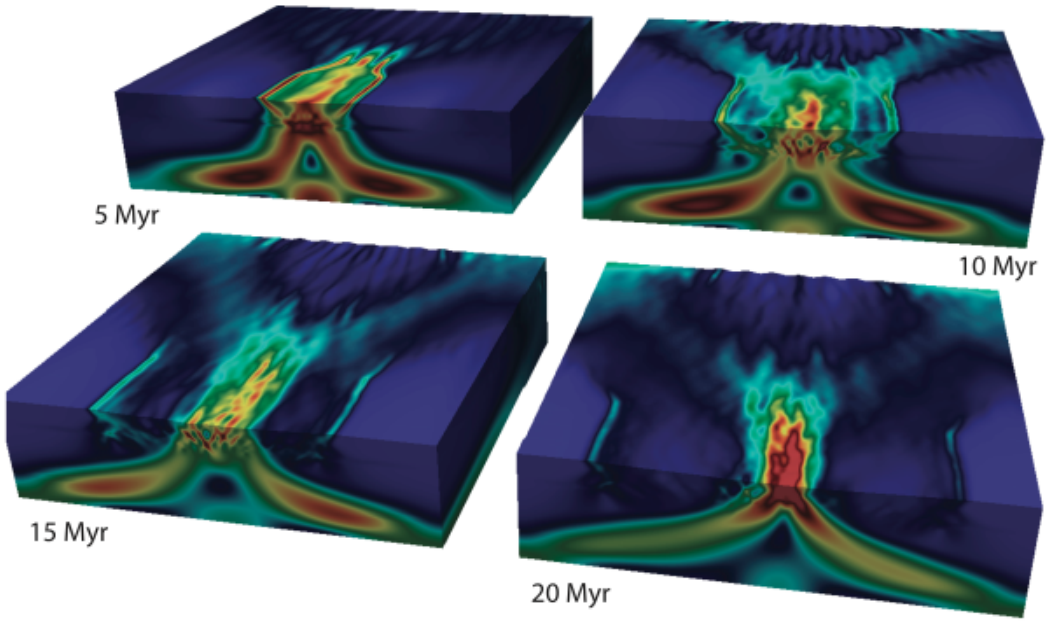
pTatin3D User Guide

April 22, 2021



DEVELOPERS

- Dave A. May: Department of Earth Sciences, University of Oxford (david.may@earth.ox.ac.u
- Laetitia Le Pourhiet: UPMC, Univ. Paris 06 (laetitia.le_pourhiet@upmc.fr)
- Jed Brown: Department of Computer Science, University of Colorado Boulder (jed.brown@colorado.edu)
- Patrick Sanan: Institute of Geophysics, ETH Zurich (patrick.sanan@erdw.ethz.ch)

Contents

1 Introduction

1.1 Overview

PTATIN3D provides a suite of functionality to study long-term geodynamic processes related to the dynamics of the lithosphere and crust. At its core, it provides support for solving non-linear, incompressible Stokes flow problems in three dimensions, using a mixed finite element method together with a marker-and-cell method.

1.2 Referencing

```
@inproceedings{may2014ptatin3d,
  title={pTatin3D: High-performance methods for long-term lithospheric
      dynamics},
  author={May, Dave A and Brown, Jed and Le Pourhiet, Laetitia},
  booktitle={High Performance Computing, Networking, Storage and Analysis,
      SC14: International Conference for},
  pages={274--284},
  year={2014},
  organization={IEEE}
}

@article{may2015scalable,
 title={A scalable, matrix-free multigrid preconditioner for finite element
     discretizations of heterogeneous Stokes flow},
  author={May, Dave A and Brown, Jed and Le Pourhiet, Laetitia},
  journal={Computer Methods in Applied Mechanics and Engineering},
  volume={290},
  pages={496--523},
  year={2015},
  publisher={Elsevier}
}
```

2 Installation

2.1 Software requirements

- PETSc 3.9 or later (essential numerical library)
- git (used to obtain the source)
- Python (non-essential: required to run the test suite)
- ParaView (non-essential: required for visualisation of simulation output)
- libz (non essential: used for output file compression)

2.2 Obtaining the source

The source code for pTatin3D is hosted at BitBucket using the Git version control system. The latex stable version can be obtained with the `git` command

```
git clone https://bitbucket.org/ptatin/ptatin3d
```

which will copy the code into the directory `ptatin/`.

2.3 Compilation

Upon successful installation of PETSc, switch into the root directory of pTatin3D. You should provide a file, named `makefile.arch`, which contains specific compiler optimizations for your operating system and compiler. If you do not provide this, a default, tailored to Mac OS X, will be copied for you. If you are trying to install pTatin3D on another system, consult the directory `config/` where several different compiler/optimization sets have been provided for you. Provided systems include Linux based cluster, Cray XT5/XE6/XC50 and the IBM BlueGene(L,P,Q). You can use these files as a starting point for your particular system by copying the closest architecture file matching your system, e.g. `cp config/machine.arch.XXX makefile.arch`, then make any necessary changes to the new `makefile.arch` to tailor it for your particular operating system / compiler.

Once `makefile.arch` is adjusted for your operating systemtype, type `make -j` from the root directory of pTatin3D.

