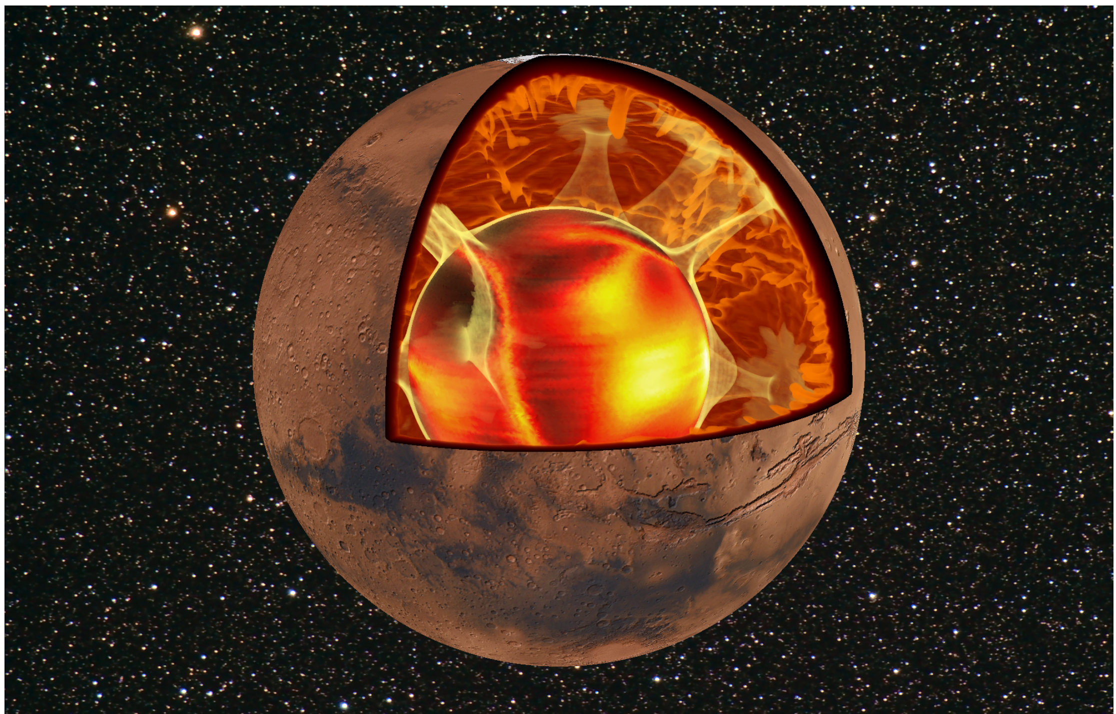
Gaia v2 Manual

Gaia v2 Repository

February 2014

"The very ink with which all history is written is merely fluid prejudice."

Mark Twain



Contents

0 How to use and extend this manual **1**

1 Introduction **2**
1.1 Purpose . . . 2
1.2 Installation and requirements . . . 3
1.3 Coding Convention . . . 4
1.4 Equations . . . 4
1.4.1 Derivation . . . 5

2 Core functionality **8**
2.1 The ini file . . . 8
2.2 Structure of Classes / Templating . . . 9
2.2.1 The Simulation class . . . 9
2.2.2 The Cell class . . . 10
2.2.3 The Grid class . . . 10
2.2.4 The CellWall class . . . 11
2.3 Internal Loops - Outer Iterations, Time-steps . . . 11
2.3.1 The Stokes solver . . . 12
2.3.2 Under-relaxation . . . 12
2.4 Time stepping . . . 13
2.5 Output files . . . 14
2.5.1 Control . . . 14
2.5.2 Field Codes . . . 14
2.5.3 Format . . . 15
2.6 Restarts . . . 15
2.7 Matrix and Solvers . . . 16
2.7.1 Matrix storage . . . 16
2.7.2 Solvers . . . 18
2.7.3 Iterative solvers . . . 18
2.7.4 Solver interfaces to PETSC and CUDA . . . 19
2.8 Modules . . . 19
2.8.1 Creating a Module . . . 20
2.8.2 Integrating modules - The module chains . . . 20
2.8.3 Interaction between modules - Providing functionality . . . 21
2.9 Boundary Conditions . . . 22
2.10 Parallelism . . . 24

2.10.1 Compiling and Running . . . 24
2.10.2 Development . . . 24
2.10.3 Domain Decomposition Cache . . . 26
2.11 All Parameters controlling core functionality . . . 26
2.11.1 Simulation control . . . 27
2.11.2 Precision . . . 28
2.11.3 Other . . . 29

3 Grid **31**
3.1 Generators . . . 31
3.2 File structure . . . 31

4 Modules **33**
4.1 Energy . . . 33
4.1.1 [Boussinesq] (Extended) Boussinesq energy module and body force 33
4.1.2 [GeoFlow] Body force module for GeoFlow experiments . . . 35
4.1.3 [Composition] Eularian transport of a chemical . . . 36
4.2 Rheology . . . 37
4.2.1 [ArrheniusViscosity] Arrhenius viscosity with variable N . . . 37
4.2.2 [FKViscosity] Frank-Kamenetskii viscosity law . . . 37
4.3 Particles . . . 38
4.3.1 [Particles] Particle-in-cell implementation (PIC) . . . 38
4.4 Output . . . 40
4.4.1 [BodyForce] Body force output module . . . 40
4.4.2 [Divergence] Divergence . . . 40
4.5 Benchmark . . . 41
4.5.1 [CavityFlow] Cavity flow . . . 41
4.5.2 [Keken] Rayleigh-Taylor instability benchmark . . . 41
4.6 Initial Condition . . . 42
4.6.1 [InitRefState] Reference density . . . 42
4.6.2 [InitSphHarmonics] Spherical harmonic (3D) or sine (2D) distortion on temperature . . . 43
4.6.3 [InitTempLinear] Linear temperature profile with boundary layers 44
4.7 Other . . . 45
4.7.1 [Box] Box grid loader . . . 45
4.7.2 [InitPrintGaiaS] Outputs ini-file contents . . . 46

5 Post-processing **47**
5.1 IDL . . . 47
5.1.1 NG-suite . . . 47
5.2 Paraview . . . 47
5.3 Gnuplot . . . 47

A An Appendix **48**

Bibliography **50**

0 How to use and extend this manual

- If you refer to a ini-variable, always use the template:
  \param{Module}{Variable} (e.g. \{StokesSphere}{Type}) or
  \param{Variable} (e.g. \{MaxTime}) if the ini-variable does not belong to a module

- Code:
  \begin{code}
  your code here
  \end{code}

- Modules:
  If you write a module manual, include all of the following in the appropriate section in chapter 4:
  \moduleName{Module name (class name)}{Headline}
  \moduleBasicArgs{Field-Code (single character)}{In-Snap (Yes/No)}{Body-Force (Yes/No)}{File name}
  \moduleExtended{Description}{Depends on}{Variables used}{Provides (methods that other modules can use)}{Howto (in what chains...)}{Initial commit (email / date)}

1 Introduction

The work on the Generic Automaton for planetary Interior Analysis started in 2006 in Berlin at the German Aerospace Center (DLR), institute for planetary research. It is a C++ code without dependencies. Please cite as:

Finite volume discretization for dynamic viscosities on Voronoi grids
Physics of the Earth and Planetary Interiors, Volume 171, Issues 14, December 2008, Pages 137146
http://dx.doi.org/10.1016/j.pepi.2008.07.007

1.1 Purpose

The gaia code is a fluid flow solver for arbitrary geometries and its main purpose is to calculate a flow field that fulfills the momentum and mass conservation equations. Primarily this tool is used for Stokes-flow with strongly varying viscosity, a scenario often arising in geophysics, especially to simulate mantle convection. During the last years the equations were extended to support flows driven by inertia, enabling simulations with a Mach number up to 0.2. The module system provides the user with the ability to interact with the core functionality without having to manipulate crucial components of the source code and provide a means to a simulation.

A summary about the core features:

- Navier-Stokes Solver for low Mach-number flow
- Compressible flow with ALA
- Supports Newtonian fluids (Strain   Stress)
- (In)finite Prandtl / Reynolds number (MUSCL adv.)
- Co-located finite-volume discretization
- Arbitrary irregular grid support (must provide Voronoi information)

- Massively parallel, multiple solvers, library independent.
- Fully implicit with a choice of solvers
- Module system provides additional:
  - Energy solvers
  - Different rheologies (Arrhenius / Bingham /)
  - Tracers for chemical convection

1.2 Installation and requirements

The source code complies with the Cxx98 standard and has no immediate dependencies except for a C++ compiler. If you have access to the SVN repository, the following steps might be helpful:

```
mkdir gaia
mkdir gaia/v2
mkdir gaia/grid
svn co https://svn.dlr.de/Gaia/GaiaS_svn/branches/v2/ gaia/v2/
svn co https://svn.dlr.de/Gaia/GaiaS_svn/grid gaia/grid/
```

The default version of make tries to compile the MPI version that requires an MPI-dev package on your system. To compile the single-core version just run

```
make clean;make serial
```

Alternatively you can invoke the compiler directly with

```
<your favorite c++ compiler> -o GaiaS GaiaS.cpp
```

This will produce an executable that you can test with

```
./GaiaS -i GaiaS_blankenbach.ini
```

The MPI version requires an MPI package that can be installed on Ubuntu with:

```
sudo apt-get install mpich2-devel
make clean;make
mpirun -n 4 ./GaiaP -i GaiaS_blankenbach.ini
```

Now the code runs with 4 threads on your local machine. Other make-targets include:
make cuda: creates GaiaC, solver runs on CUDA sm2.0 hardware
make petsc: creates GaiaS, uses the petsc library and enables the use of direct solvers. No parallelism possible!

