{\mathrm{m}^3}$, $T_{\text{ref}} = 293\mathrm{K}$, $\beta = 2 \cdot 10^{-5}\frac{1}{\mathrm{K}}$.

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

  Simple models assume a radially inward gravity vector of constant magnitude (e.g., the surface gravity of Earth, $9.81\frac{\mathrm{m}}{\mathrm{s}^2}$), 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 $\frac{\mathrm{J}}{\mathrm{kg}\cdot\mathrm{K}} = \frac{\mathrm{m}^2}{\mathrm{s}^2\cdot\mathrm{K}}$.

  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 $\frac{\mathrm{J}}{\mathrm{kg}\cdot\mathrm{K}}$ for granite[2] For the earth mantle, a value of 1250 $\frac{\mathrm{J}}{\mathrm{kg}\cdot\mathrm{K}}$ is within the range suggested by the literature.

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

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

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

As a rule of thumb for its order of magnitude, wikipedia quotes values of 1.83–$2.90 \frac{\mathrm{W}}{\mathrm{m}\cdot\mathrm{K}}$ for sandstone and 1.73–$3.98 \frac{\mathrm{W}}{\mathrm{m}\cdot\mathrm{K}}$ for granite.[3] The values in the mantle are almost certainly higher than this though probably not by much. The exact 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 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, BK11] 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

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

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 [KHB11] that gives a detailed description and rationale for the various building blocks.

2.4 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.

2.4.1 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., $\frac{1}{\rho}\nabla \mathbf{u} \cdot \rho \approx \|u\| 0.5/3000\text{km}$ (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.

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 5.13 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.

2.4.2 Almost linear models

To be written

3 Installation

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

3.1 Prerequisites

1. *Obtain Trilinos:* We recommend Trilinos Version 10.4.2, which can be downloaded from http://trilinos.sandia.gov. For installation instructions see the deal.II README. 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` you can find in

   ```
   $DEAL_DIR/doc/external-libs/p4est-setup.sh
   ```

   (and not according to the p4est stand-alone installation instructions).

3. *Obtain deal.II:* 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
   ```

   You may want to set the environment variable[4] `DEAL_DIR` to the directory where you checked out DEAL.II.

4. *Configure and compile* DEAL.II*:* Now it is time to configure DEAL.II. Remember to point the `./configure` script to the paths where you installed p4est and Trilinos and make sure you use MPI compilers. Example line:

[4]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=/w/trilinos-10.4.2 \
--with-p4est=/w/p4est-0.3.3.8
```

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. *Test 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 `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. 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-2d parameter-file.prm
```

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

```
mpirun -np 32 ./lib/aspect-2d 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 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 5.7 and 5.8 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 5.17). 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------- 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, Aspect generates this output in VTK format[5] as that is widely used by a number

[5]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.

of excellent visualization packages and also supports parallel visualization.[6] 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.

- *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 (cm/year)
# 8: Max. velocity (cm/year)
# 9: Minimal temperature (K)
# 10: Average temperature (K)
# 11: Maximal temperature (K)
# 12: Average nondimensional temperature (K)
# 13: Core-mantle heat flux (W)
# 14: Surface heat flux (W)
0 0.0000e+00 33 2.9543e+07 8                     "" 0.0000 0.0000   0.0000    0.0000 ...
0 0.0000e+00 34 1.9914e+07 8 output/solution-00000 0.0946 0.1829 300.0000 3007.2519 ...
1 1.9914e+07 33 1.9914e+07 8 output/solution-00001 0.1040 0.2172 300.0000 3007.8406 ...
2 3.9827e+07 33 1.9914e+07 8                     "" 0.1114 0.2306 300.0000 3008.3939 ...
```

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

Note: As noted in Section 2.1, 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.

4.2 Selecting between 2d and 3d runs

DEAL.II, upon which ASPECT is based, has a feature 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

[6]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, let the authors of ASPECT know and we can add this feature.

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.

By default, ASPECT is compiled for 2d. However, it is simple to change this: at the top of the `Makefile`, find the definition of the `deal_II_dimension` variable, set it to three, and issue the following commands