1.3 Coding Convention

Not all parts of the code obey these standards, nevertheless we should try to enforce them for readability:

- Be descriptive. Comment your code.
- Method names in camelCaseMethod()
- Static method names in UpperCapsMethod()
- Variable names with_under_scores

1.4 Equations

The combination of momentum and mass conservation equation provide two fundamental variables, the velocity (flow field) and pressure. The fluid is assumed to be incompressible, but with possible static variations of density. Furthermore the fluid is supposed to have a linear stress-strain relation. The coupled equations solved by gaia are:

$$\frac{1}{Pr}\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \nabla\cdot\left(\mu(\nabla\mathbf{v} + (\nabla\mathbf{v})^T)\right) + \nabla\left(-\frac{2\mu}{3}\nabla\cdot\mathbf{v}\right) + \mathbf{f}$$

$$\nabla\cdot(\rho\mathbf{v}) = 0$$

The user can provide viscosity, body force and boundary conditions through modules. For numerical details please read:

1.4.1 Derivation

Wikipedia:

==General form of the equations of motion==

The generic body force **b** seen previously is made specific first by breaking it up into two new terms, one to describe forces resulting from stresses and one for "other" forces such as gravity. By examining the forces acting on a small cube in a fluid, it may be shown that:

$\rho \frac{D\mathbf{v}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{f}$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor, and $\mathbf{f}$ accounts for other body forces present. This equation is called the [[Cauchy momentum equation]] and describes the non-relativistic momentum conservation of *any* continuum that conserves mass. $\boldsymbol{\sigma}$ is a rank two symmetric tensor given by its covariant components:

$$\sigma_{ij} = \begin{pmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{pmatrix}$$

where the σ are [[normal stress]]es and τ [[shear stress]]es. This tensor is split up into two terms:

$$:\sigma_{ij} = \begin{pmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{pmatrix} = -\begin{pmatrix} \pi & 0 & 0 \\ 0 & \pi & 0 \\ 0 & 0 & \pi \end{pmatrix} + \begin{pmatrix} \sigma_{xx}+\pi & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy}+\pi & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz}+\pi \end{pmatrix} = -\pi I + \mathbb{T}$$

where I is the 3 x 3 identity matrix and $\mathbb{T}$ is the deviatoric stress tensor. Note that the [[pressure]] "" is equal to minus the mean normal stress: Batchelor 2000—pp=141

$:\pi = -\frac{1}{3}\left(\sigma_{xx} + \sigma_{yy} + \sigma_{zz}\right).$

The motivation for doing this is that pressure is typically a variable of interest, and also this simplifies application to specific fluid families later on since the rightmost tensor $\mathbb{T}$ in the equation above must be zero for a fluid at rest. Note that $\mathbb{T}$ is [[traceless]]. The NavierStokes equation may now be written in the most general form:

$:\rho \frac{D\mathbf{v}}{Dt} = -\nabla\pi + \nabla \cdot \mathbb{T} + \mathbf{f}$

This equation is still incomplete. For completion, one must make hypotheses on the forms of $\mathbb{T}$ and π, that is, one needs a constitutive law for the stress tensor which can be obtained for specific fluid families and on the pressure; additionally, if the flow is

assumed compressible an equation of state will be required, which will likely further require a conservation of energy formulation.

The formulation for Newtonian fluids stems from an observation made by Isaac Newton that, for most fluids,

$\tau \propto \frac{\partial u}{\partial y}$

In order to apply this to the NavierStokes equations, three assumptions were made by Stokes:

* The stress tensor is a linear function of the strain rates.
* The fluid is isotropic.
* For a fluid at rest, $\nabla \cdot \mathbb{T}$ must be zero (so that hydrostatic pressure results).

Applying these assumptions will lead to:

$\mathbb{T}_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \delta_{ij} \frac{2}{3} \frac{\partial u_k}{\partial x_k} \right)$

That is, the deviatoric of the deformation rate tensor is identified to the deviatoric of the stress tensor, up to a factor μ. δ_{ij} is the Kronecker delta. μ and λ are proportionality constants associated with the assumption that stress depends on strain linearly; μ is called the first coefficient of viscosity (usually just called "viscosity") and λ is the second coefficient of viscosity (related to bulk viscosity). The value of λ, which produces a viscous effect associated with volume change, is very difficult to determine, not even its sign is known with absolute certainty. Even in compressible flows, the term involving λ is often negligible; however it can occasionally be important even in nearly incompressible flows and is a matter of controversy. When taken nonzero, the most common approximation is $\lambda \approx -\frac{2}{3}\mu$.

A straightforward substitution of $\mathbb{T}_{ij}$ into the momentum conservation equation will yield the 'NavierStokes equations for a compressible Newtonian fluid':

$\rho \left(\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} \right) = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial x} \left(2\mu \frac{\partial u}{\partial x} - \frac{2\mu}{3} \nabla \cdot \mathbf{v} \right) + \frac{\partial}{\partial y} \left(\mu \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right) + \frac{\partial}{\partial z} \left(\mu \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right) + \rho g_x$

$\rho \left(\frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} + w \frac{\partial v}{\partial z} \right) = -\frac{\partial p}{\partial y} + \frac{\partial}{\partial x} \left(\mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right) + \frac{\partial}{\partial y} \left(2\mu \frac{\partial v}{\partial y} - \frac{2\mu}{3} \nabla \cdot \mathbf{v} \right) + \frac{\partial}{\partial z} \left(\mu \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \right) + \rho g_y$

$\rho \left(\frac{\partial w}{\partial t} + u \frac{\partial w}{\partial x} + v \frac{\partial w}{\partial y} + w \frac{\partial w}{\partial z} \right) = -\frac{\partial p}{\partial z} + \frac{\partial}{\partial x} \left(\mu \left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \right) \right) + \frac{\partial}{\partial y} \left(\mu \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right) \right) + \frac{\partial}{\partial z} \left(2\mu \frac{\partial w}{\partial z} - \frac{2\mu}{3} \nabla \cdot \mathbf{v} \right) + \rho g_z$

or, more compactly in vector form,

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \nabla\cdot\left(\mu(\nabla\mathbf{v} + (\nabla\mathbf{v})^T)\right) + \nabla\left(-\frac{2\mu}{3}\nabla\cdot\mathbf{v}\right) + \rho\mathbf{g}$$

where the matrix transpose has been used. Gravity has been accounted for as "the" body force, ie $\mathbf{f} = \rho\mathbf{g}$. The associated mass continuity equation is:

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{v}) = 0$$

2 Core functionality

This chapter introduces some core concepts of the code and how to use them.

2.1 The ini file

This file provides parameter-value pairs to provide parameters to the simulation. The file itself uses the '=' character to separate parameter from value. Value can be any string, all following or preceding spaces or tabs are truncated. Only one pair per line is allowed. The carriage return character(s) must match the ones where you compiled the simulation. The ';' character denotes comments and is similar to C++'s '//'. An example:

```
;
; -------------- Basic Setup --------------

GridFile    =   ../grid/spiral_16.grid
ThomsonPath =   ../grid/thomson

StokesSphere/Type    =   Box
StokesSphere/Radius    =   0.1

@another.ini
```

To distinct variable names and avoid conflicts, the module name is written in front of a variable. Including other ini files is possible too with the '@' character, followed by the file name. The inclusion will take place where the '@' line is located and all existing variables will be overwritten (top-down parsing).

The default ini file for the simulation is called 'GaiaS.ini'. This can be overwritten with the command line switch '-i <ini-file>'. This (or the default) ini file will be parsed at

the beginning. The parsed values can be accessed via a globally available instance called "ini". This object has the following methods:

```
static Ini ini;

class Ini {
public:
    Ini(const char* file);

    // Check if parameter is defined (bail_out = true: exit if not)
    bool Check(string var_name, bool bail_out = false);

    // Parse & append from a text file, multiple files possible
    void Load(const char* file);

    // converts automatically to int and double!
    void SetValue(const char* var_name, const char* var_value);

    // bail_out = true (default) means exit if not found
    string  GetStringValue(const char* var_name, bool bail_out = true);
    double  GetDoubleValue(const char* var_name, bool bail_out = true);
    int     GetIntValue(const char* var_name, bool bail_out = true);

    // 1 / 0 or "yes"/"no"
    bool    GetBoolValue(const char* var_name, bool bail_out = true);

    void Print();
};

Example usage in a module:

T my_floating_value = ini.GetDoubleValue("MyModule/MyVarName");
```

2.2 Structure of Classes / Templating

The code does make use of the STL (href?), so basic knowledge about the C++ classes vector and valarray are necessary. All classes use a template parameter to define the floating point type. This type is defined in the GaiaS.cpp during instanciation of the Simulation class and is carried out throughout the other classes.

Furthermore, the source code is implicitly written, meaning that the compiler compiles a single .o file (except ini), instead of multiple .o files.

2.2.1 The Simulation class

This is the main class, controlling execution, loops, connects to the grid class and builds / holds matrices for certain operations like divergence / gradient. It also contains the

cell array that describes the geometric and Voronoi properties.

2.2.2 The Cell class

The cell class contains information specific to a discrete volumetric element. It is based on reordered information from the grid's faces array and contains information and ways to manipulate boundary conditions.

A typical loop over all cells and their neighbors is illustrated in the smooth() method in impl_simulation.hpp:

```
for (int i = 0; i < cells.size(); i++) {
    Cell <T> &cell = cells[i]; // shortcut
    if (cell.isBoundaryCell(true))
        continue;
    T sum = 0;
    T value = 0;
    for (int j = 0; j < cell.neighbor.size(); j++) {
        int ne = cell.neighbor[j]; // the neighbor index
        CellWall<T> &f = grid.faces[cell.walls[j]];
        value += f.area * scalar[ne];
        sum += f.area;
    }
    value /= sum; // this is now the average value
    smoothed[i] = central_weight * scalar[i] +
                  (1. - central_weight) * value;
}
```

2.2.3 The Grid class

Contains all information about the grid file and contains the central face and position array.

Important concepts:

- Nodal positions are grouped into layers (or shells), coming from spherical nature to easily access outer/ innermost shells. In general this is not necessary, the whole domain can be just one shell. However, there has to be an inner / outer shell containing only boundary nodes, so a minimum of three shells must exist. See grid.n_inner and grid.n_outer.
- The grid is storing nodal positions, face information and Delaunay triangulation information. (it does not triangulate, the grid generator does that)

2.2.4 The CellWall class

Because these faces are Voronoi faces, they are always perpendicular and always midway to the two parting nodes. The problem is they can be shifted, this is why the center position is not exactly between n1 and n2.

The important properties of the CellWall class:

```
template<typename T>
class CellWall { // also called face
public:
    int neighbor1, neighbor2; // the nodes it divides
    int index;  // in the face array

    vector<int> vvert_indices; // indices of vverts
    Point<T> normal; // normalized vector pointing from n1 to n2
    Point<T> center; // center of face
    InterpolationEntry center_composition;
    InterpolationEntry normal_plus;
    InterpolationEntry normal_minus;
    T area;
    T distance; // between n1 and n2
    T s_distance; // discretization star distance
    bool is_open; // happens on outer faces
};
```

2.3 Internal Loops - Outer Iterations, Time-steps

Running the simulation consists of two nested loops. The outermost loop is the time-stepping loop that advances time and determines the next step width (delta-time). It finishes the simulation if a desired count of steps or time is reached. The limit to this iteration is `MaxSteps` or `MaxTime`, whatever happens first.

The next loop within the time-loop is called the *outer-iteration*. This loop couples non-linear effects like temperature, non-Newtonian viscosities or under-relaxed momentum solutions. It is controlled by the velocity difference between the previous outer iteration. If the difference vanishes or a max iteration count is reached, the outer loop ends and the time-step is done. The relative tolerance is set by `ConvVelRTol`, the according absolute limit is `ConvVelATol`. The iteration limit is controlled by `IterLimitOuter`.

The outer loop calls possible energy modules. These modules have the possibility to alter the temperature / buoyancy field. It is possible to react purely on the temperature residuals instead of velocity. If this is desired, the parameters `ConvTempRTol` and `ConvTempATol` are equivalent in functionality. If either one exists, the velocity residuals are ignored!

The Debug parameter controls the output on the console: 0 = output every time step, 1 = output every outer iteration, 2 = output residual after 100 iterations (with timing), 3 = show all iterations of matrix solvers.

2.3.1 The Stokes solver

The momentum matrix contains the velocity and pressure, making it a huge matrix and a saddle point problem. The precision (relative residual drop) of the momentum matrix-solver can be adjusted by RTolMM. The limit of solver iterations is determined by the parameter MMSolverIts. The type of solver can be altered by MMSolver, with its default type is BICGSL, check Simulation::Init() for available alternatives.

The equation for pressure is multiplied with the viscosity to ensure convergence for iterative solvers. You can multiply all equations for pressure with a constant without altering the theoretical result. However, this does alter the residuum for iterative solvers, shifting weight from the accuracy of momentum towards mass conservation and vice versa. The factor can be adjusted with the Penalty parameter (default 1). Lowering this value can cause fewer iteration at the cost of a worse mass-conservation. The total precision is nevertheless altered by RTolMM, controlling mass and momentum precision.

2.3.2 Under-relaxation

To speed up convergence for steady-state problems or to couple in non-linear effects it is possible to employ under-relaxation for *only* velocity. The method is similar to (ref Ferziger p112) and allows velocity to change only a fraction α, or urf_mm (n is the outer iteration, ϕ can be any velocity component):

$$\phi^n = \phi^{n-1} + \alpha(\phi^{new} - \phi^{n-1})$$

Leading from $Ax = b$ to:

$$\left(\frac{\alpha}{diag(A)}\mathbb{I}\right) Ax = b + \frac{1-\alpha}{\alpha} diag(A) x^{n-1}$$

This method can only be employed with velocity as the main diagonal for pressure is zero. The result of under-relaxation has dramatic influence on the convergence of iterative solvers, but because the result is only a fraction of the true result it needs to be iterated (outer-iterations, control with ConvVelRTol) until convergence is reached.

A few important things about the nature of α:

- For a purely linear problem, it is always faster to use $\alpha = 1$ than $\alpha < 1$ and iterate to achieve the same result.
- A value of $\alpha < 1$ will keep the iterations needed to achieve convergence roughly constant for various resolutions. However, the result of a single solve will be different (smaller).
- To achieve the same result on a different resolution for a specific α, a scaling is required depending on the number of cells.
- Under-relaxation is especially useful if a (quasi-) steady-state solution is sought. To do so, limit the outer iterations to one by setting `IterLimitOuter`=1 and `urf_mm`=0.99. The time accuracy of intermediate solutions is now wrong but the steady state will be correct. It is often useful to start in this mode and then restart with a different α or `urf_mm`.
- The accuracy of mass conservation ($\nabla \cdot u = 0$) is independent of α and only depends on `RTolMM` and `Penalty`.

2.4 Time stepping

Ho to advance a scalar field like temperature in time is up to the modules that handle it, in case of finite-Pr convection, velocity is advected with first-order implicit Euler schema. Nonetheless the simulation has to decide how quickly to advance in time. There are several mechanisms:

- Fixed time-step: `TSType`=FIX, `TSFactor`=<desired TS>
- Courant factor: `TSType`=COURANT, `TSFactor`=<desired multiplicator>
  The Courant criteria defines that no theoretical particle could cross more than the smallest cell in the simulation for the current velocity field. A factor of 0.5 would double that space, therefore allowing a time step twice as big as with a factor of 1.
- Delta factor: `TSType`=DELTA, `TSFactor`=<desired multiplicator>
  The next time step is related to the velocity residual of the last time step and the first result of the outer iteration (first velocity residual output). If a simulation reaches a steady state, the time steps increase dramatically because the velocity does not change anymore.

Further control parameters are:

- MaxDT: Absolute upper limit of the time-step
- InitialDT: Always the initial time step, also for restarts.
- ReduceTimeSteps (boolean): If the outer iteration limit was hit, reduce the next time step.

The simulation does also limit the next time-step increase; the maximal increase is fixed to a factor of two. This is valid for fixed DT as well, the simulation will always start with the initial DT and double it until TSFactor is reached.

2.5 Output files

Output files are binary or ASCII files containing the contents of arbitrary fields. They are written to the current directory or one that is specified with OutputPath and have the format of PX_OUT_<CaseID>_<time step>. All fields are gathered first in case of parallel execution, so there is no difference between single-thread and MPI versions.

2.5.1 Control

The output files are either written every OutputIter iterations or, in case OutputTime is greater than zero, at the iteration that matches that time fraction closest. For debug reasons it is often helpful to generate an output file programatically, which can be done by simply calling produce_output(). The *OuterIterationOutput* module demonstrates this feature.

2.5.2 Field Codes

Field codes are *case sensitive* single character identifiers for scalar, vector or arbitrary fields. Modules can use these codes to register their own scalar field. Which fields will be written in the output file is determined by OutputType. The following codes are reserved by the core:

- T: Temperature
- P: Pressure
- V: Viscosity
- v: Velocity

- S: Strain rate
- D: Viscous dissipation
- I: Ini-file contents

2.5.3 Format

The output format can be either binary or ASCII. Binary has, besides the precision, more advantages as data does not neet to be copied or processed. Especially when running in parallel, it is highly recommended to use binary output. The controlling parameter is `OutputFormat`=BIN/ASCII.

The binary file format follows the endianess of your system. The structure looks like this:

```
6 characters: "GP2OUT"
1 int: length of following string (N)
N characters: full path to grid file, not null terminated
1 double: The current time for this time step
REPEAT
    1 character: field code (FC)
    1 int: value count N
    N doubles: The associated data (always doubles)
UNTIL FC= 'I'
All parameters, as key=value(return) pairs, until end of file
```

For ASCII output, the first line is a headline containing each field code, with the first three being the node coordinates (X Y Z). After that the desired fields from `OutputType` appear on one line per node.