```
make clean
make
```

(or do `make -jN` where `N` is the number of processors in your machine) to obtain an executable `lib/aspect-3d`.

This executable can then be run in exactly the same way as before and, in most cases, with the same input file. Be prepared to wait much longer for computations to finish, 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

To be written

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.

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 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 5.5), some details about the stabilization (Section 5.6), and how adaptive mesh refinement is supposed to work (Section 5.15).

- 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 5.16 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 5.13, 5.14.
– The geometry: Sections 5.7, 5.8.
– The gravity description: Sections 5.9, 5.10.
– Initial conditions for the temperature: Sections 5.11, 5.12.
– Temperature boundary conditions: Sections 5.3, 5.4.
– Postprocessors: Sections 5.17, 5.18.

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:* `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:* `End time`

  Value: 2e8

  Default: 1e8

  Description: The end time of the simulation. Units: years.

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

- *Parameter name:* `Output directory`

  Value: output

  Default: output

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

  Possible values: [DirectoryName]

- *Parameter name:* `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]

5.3 Parameters in section `Boundary temperature model`

- *Parameter name:* `Model name`

  Value: box

  Default:

  Description: Select one of the following models:

  '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 'Spherical constant'.

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

  Possible values: [Selection spherical constant—box]

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

- *Parameter name:* `Inner temperature`

  Value: 6300

  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: 300

  Default: 0

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

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

5.5 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. 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: true

  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).

  Possible values: [Bool]

5.6 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: 0.5

  Default: 0.11

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

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

5.7 Parameters in section `Geometry model`

- *Parameter name:* `Model name`

  Value: box

  Default:

  Description: Select one of the following models:

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

  'box': A box geometry with fixed length 1 in each coordinate direction.

  Possible values: [Selection spherical shell—box]

5.8 Parameters in section `Geometry model/Spherical shell`

- *Parameter name:* `Inner radius`

  Value: 5698e3

  Default: 3481000

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

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

- *Parameter name:* `Opening angle`

  Value: 180

  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: 10415e3

  Default: 6336000

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

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

5.9 Parameters in section `Gravity model`

- *Parameter name:* `Model name`

  Value: vertical

  Default:

  Description: Select one of the following models:

  'vertical': A gravity model in which the gravity direction is vertically downward and at constant magnitude.

  '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.

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

5.10 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)]

5.11 Parameters in section `Initial conditions`

- *Parameter name:* `Model name`

  Value: perturbed box

  Default:

  Description: Select one of the following models:

  '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'.

  '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.

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

5.12 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: 4.71238898038468985769

  Default: 0e0

  Description: The angle where the center of the perturbation is placed.

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

- *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)]

5.13 Parameters in section `Material model`

- *Parameter name:* `Model name`

  Value: simple

  Default:

  Description: Select one of the following models:

  'table': A material model that reads tables of pressure and temperature dependent material coefficients from files.

  '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'.

  Possible values: [Selection table—simple]

5.14 Parameters in section `Material model/Simple model`

- *Parameter name:* `Reference density`

  Value: 1

  Default: 3300

  Description: Reference density ρ_0. Units: kg/m^3.

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

- *Parameter name:* `Reference temperature`

  Value: 1

  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: 1

  Default: 5e24

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

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

5.15 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.

  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: 1

  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: 4

  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)]

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

  Value: 5

  Default: 10

  Description: The number of time steps after which the mesh is to be adapted again based on computed error indicators.

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

5.16 Parameters in section `Model settings`

- *Parameter name:* `Include shear heating`

  Value: false

  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:* `Radiogenic heating rate`

  Value: 1

  Default: 0e0

  Description: H0

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

5.17 Parameters in section `Postprocess`

- *Parameter name:* `List of postprocessors`

  Value: all

  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:

  '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'.

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

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

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

  Possible values: [MultipleSelection visualization—velocity statistics—temperature statistics—heat flux statistics—all]

5.18 Parameters in section `Postprocess/Visualization`

- *Parameter name:* `Time between graphical output`

  Value: 1

  Default: 50

  Description: The time interval between each generation of graphical output files. Units: years.

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

6 Cookbooks

To be written

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.

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.[7]

  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'.");
  ```

[7] 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.

Here, the first argument to the macro is the name of the class. The second is the name by which this model can be selected in the parameter file. And the third one is a documentation string that describes the purpose of the class (see, for example, Section 5.9 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 solvers. 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 following basic interface:

```
template <int dim>
class aspect::MaterialModel::Interface
{
  public:
    virtual
    double viscosity (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_conductivity (const double temperature,
                          const double pressure,
                          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
    bool is_compressible () const = 0;
```

```
        static
        void
        declare_parameters (ParameterHandler &prm);

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

Here, the first four functions refer to the coefficients η, C_p, k, ρ in equations (1)–(3), each as a function of temperature, pressure and position. Implementations of these methods may of course choose to ignore dependencies on any of these arguments. The compressibility coefficient describes $\frac{\partial\rho}{\partial p}$.

The function `is_compressible` returns whether we should consider the material as compressible or not, see Section 2.4.1 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
        std::set<unsigned char>
        get_temperature_dirichlet_boundary_indicators () const = 0;

        virtual
        std::set<unsigned char>
        get_zero_velocity_boundary_indicators () const = 0;

        virtual
        std::set<unsigned char>
        get_tangential_velocity_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 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.

7.1.6 Postprocessors: Evaluating the solution after each time step

Postprocessors are arguably the most complex and powerful of the plugins available in Aspect. 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.
- 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,[8] 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.
- *Surface or crustal processes:* Another possibility would be to keep track of surface or crustal processes induced by mantle flow. An example would be to keep track of the thermal history of a piece of crust by updating it every time step with the heat flux from the mantle below. One could also imagine integrating changes in the surface topography by considering the surface divergence of the surface velocity computed in the previous time step: if the surface divergence is positive, the topography is lowered, eventually forming a trench; if the divergence is negative, a mountain belt eventually forms.

In all of these cases, the essential limitation is that postprocessors are *passive*, i.e., that they do not affect the simulation but only observe it.

The statistics file. Postprocessors fall into two categories: ones that produce lots of output every time they run (e.g., the visualization postprocessor), and ones that only produce one, two, or in any case a small and fixed number of often numerical results (e.g., the postprocessors computing velocity, temperature, or heat flux statistics). While the former are on their own in implementing how they want to store their data to disk, there is a mechanism in place that allows the latter class of postprocessors to store their data into a central file that is updated at the end of each time step, after all postprocessors are run.

To this end, the function that executes each of the postprocessors is given a reference to a `dealii::TableHandler` object that allows to store data in named columns, with one row for each time step. This table is then stored in the `statistics` file in the directory designated for output in the input parameter file. It allows for easy

[8] 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.

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.

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 [KHB11]. 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, BK11].

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.

- *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 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.

[BK11] W. Bangerth and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*, 2011. `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.

[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.

[KHB11] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation through modern numerical methods. *submitted*, 2011.

[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.