2.6 Restarts

Restarts depend on PX_SNAP_<`CaseID`>_<rank> files. The simulation produces them every `SnapshotIter` iterations. They contain all fields and the current status, almost like in the output file, but also storing internal fields and does not gather, so each thread (rank) gets a separate file. Optionally, the path for those can be changed with `SnapshotPath`.

If you want to restart, simply set `Restart` to yes. Remember that the `CaseID`, grid file and CPU count must match.

Internally, the simulation starts normally, including the execution of all init modules, but afterwards overwrites all fields.

2.7 Matrix and Solvers

2.7.1 Matrix storage

All matrices are stored in the compressed-sparse-row (CSR) format. The public class definition looks like:

```
template<typename T>
class HBMat {

public:

    bool first_build;
    bool store_ap;
    bool rebuild;
    vector<T> values;
    vector<int> Ap_pos;
    vector<int> col_indices;
    vector<int> row_ptr;

    int current_row;

    int m, n, nnz;

    HBMat(bool store_ap = false, int m = 0, int n = 0, int nnz = 0);
    void begin_setup();
    void end_setup();
    void start_new_row();
    void end_row();

    // Elements must come sorted
    void insert_element_fast(int col, T elem);
    unsigned int get_validx(int row, int col);
    void add_element(int col, T value); // mat[current_row, col] += val
    void sanity_check();
    void reset_ap(valarray<T> &scalar); // replaces all Ap's
    void reset_ap(int row, T value);
    void add_ap(int row, T value);
    T get_ap(int row);
    void mul_ap(int row, T value);
    void extract_ap(valarray<T> &scalar);
    T getPerformance();         // return gflops for local matrix
    void mul(const valarray<T> &x, valarray<T> &result);
    HBMat transposed(); // does not work in parallel
    HBMat transposedF(); // Fast version of transposed(), requires twice mem
    void print();
    void print(string fname);
    // can be read with matlab's spconvert(), b and x vectors are written in b.txt and x.txt
    void printMatlab(string fname, valarray<T> &b, valarray<T> &x);
    void normalize(bool keep_sign);
    int getNnz();
    // signals that it is the first build or someone called enableRebuild(). After end_setup = false.
    bool needsRebuild();
    // !! This does not mean new coefficient positions may appear or others get removed, its just a s
    void enableRebuild();
    T getOffDiagonalSum(int row);
};
```

A matrix has to be build row after row, randomly setting (row, col) values is not possible, but randomly setting col in the currect row is possible. After the first build, the matrix remembers its structure and consecutive builds will be faster. That also means that

the *structure cannot be enhanced*. A ”structure” refers to the position of values (row, col), not their actual value. However, during build, the columns within a row can be arbitrary, they don’t have to be sorted. A typical matrix setup looks like:

```
HBMat<T> A(false, 10, 10, 50); // The nnz parameter is a first guess
A.begin_setup();
for (each row)
    A.start_new_row();
    for (some col)
        A.add_element(col, value);
    A.end_row();
end
A.end_setup();
```

2.7.2 Solvers

Solver classes share a common interface so they can be exchanged / extended easily. The basic interface looks like this:

```
class MatSolver {
MatSolver(Simulation<T> *s);
    long double   dot(valarray<T> a, valarray<T> b);
    long double   norm(valarray<T> a);
    // This is the method all real solver classes must provide.
    //  = 0 means must overwrite!
    virtual int solve(HBMat<T> A, valarray<T> x, valarray<T> b,
                int max_it, T a_tol, T r_tol) = 0;
};
```

To use a solver, create an instance and invoke:

```
valarray<T> x(A.n); // x & b should have the same amount of elements
valarray<T> b(A.n); // as the matrix has rows
MatSolverBicgs<T> solver(this);
int iters = solver.solve(A, x, b, 100, 1e-4);
// x now contains solution
```

To develop a new solver, use the provided norm() and dot() methods of the basic solver class as they use long double values (also for global MPI ops).

2.7.3 Iterative solvers

The few available iterative solvers mostly do Jacobi preconditioning. It is very important to not have `FixPressure` set! This would double the complexity for these kind of solvers. The available solvers are:

- MatSolverSteepest: Implementing steepest descent algorithm. Just for reference, very inefficient.
- MatSolverBicgs (`MMSolver` = BICGS): Implements the BiCGStab algorithm.
- MatSolverBicgsl (`MMSolver` = BICGSL): Implements the BiCGStab(l) algorithm. Control of l with **`BICGSL/ell`**. Default l=2.
- MatSolverTfqmr (`MMSolver` = TFQMR): Implements the transpose-free QMR algorithm. Similar efficiency to BICGStab but suffers stalling.
- MatSolverJacobi: Implements the Jacobi algorithm. Requires a factor with `JacobiFactor`. Cannot be used with zeros in main diagonal.

2.7.4 Solver interfaces to PETSC and CUDA

Two more solver classes simulate an interface to the PETSC and CUDA cusp libraries. They can be enabled by selecting them via `MMSolver` = PETSC/CUDA and compiling the distinct executables via make petsc / cuda. These external packages are not aware of the employed domain decomposition so they can be only used in a serial fashion. The executable is usually built with:

```
make petsc
- or -
make cuda
```

For the PETSC version, all command line options for PETSC can be used, enabling the use of MUMPS or similar as direct solvers. For direct solvers it is usually advised to fix a pressure point via `FixPressure` = <global cell index>. The use of PETSC's iterative solvers is not advised as a single iterate may be slow and suffer from precision errors. The same is unfortunately true for the CUDA version.

2.8 Modules

Modules are separate classes within separate files in the module directory. They have access to the Simulation instance and therefore to the Grid instance. Their execution is triggered by Module Chains that are specified in the ini file.

2.8.1 Creating a Module

To create a module, go to the modules folder and run the shell script with the final class- and module name.

```
cd modules
./create_module PlateTectonics
```

Now edit the newly created file and add functionality. It is often helpful to see how other modules interact with the simulation. A module will not be active (i.e. no instance will exist) if it does not appear in at least one module chain.

The module's exec method has a string parameter that is the string that you specify after a module's name in the chain: Box/Init will call the module Box with $param = "Init"$. This way you can distinguish from what chain the module was called.

By default, each module instance has an empty valarray called "field". You can resize / fill this field (valarray<T>) as you desire; if you want this field in you output-file, you just need to give it a field-code (see REF). Now, if this character appears in `OutputType`, it will be in your output file. Another advantage is that you can easily persist this field in a snapshot, meaning it will recover after a restart, by modifying the constructor:

```
1 // replace with unique character if you want "field" to appear in output
2 this->field_id = 'X';
3 this->in_output = false;
4 this->in_snapshot = false;
```

2.8.2 Integrating modules - The module chains

A module chain is a class with a list (or chain) of *instances* of modules. These chains are executed at specific positions in the loops or at certain events. Within the ini file, modules with optional parameters (distinguished with a slash) are comma-separated. An example ini-file line would be:

```
1 MCInit = Box/Init, InitTempLinear, InitSphHarmonics
```

This would execute the three modules in that order after initialization, where the first module is called with the string "Init" as parameter, similar to a call like:

```
1 Box::GetCurrentInstance(sim)->exec("Init");
```

although never use it that way, always check the return of GetCurrentInstance() for NULL in case there is no instance available.

The following chains exist in the simulation:

- `MCInit`: Executed after core initialization, only called once.
- `MCOutput`: Executed before an output file is written.
- `MCPreTS`: Executed first within each time step.
- `MCPostTS`: Executed last within each time step.
- `MCEnergy`: Executed first for each outer iteration. Supposed to modify temperature, but not required to.
- `MCPostOuter`: Executed after each outer iteration.
- `MCRheology`: Executed every time the simulation is required to update the viscosity field. These modules are supposed to modify viscosity.
- `MCPrePressure`: Executed first for each inner iteration.
- `MCBody`: *Special*: This calls a module's bodyForce(index) method instead of exec() to obtain a vector that acts as a body force for the momentum equation. If more than one module is in that chain, the result is summed.

Important: Each module has a single instance. If you call the same module in different chains, it will always be the same instance, meaning you can access data you stored in the class definition from all chains.

2.8.3 Interaction between modules - Providing functionality

Modules like Particles provide a functionality that other modules might want to use. To access instances of available modules, a convenience static method exists within the template (older modules don't have that):

```
StokesSphere<T>* myStokesSphere =
    StokesSphere<T>::GetCurrentInstance(this->s);
```

This can be called from any module and would give you access to the functionality of StokesSphere module. All publicly defined methods are now available, but if the requested module appears in no module chain, the result is NULL, so always check for that. Another example from particles:

```
Particles<T> *part = Particles<T>::GetCurrentInstance(this->s);
if (part == NULL) {
    cerr << "No particles found. Enable first!\n";
    exit(23);
}
```

If a module wants to provide functionality to other modules, it is indicated in the 'Provides' section in chapter 4.

2.9 Boundary Conditions

Two default boundary conditions (BCs) exist that are automatically linked to the first shell (inner / bottom boundary condition) and to the last shell (outer / top boundary condition). Their default behavior can be controlled with the parameters `BCBottomVisc` and `BCTopVisc`. The viscosity at those boundaries can be either zero for free-slip or "inf" for no-slip. Intermediate values are not possible yet.

The BC behavior for energy is controlled with the `BCBottomHFlow` / `BCBottomHValue` resp. `BCTopHFlow` / `BCTopHValue` parameters. "HFlow" refers to heat flow (yes/no) and triggers if the HValue is a heat-flow or a fixed temperature value.

Within the code, BCs are defined separately within the *BoundaryCondition* class. It is linked to the Cell class that provides an easy management functionality. Generally, one can check if a cell is a BC and retrieve the information with:

```
if (cell.isBoundaryCell()) {
    BoundaryCondition<T> *bc = cell.getBC();
    if (bc->viscosity == 0) // No-slip
        ...
}
```

The BC class holds the following information:

```
class BoundaryCondition {
public:
    // by default, create insulating free-slip
    BoundaryCondition();
    // this is for internal use and reflects on how many cells this
    // BC is in use.
    int count;

    // _only_ Thermal BC, for MM it's a compute node
    bool thermalBoundary;

    // value is either fixed temperature or heat flow
    bool heatFlow;  // false means value is fixed-T!
    T   value;


    // Momentum related:

    // For in-, out- or shear-flow, this value is non-zero
    Point<T> velocity;

    // If zero then no-slip, Inf means free-slip.
    // Intermediates don't work yet, as it would req. a gradient
    T   viscosity;

    // Convenience instanciation:
    static BoundaryCondition<T>* InsulatingFreeSlip();
    static BoundaryCondition<T>* FixedTempFreeSlip(T temp);
    static BoundaryCondition<T>* InsulatingNoSlip();
    static BoundaryCondition<T>* FixedTempNoSlip(T temp);
};
```

Setting and removing boundary conditions is handled by the Cell class, please look at the Box module for a demonstration:

```
BoundaryCondition<T>* sidewallBC = BoundaryCondition<T>::InsulatingFreeSlip();
for (int i = g->n_inner; i < g->n_outer; i++) {
    if (s->cells[i].getVolume() > 0) // Box grids have negative volume on open cells
        continue;
    s->cells[i].setBoundaryCondition(sidewallBC, true); // no halo check true here
}
// Important for parallel runs, always execute after changing BC setup!
s->exchange_bc();
```

2.10 Parallelism

2.10.1 Compiling and Running

The complete code is makes use of the MPI library to communicate between different CPUs and thus run in parallel. The default "make" tries to build the MPI version, ending with a P, that can be run as follows:

```
mpirun -np 4 ./GaiaP -i GaiaS_blankenbach.ini
```

The according development packages for your favorite MPI library need to be present to build the MPI version. The command *mpicxx* or similar should be present.

2.10.2 Development

To determine the rank and process count, there are two global integers called *rank* and *cpu_count* to determine the own process number (*rank*) amongst all started by MPI *cpu_count*.

The file impl_mpi.hpp contains routines to distribute the grid among different CPUs, called domain-decomposition. Each process (rank) holds only a portion of the grid. This portion of the grid is typically surrounded by halo-cells or boundary cells.

Halo cells are cells whose values reside on a different CPU but can be obtained upon request. At the beginning the grid is divided and distributed. The *grid.gross_points* and all other grid specific variables reflect only the part of the grid that is local to the rank, including halo cells! A sum of *grid.gross_points* among all processes would therefore always lead a greater number than the true amount of all cells. To distinguish halo cells, the cell class has a boolean property called *is_halo_cell*. A more viable variant is to check for boundary or halo cell together by using *cell.isBoundaryCell(true)*. The boolean parameter defines if the result should be "or"ed with the is_halo_cell property or not.

The code provides some convenience functions for MPI handling, so you should avoid using MPI calls directly (this way your code will work in the serial version without MPI libraries). These set of functions include filling halo cells and computing sums and norms.

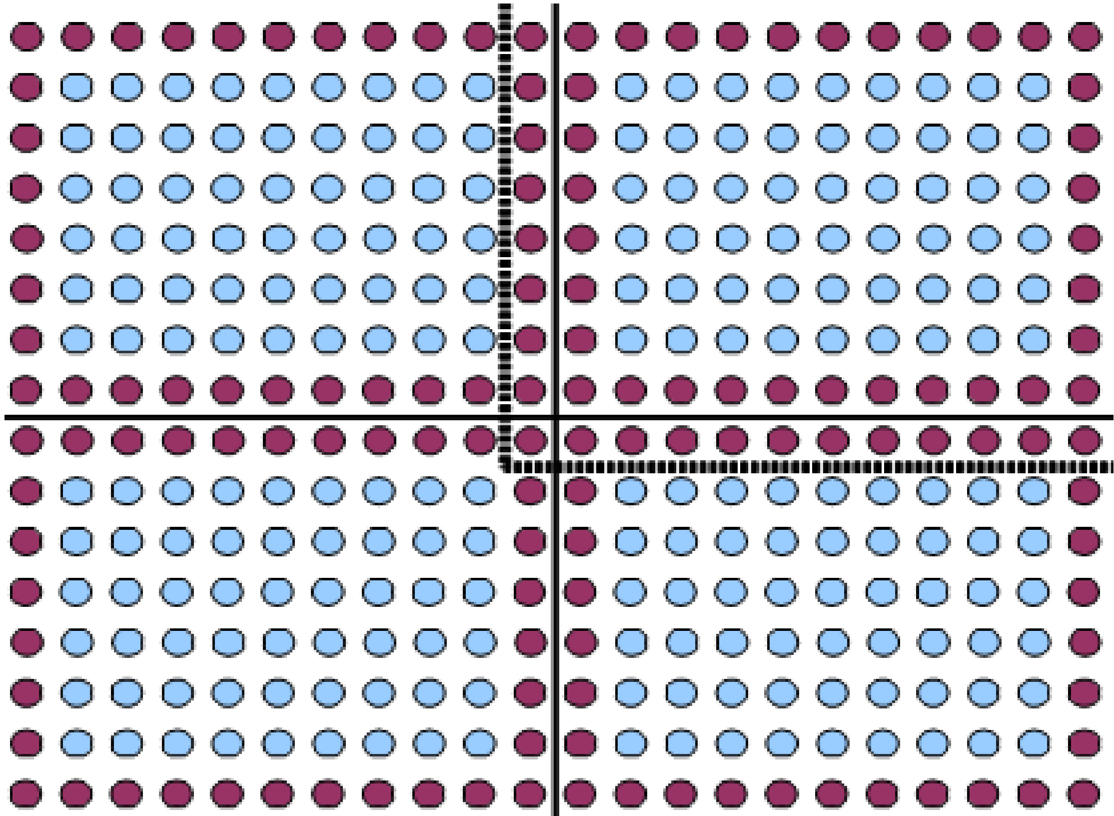


FIGURE 2.1: Domain Decomposition

Hint: You usually only need the expensive exchange() call if you compute some kind of partial derivative like grad / div, but these routines do it already for you. The more you use these calls, the more inefficient it will be.

- exchange(valarray<T> in, valarray<T> out)

  The "in" array can be the same as "out". After calling, the out array will contain proper values for the halo cells, other values are not affected by this call. The array size must be a multiple of gross_points, enabling the exchange of more than one value with a single call. To use this, align your data within the array field-by-field (planar), i.e. [AAA...,BBB....] and not interleaved [ABABAB...].

- exchange(vector<Point<T> > vec)

  Same functionality, but with an array of positions / velocities.

- global_sum(T local_sum)

- global_norm(valarray<T> scalar[, T norm])
  Takes care of not accounting halo cells when computing.

- global_min(T local_min)

- global_max(T local_max)

Make sure these methods are called on all CPUs, don't use them within an if statement that depends on a local value, or your code will block. A common mistake is to use it with cout:

```
1   cout << "Mean temperature: " << global_norm(temperature, 1) << endl;
```

The code above would block as cout is filtered to be executed only on rank 0. The proper way to do it:

```
1   T mean_temp = global_norm(temperature, 1);
2   cout << "Mean temperature: " << mean_temp << endl;
```

2.10.3 Domain Decomposition Cache

To avoid the time and memory consuming decomposition and distribution at each start, the ini variable `DDCache` can be used to create cache files for a certain grid and a certain number of processes.

To create cache files, set `DDCache` to the desired amount of processes together with the desired `GridFile` and run the MPI version explicitly with a single process (-np 1). Now the cache will be generated and the program will exit. The files will be in the same folder as the grid file, named DDCACHE_*GridFile_cpu_count_rank*. Remember that for big grids a lot of memory is required for the decomposition because the whole grid must be loaded at once at the beginning.

To use the cache, set `DDCache` to the desired amount of processes or simply set it to "use". Now start the simulation as usual with the proper amount of processes.

2.11 All Parameters controlling core functionality

Some parameters (like Advection or InitialModeL) now belong to modules but don't have their module prefix. This will hopefully change sometime. Others like `DRef` or `Ra` are in here for generic use, many modules require them although the core does not. If there is no default for a parameter ("N/A"), you have to specify it within the ini file.

2.11.1 Simulation control

`Debug`= N/A	Level of verbosity, 0 = min, 5 = max
`CaseID`= N/A	A short string to identify output and snapshot files.
`GridFile`= N/A	Grid file
`OutputPath`= ./	Pathname for output files
`OutputFormat`= BIN	Output format (ASCII/BIN)
`OutputTime`= 0	Time interval for output, simulation time
`OutputIter`= N/A	Time step interval for output, simulation time
`OutputType`= N/A	A string with field codes, see output section
`UseSnap`= N/A	Like Restart, but time starts at zero.
`MaxTime`= 1.2	Maximum scaled time, 0 = endless
`InitialDT`= N/A	Initial time step, regardless of restarts
`MaxDT`= N/A	Maximal time step
`TSType`= N/A	DELTA / FIX / COURANT, see time section
`TSFactor`= N/A	See time section
`SnapshotPath`= ./	Pathname for snapshot files
`SnapTime`= 0	Time interval for snapshot output, independent of SnapshotIter, unit: simulation time
`SnapRunTime`= 0	Time interval for Snapshot Output, independent of SnapshotIter, unit: run time (seconds) (???)
`SnapshotIter`= N/A	Time step interval for Snapshot Output

`Restart`= N/A	Try to load restart files? (yes/no)
`RestartFromSnap`= 0	???
`RadialSplit`= 0	Domain decomposition for spherical grids: how many divisions radially?
`DDCache`= 0	Domain decomposition cache value, set to desired CPU count but let it run with just one to build the cache, if CPU count matches this number it tries to use it
`MaxSteps`= 0	Amount of time steps after simulation ends, 0=endless
`seed`= 17031979	The initial seed for random number generator
`Dref`= 1	Reference depth
`Tref`= 1	Reference temperature

2.11.2 Precision

`MaxVelocity`= 1e90	End simulation if rms-velocity is above that value (breakdown-indicator)
`ReduceTimeSteps`= 0	1: reduce timestep if IterLimitOuter reached
`IterLimitOuter`= 80	Maximum amount of outer iterations
`ConvTempATol`= 1e-4	Absolute tolerance for temperature residual
`ConvTempRTol`= 1e-2	Relative tolerance for temperature residual. Careful: if ConvVelRTol > 0, these limits are ignored.
`ConvVelATol`= 1e-2	Absolute tolerance for velocity residual
`ConvVelRTol`= 0	Relative tolerance for velocity residual. $0 =$ ignore (use ConvTempR/ATol)

`ConvDivRes`= 1e-2	Terminate inner iterations if mean absolute divergence does not lower by more than this value
`Advection`= 2	Advection scheme: 0 = Upwind, 1 = CDS, 2 = MUSCL, 3 = Particles (must be setup)
`urf_mm`= 1	Under-relaxation factor for momentum equation (1>x>0)
`RTolMM`= 1e-4	Matrix solver relative tolerance for momentum equation
`MMSolver`= BICGS	Name of solver to use. Choose between SD (steepest descent), BICGS, TFQMR, CUDA, PETSC. Might require special compile options.
`IniStrainRate`= 1e-19	Initial strain rate
`ChasteTS`= no	Don't let the stokes solver "remember" anything from previous TS, pressure and velocity is set to 0. Don't do this for finite Prandtl number.
`NonNewtonianRheology`= no	Strain-rate update during outer-iterations?
`MaxViscContrast`= 1e30	Limit viscosity contrast. Always from lowest viscosity value.

2.11.3 Other

`Compressibility`= 0	0: Ignore density variations. = 1 includes compressibility for static reference density, = 2 dynamic
`PrInverted`= 0	For absent thermal convection, this is the Reynolds number, otherwise $1/Prandtl$
`Ta`= 0	Taylor number
`Ra`= 1	The bottom-heated Rayleigh number (see module Boussinesq)
`RaQ`= 0	The internally-heated Rayleigh number (see module Boussinesq)

`Di`= 0.0	Dissipation number (see module Boussinesq)
`InitialTemp`= 1.0	Initial value for temperature field
`InitialAmp`= 0.05	Amplitude for perturbation (see module InitSphHarmonics)
`AmplRScaling`= yes	??
`InitialModeL`= 0	Spherical harmonics L (see module InitSphHarmonics)
`InitialModeM`= 0	Spherical harmonics M (see module InitSphHarmonics)
`InitialModeL2`= 0	Spherical harmonics L/M (second disturbance) (see Module InitSphHarmonics)
`InitialModeL3`= 0	Spherical harmonics L/M (third disturbance) (see Module InitSphHarmonics)

3 Grid

The grid information is stored in a file, usually generated by a grid generator and read by the Grid class within the simulation. The information consists of positions of the cells, volumes and Voronoi information such as face areas and neighboring nodes. The grid has to stored with proper Voronoi information. Some basic grids are available in the repository under https://svn.dlr.de/Gaia/GaiaS_svn/grid/.

3.1 Generators

The 3D spherical shell grid-generator is writrten in C and available at https://svn.dlr.de/Gaia/GaiaS_svn/trunk/c_source/gridgen/. It requires a current qhull library installed and patched. It can create various grids by taking input points on a unity shell and project them or by using the internal spiral engine to create points.

For 2D spherical or box grids, only IDL generators exist.

3.2 File structure

There are two kinds of grid files the simulation can read: ASCII based (ending with .grid_ascii) and binary (.grid). The endianess of the binary version depends on the system it was created on, integers have 32bits and doubles 64. The file has the following structure:

- 3 characters identifying the grid type: currently supported "BOX" or "SPH"
- 1 character identifying dimensions: "3" for 3D, "2" for 2D
- 5 characters fixed: "GRID_"
- 1 character identifying binary grids: "B" for binary, "A" for ASCII.
- 2 characters as version, currently "04"

- 1 double inner radius
- 1 double outer radius
- 1 double resolution (should be roughly the mean distance between cells)
- 1 int amount of shells (radial discretization or y resolution for box)
- For each shell (n_total = 0):
  - 1 int (=n) amount of locations for that shell; n_total += n
  - n * 3 doubles expressing x, y and z locations of cell. Z is always present, even for 2D cases!
- 1 int (=n) amount of Voronoi vertices (vverts)
- n * 3 doubles location of vertices
- 1 int (=n) amount of surface areas for each cell (n should be equal to n_total)
- n_total doubles surfaces (sum of face areas for each closed cell)
- 1 int (=n) amount of volumes for each cell (n should be equal to n_total)
- n_total doubles volumes
- 1 int (=n) amount of tetrahedra (from Delaunay triangulation)
- n * 4 int indices for locations connecting a single tetrahedron
- 1 int (=n) amount of faces
- For each face (polygon dividing two cells):
  - 1 int neighbor index 1 (from locations array)
  - 1 int neighbor index 2 (from locations array)
  - 1 double face area
  - 1 int amount of vverts connecting the polygon (=n_v)
  - n_v int indices to vvert

4 Modules

This chapter describes the functionality of available modules. Look at section 2.4 on how modules work.

4.1 Energy

4.1.1 [Boussinesq] (Extended) Boussinesq energy module and body force

Field-Code	In-Snap	Body-Force	File Name
W	No	Yes	Boussinesq.hpp

Description:	This module modifies temperature according to the energy equation and provides a radial acting body force. The energy equation is capable of handling standard Boussinesq, extended Boussinesq and (T)ALA. TODO: Write up how total temperature is defined. The equation for standard B. is:

$$\frac{\partial T}{\partial t} + \vec{u} \cdot \nabla T = \nabla^2 T + \frac{Ra_Q}{Ra} \tag{4.1}$$

Only the CDS advection guarantees energy balance, but creates wiggles if the local Peclet number becomes too high.
To change various compressibility approximations use (`MCBody` never changes):

- Std. Boussinesq:
  Compressibility = 0 ;(default)
  MCEnergy = Boussinesq
- Ext. Boussinesq:
  Compressibility = 0 ;(default)
  MCEnergy = Boussinesq/Compress
- TALA:
  Compressibility = 1
  MCEnergy = Boussinesq/Compress
  MCInit = InitRefState
- ALA:
  Compressibility = 1
  MCEnergy = Boussinesq/Compress
  MCInit = InitRefState
  Boussinesq/ALA = yes

Depends on: [Particles, InitRefState]

Variables used (incl. defaults):	Advection = 0 (upwind) Advection = 1 (CDS - default) Advection = 2 (MUSCL) Advection = 3 (particles - set up particle chains!) **Boussinesq/ALA** = no Di = 0 (Dissipation number) Ra = reqd. (Rayleigh number) RaQ = 0 (Internally heated Rayleigh number) T0 = 0 (Non-dimensional surface temperature)
Provides:	Body force ::update_phi(valarray); Calculates the phi field for MUSCL advection 'W' field for output providing work-done
Howto:	MCEnergy = Boussinesq[/Compress] MCBody = Boussinesq
Initial:	christian.huettig@nianet.org / Jan 31 2012

4.1.2 [GeoFlow] Body force module for GeoFlow experiments

Field-Code	In-Snap	Body-Force	File Name
-	No	Yes	GeoFlow.hpp

Description:	This module provides a radial acting body force purely based on the temperature, radius and the Rayleigh number to simulate GeoFlow ISS-conditions. Additionally an unidirectional body force can be applied to simulate Earth-lab-conditions. http://www.eusoc.upm.es/en/e-usoc/spacemission/geoflow.html
Depends on:	-
Variables used (incl. defaults):	**GeoFlow/Ra_z** = 0.0 //Rayleigh number for the unidirectional buoyancy force (for experiments on Earth) **GeoFlow/radius_power** = -5.0 //power of the radius multiplying the buoyancy force (for ISS experiments $Rar^{-5}Te_r$)

Provides:	Body force
Howto:	MCInit = GeoFlow/Init MCBody = GeoFlow MCEnergy = Boussinesq use no-slip boundary conditions to simulate GeoFlow parameters
Initial:	ina05ro@yahoo.com / Mar 19 2013

4.1.3 [Composition] Eularian transport of a chemical

Field-Code	In-Snap	Body-Force	File Name
C	yes	yes	Composition.hpp

Description:	Solves advection of a distinct chemical field that counteracts buoyancy (different density) with a compositional Rayleigh number. TODO: write formula
Depends on:	-
Variables used (incl. defaults):	`Composition/RaC` = reqd. `Composition/LewisNumber` = not used, implicit advection provides diffusion anyway
Provides:	compositional field in field property.
Howto:	MCInit=Composition/Init MCEnergy=Boussinesq, Composition/Solve MCBody = Boussinesq,...,Composition
Initial:	christian.huettig@dlr.de / 28.6.11

4.2 Rheology

4.2.1 [ArrheniusViscosity] Arrhenius viscosity with variable N

Field-Code	In-Snap	Body-Force	File Name
-	-	-	RH_ArrheniusViscosity.hpp

Description: Updates the viscosity field of the simulation according to the Arrhenius-law:

$$\eta = \epsilon_{ref}^{\frac{N-1}{N}} \epsilon^{\frac{1-N}{N}} \exp\left(\frac{E+Vd}{N(T+T_0)} - \frac{E+VD_{ref}}{N(T_{ref}+T_0)}\right) \tag{4.2}$$

All variables are non-dimensionalized. E activation energy; V activation volume; d depth; N creep-exponent; D_{ref} reference depth; T_{ref} reference temperature; T temperature; T_0 surface temperature; ϵ_{ref} reference strain-rate; ϵ strain-rate;

Depends on: -

Variables used (incl. defaults):
ArrheniusViscosity/E = reqd. ; Activation energy
ArrheniusViscosity/V = reqd. ; Activation volume
ArrheniusViscosity/N = reqd. ; N=0 diffusion creep; N=3 dislocation creep.
Also needed: T0, Tref, Dref, EpsRef

Provides: -

Howto: MCRheology = ArrheniusViscosity

Initial: ?

4.2.2 [FKViscosity] Frank-Kamenetskii viscosity law

Field-Code	In-Snap	Body-Force	File Name
-	-	-	RH_FKViscosity.hpp

Description:	Updates the viscosity field of the simulation according to the FK-law: $$\eta = \exp\left(\log \Delta\eta_T \left(T_{ref} - T\right) + \log \Delta\eta_P \left(d - D_{ref}\right)\right) \quad (4.3)$$ All variables are non-dimensionalized. d depth; D_{ref} reference depth; T_{ref} reference temperature; T temperature;$\Delta\eta_T$ viscosity contrast due to temperature;$\Delta\eta_P$ viscosity contrast due to pressure
Depends on:	-
Variables used (incl. defaults):	**`FKViscosity/ViscT`**= reqd., equivalent to $\Delta\eta_T$ **`FKViscosity/ViscP`**= reqd., equivalent to $\Delta\eta_P$ Also needed: `Tref`, `Dref`
Provides:	-
Howto:	`MCRheology`= FKViscosity
Initial:	?

4.3 Particles

4.3.1 [Particles] Particle-in-cell implementation (PIC)

Field-Code	In-Snap	Body-Force	File Name
p	yes	no	ParticleHelper.hpp

Description:	This module attempts to provide an infrastructure to track different materials, including feedback to the momentum solver. The particles are mass-less, meaning they are not necessarily conserving mass if treated as such. Initialization is either random to match desired cell density or regular gridded in case of BOX grids. The advection is done using a fourth order Runge-Kutta scheme, see Fig. 4.1.

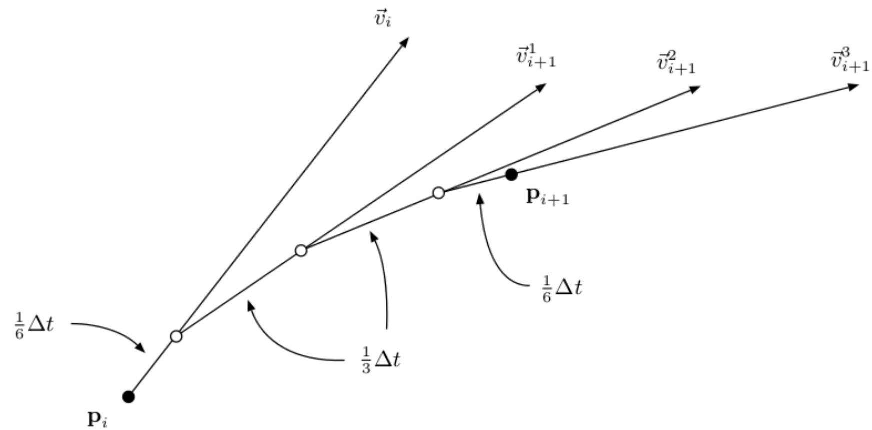


FIGURE 4.1: Runge-Kutta 4th order advection

Depends on:	SupportGrid module
Variables used (incl. defaults):	**Particles/Density** = 20; desired particles per cell **Particles/InvDistPower** = 1;
Provides:	The following commands are available for other modules or per call in a module chain: • Init
Howto:	Howto
Initial:	submitted-by-when

4.4 Output

4.4.1 [BodyForce] Body force output module

Field-Code	In-Snap	Body-Force	File Name
B	No	No	BodyForce.hpp

Description:	This module outputs the body force vector or if specified with /S, a scalar containing the length of the body force vector for each cell.
Depends on:	-
Variables used (incl. defaults):	-
Provides:	-
Howto:	`MCOutput`= BodyForce[/S] `OutputType` = B
Initial:	christian.huettig@nianet.org / 14.03.2012

4.4.2 [Divergence] Divergence

Field-Code	In-Snap	Body-Force	File Name
U	no	no	Divergence.hpp

Description:	Calculates the divergence of the current velocity field.
Depends on:	-
Variables used (incl. defaults):	-
Provides:	-

Howto:	MCOutput= Divergence OutputType = U
Initial:	christian.huettig@nianet.org / 11/2011

4.5 Benchmark

4.5.1 [CavityFlow] Cavity flow

Field-Code	In-Snap	Body-Force	File Name
-	No	No	CavityFlow.hpp

Description:	Reproduces the cavity flow benchmark in a box. Sets up boundary conditions for a 1x1 box according to benchmark: NoSlip everywhere, velocity on top u = 1, v= 0. Adjust PrInverted as Reynolds number.
Depends on:	-
Variables used (incl. defaults):	Box
Provides:	-
Howto:	See GaiaS_cavity.ini for details.
Initial:	christian.huettig@dlr.de / 7th March 2013

4.5.2 [Keken] Rayleigh-Taylor instability benchmark

Field-Code	In-Snap	Body-Force	File Name
C	yes	yes	Keken.hpp

Description:	Reproduces Keken et. al. benchmark cases. Title: A comparison of methods for the modeling of thermochemical convection. JGR 1997 ATTENTION: special box grids needed with aspect ratio (width) of 0.9142.
Depends on:	Particles
Variables used (incl. defaults):	**`Keken/RaC`** = reqd.; Body force multiplier, just for time scaling. **`Keken/Viscosity`** = reqd.; The viscosity of the chemical. Everything else has $\eta = 1$.
Provides:	-
Howto:	See GaiaS_keken.ini !
Initial:	christian.huettig@dlr.de Sept 2013

4.6 Initial Condition

4.6.1 [InitRefState] Reference density

Field-Code	In-Snap	Body-Force	File Name
-	-	-	InitRefState.hpp

Description:	Initializes the refdensity and reftemp array in Simulation. Supports Adams-Williamson profile: $$\rho_{ref} = \exp\left(Di\, d\right) \qquad T_{ref} = T_0 \exp\left(Di\, d\right) \tag{4.4}$$
Depends on:	-
Variables used (incl. defaults):	`Di` = 0; Dissipation number `T0` = 0; Surface temperature

Provides:	-
Howto:	MCInit = InitRefState/AdamsWilliamson
Initial:	christian.huettig@dlr.de / 15.3.13

4.6.2 [InitSphHarmonics] Spherical harmonic (3D) or sine (2D) distortion on temperature

Field-Code	In-Snap	Body-Force	File Name
-	-	-	InitSphHarmonics.hpp

Description:	Adds [multiple] spherical harmonic (3D) or sine (2D) fields to current temperature field. The mode and amplitude is determined by variables and reaches its peak at mid-depth, damped by a half sine across depth.
Depends on:	-
Variables used (incl. defaults):	`InitialModeL`[2,3,4,...] = 0; L or frequency for sine in 2D `InitialModeM`[2,3,4,...] = 0; M, no use in 2D `InitialAmp`[2,3,4,...] = 0; Amplitude

Provides:

```
void sph_harmonic_distortion(valarray<T> &scalar, T amplitude, T l, T
m, T power = 1.);
static double sph_harm(double theta, double phi, int l, int m);
static inline int factorial(int n);
static double legendre(int l, int m, double x);
```

Howto:	`MCInit`=InitTempLinear, InitSphHarmonics To trigger cubical mode (6 plumes in 3D): `InitialModeL` = 4 `InitialModeM` = 0 `InitialAmp` = 0.15 `InitialModeL2` = 4 `InitialModeM2` = 4 `InitialAmp2` = 0.15 To trigger tetrahedral mode (4 plumes in 3D): `InitialModeL` = 3 `InitialModeM` = 2 `InitialAmp` = 0.15
Initial:	from v1

4.6.3 [InitTempLinear] Linear temperature profile with boundary layers

Field-Code	In-Snap	Body-Force	File Name
-	-	-	InitTempLinear.hpp

Description:	Sets up initial temperature condition to a linear profile depending on depth. Optional provide linear boundary layers. Grid differentiation for spherical shell and full sphere is respected.
Depends on:	-
Variables used (incl. defaults):	`InitialTemp` = 1; Maximal (bottom) temperature **`ITL/TopLayerDepth`** = 0; depth of top layer **`ITL/TopLayerMax`** = 0; temperature at TopLayerDepth **`ITL/BottomLayerDepth`** = 1; depth of bottom layer **`ITL/BottomLayerMin`** = 1; temperature at BottomLayerDepth The temperature between TopLayerDepth and BottomLayerDepth is linearly interpolated.

Provides:	-
Howto:	MCInit=InitTempLinear
Initial:	from v1

4.7 Other

4.7.1 [Box] Box grid loader

Field-Code	In-Snap	Body-Force	File Name
-	No	No	Box.hpp

Description:	Provides the ability to load 2D box grids. Also implements domain decomposition and replaces the sim-¿grid instance with a RegionalGrid class. The radius array becomes one, the shell_radius gets the y value of the first cell in a row. Boundary conditions are properly implemented.
Depends on:	-
Variables used (incl. defaults):	-
Provides:	-

Howto:	MCInit = Box/Init
	Some box2_* grid file.
	
	To access box properties:
	//within another module:
	if (Box::GetCurrentInstance(this->s) != NULL) {
	RegionalGrid<T> *g = static_cast<RegionalGrid<T> *>(s->grid);
	cout << g->length_x << " " << g->length_y << endl; }

Initial:	ported by christian.huettig@nianet.org / 2/2/12, originally implemented by Lena Noack lena.noack@dlr.de * Update to v1-functionality: lena.noack@dlr.de / 08.02.12

4.7.2 [InitPrintGaiaS] Outputs ini-file contents

Field-Code	In-Snap	Body-Force	File Name
-	-	-	InitPrintGaiaS.hpp

Description:	Outputs ini-file contents as it was parsed by all four parsers (bool, int, double, string). Mainly for debug.
Depends on:	-
Variables used (incl. defaults):	-
Provides:	-
Howto:	-
Initial:	?

5 Post-processing

5.1 IDL

5.1.1 NG-suite

5.2 Paraview

5.3 Gnuplot

A | An Appendix

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus at pulvinar nisi. Phasellus hendrerit, diam placerat interdum iaculis, mauris justo cursus risus, in viverra purus eros at ligula. Ut metus justo, consequat a tristique posuere, laoreet nec nibh. Etiam et scelerisque mauris. Phasellus vel massa magna. Ut non neque id tortor pharetra bibendum vitae sit amet nisi. Duis nec quam quam, sed euismod justo. Pellentesque eu tellus vitae ante tempus malesuada. Nunc accumsan, quam in congue consequat, lectus lectus dapibus erat, id aliquet urna neque at massa. Nulla facilisi. Morbi ullamcorper eleifend posuere. Donec libero leo, faucibus nec bibendum at, mattis et urna. Proin consectetur, nunc ut imperdiet lobortis, magna neque tincidunt lectus, id iaculis nisi justo id nibh. Pellentesque vel sem in erat vulputate faucibus molestie ut lorem.

Quisque tristique urna in lorem laoreet at laoreet quam congue. Donec dolor turpis, blandit non imperdiet aliquet, blandit et felis. In lorem nisi, pretium sit amet vestibulum sed, tempus et sem. Proin non ante turpis. Nulla imperdiet fringilla convallis. Vivamus vel bibendum nisl. Pellentesque justo lectus, molestie vel luctus sed, lobortis in libero. Nulla facilisi. Aliquam erat volutpat. Suspendisse vitae nunc nunc. Sed aliquet est suscipit sapien rhoncus non adipiscing nibh consequat. Aliquam metus urna, faucibus eu vulputate non, luctus eu justo.

Donec urna leo, vulputate vitae porta eu, vehicula blandit libero. Phasellus eget massa et leo condimentum mollis. Nullam molestie, justo at pellentesque vulputate, sapien velit ornare diam, nec gravida lacus augue non diam. Integer mattis lacus id libero ultrices sit amet mollis neque molestie. Integer ut leo eget mi volutpat congue. Vivamus sodales, turpis id venenatis placerat, tellus purus adipiscing magna, eu aliquam nibh dolor id nibh. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sed cursus convallis quam nec vehicula. Sed vulputate neque eget odio fringilla ac sodales urna feugiat.

Phasellus nisi quam, volutpat non ullamcorper eget, congue fringilla leo. Cras et erat et nibh placerat commodo id ornare est. Nulla facilisi. Aenean pulvinar scelerisque eros

eget interdum. Nunc pulvinar magna ut felis varius in hendrerit dolor accumsan. Nunc pellentesque magna quis magna bibendum non laoreet erat tincidunt. Nulla facilisi.

Duis eget massa sem, gravida interdum ipsum. Nulla nunc nisl, hendrerit sit amet commodo vel, varius id tellus. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc ac dolor est. Suspendisse ultrices tincidunt metus eget accumsan. Nullam facilisis, justo vitae convallis sollicitudin, eros augue malesuada metus, nec sagittis diam nibh ut sapien. Duis blandit lectus vitae lorem aliquam nec euismod nisi volutpat. Vestibulum ornare dictum tortor, at faucibus justo tempor non. Nulla facilisi. Cras non massa nunc, eget euismod purus. Nunc metus ipsum, euismod a consectetur vel, hendrerit nec nunc.

Bibliography

Index

ArrheniusViscosity
- ArrheniusViscosity/E, 37
- ArrheniusViscosity/N, 37
- ArrheniusViscosity/V, 37

Base
- Advection, 29, 35
- AmplRScaling, 30
- BCBottomHFlow, 22
- BCBottomHValue, 22
- BCBottomVisc, 22
- BCTopHFlow, 22
- BCTopHValue, 22
- BCTopVisc, 22
- CaseID, 14–16, 27
- ChasteTS, 29
- Compressibility, 29
- ConvDivRes, 29
- ConvTempATol, 11, 28
- ConvTempRTol, 11, 28
- ConvVelATol, 11, 28
- ConvVelRTol, 11, 12, 28
- DDCache, 26, 28
- Debug, 12, 27
- Di, 30, 35, 42
- DRef, 26
- Dref, 28, 37, 38
- EpsRef, 37
- FixPressure, 18, 19
- GridFile, 26, 27
- IniStrainRate, 29
- InitialAmp, 30, 43, 44
- InitialAmp2, 44
- InitialDT, 14, 27
- InitialModeL, 30, 43, 44
- InitialModeL2, 30, 44
- InitialModeL3, 30
- InitialModeM, 30, 43, 44
- InitialModeM2, 44
- InitialTemp, 30, 44
- IterLimitOuter, 11, 13, 28
- JacobiFactor, 19
- MaxDT, 14, 27
- MaxSteps, 11, 28
- MaxTime, 11, 27
- MaxVelocity, 28
- MaxViscContrast, 29
- MCBody, 21, 34
- MCEnergy, 21
- MCInit, 21, 44, 45
- MCOutput, 21, 40, 41
- MCPostOuter, 21
- MCPostTS, 21
- MCPrePressure, 21
- MCPreTS, 21
- MCRheology, 21, 37, 38
- MMSolver, 12, 19, 29
- MMSolverIts, 12
- NonNewtonianRheology, 29
- OutputFormat, 15, 27
- OutputIter, 14, 27
- OutputPath, 14, 27
- OutputTime, 14, 27
- OutputType, 14, 15, 20, 27, 40, 41

Penalty, 12, 13
PrInverted, 29, 41
Ra, 26, 29, 35
RadialSplit, 28
RaQ, 29, 35
ReduceTimeSteps, 14, 28
Restart, 16, 28
RestartFromSnap, 28
RTolMM, 12, 13, 29
seed, 28
SnapRunTime, 27
SnapshotIter, 15, 27
SnapshotPath, 15, 27
SnapTime, 27
T0, 35, 37, 42
Ta, 29
Tref, 28, 37, 38
TSFactor, 13, 27
TSType, 13, 27
urf_mm, 12, 13, 29
UseSnap, 27

BICGSL
BICGSL/ell, 19

Boussinesq
Boussinesq/ALA, 35

Composition
Composition/LewisNumber, 36
Composition/RaC, 36

FKViscosity
FKViscosity/ViscP, 38
FKViscosity/ViscT, 38

GeoFlow
GeoFlow/Ra_z, 35
GeoFlow/radius_power, 35

ITL
ITL/BottomLayerDepth, 44
ITL/BottomLayerMin, 44
ITL/TopLayerDepth, 44
ITL/TopLayerMax, 44

Keken
Keken/RaC, 42
Keken/Viscosity, 42

Particles
Particles/Density, 39
Particles/InvDistPower, 39