- Compiling with a particular PETSc build

  ```
  make -j PETSC_DIR=/path/to/your/petsc PETSC_ARCH=your.petsc.build
  ```

- Compiling with a particular compilation flags (C and Fortran)

  ```
  make -j TATIN_CFLAGS='-O2' TATIN_FFLAGS='-fast'
  ```

The following targets for make exist: `all`, `clean`, `drivers`, `test`, `testcheck`, `testall`, `testallcheck`, `releaseinfo`.
As discussed further in Section 5, you can run `make test` to get a first check that pTatin3D has been compiled correctly.
The target `releaseinfo` will automatically update the contents of the file `include/ptatin_version_info.h`. This file is updated to reflect the current revision information associated with the Git master branch. This information is propagated into log files and the output of pTatin3D.

The make system will create all objects, libraries and binary files in the following paths:

- `ptatin3d/${PETSC_ARCH}/obj`
- `ptatin3d/${PETSC_ARCH}/lib`
- `ptatin3d/${PETSC_ARCH}/bin`

The current build system allows you to conveniently compile different pTatin3D using different PETSc builds (e.g. a debug PETSc build versus an optimised build).

2.3.1 Prefix PETSc Builds

pTatin3D also support the use of "prefix" builds, which may be available as modules on clusters, where one supplies `PETSC_DIR` without `PETSC_ARCH`. In this case, the three directories mentioned will be created in the root pTatin3D directory.

2.3.2 GPU Support

PTATIN3D can be configured to compile special operator kernels for use with GPUs. CUDA and/or OPENCL can be enabled by adding `CONFIG_CUDA=y` and/or `CONFIG_OPENCL=y`, respectively, to `makefile.arch`, along with suitable values for compilers, libraries, and include paths. See the examples in `config/` for more.

3 Methods

3.1 Governing equations

We solve the mechancis, energy and transport problem within a time-dependent domain denoted via $\Omega(t)$, with boundary $\partial\Omega(t)$. The outward pointing normal to $\partial\Omega$ is denoted via $\boldsymbol{n}$.

Mechanics

$$\nabla \cdot \eta \left(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T \right) - \nabla p = \boldsymbol{f}_u$$
$$\nabla \cdot \boldsymbol{u} = f_p$$

Boundary conditions:

$$\hat{\boldsymbol{u}}_1 \text{ on } \partial\Omega_1^M$$
$$\hat{u}_2 = \boldsymbol{v} \cdot \boldsymbol{n} \text{ on } \partial\Omega_2^M$$
$$\hat{\boldsymbol{\sigma}}_3 = \boldsymbol{\sigma} \text{ on } \partial\Omega_3^M$$
$$\hat{\boldsymbol{\sigma}}_4 = \boldsymbol{\sigma} \cdot \boldsymbol{n} \text{ on } \partial\Omega_4^M$$

Energy

$$\rho C_p \frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \nabla T + \nabla \cdot (k \nabla T) = f_T$$

Boundary conditions:

$$\hat{T}_1 \text{ on } \partial\Omega_1^E$$
$$q_2 = -k \nabla T \cdot \boldsymbol{n} = 0 \text{ on } \partial\Omega_2^E$$

Transport

$$\frac{D\Psi}{Dt} = g_\Psi,$$

where Ψ is a scalar, vector or tensor.
Boundary conditions:

$$\hat{\Psi}_1 \text{ on } \partial\Omega_1^\Psi,$$

where $\partial\Omega_1^\Psi$ denotes the segment of the boundary where inflow occurs, e.g. $\boldsymbol{v} \cdot \boldsymbol{n} < 0$.
By default, we evolve the material region $\mathcal{R}$ according to

$$\frac{D\mathcal{R}}{Dt} = 0.$$

3.2 Coefficient forms

Mechanics

In discussing the flow laws, we denote the strain-rate tensor via

$$\dot{\boldsymbol{\epsilon}} = \tfrac{1}{2}\left(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T\right)$$

and its second invariant as

$$\dot{\epsilon}_{II} = \sqrt{\tfrac{1}{2}\dot{\epsilon}_{ij}\dot{\epsilon}_{ij}}.$$

Shear viscosity η

- Linear (constant)

$$\eta = \eta_v := f(\mathcal{R})$$

- Linear (temperature dependent)

$$\eta = \eta_v := \eta_0(\mathcal{R})\exp(-\theta(\mathcal{R})T)$$

- Nonlinear (strain-rate and pressure dependent)

$$\eta = \eta_v := A_0(\mathcal{R})\dot{\epsilon}_{II}^{1/n(\mathcal{R})-1} A(\mathcal{R})^{-1/n(\mathcal{R})} \exp\left(\frac{E(\mathcal{R}) + PV(\mathcal{R})}{n(\mathcal{R})RT}\right)$$

- Nonlinear (stress limited)

$$\eta := \frac{\tau_{\text{min}}}{2\dot{\epsilon}_{II}}$$

$$\tau_{\text{min}} = \min(2\dot{\epsilon}_{II}\eta_v, \tau_y)$$

where

$$\tau_y = f(\mathcal{R}, \epsilon) \text{ or } \tau_y = f(\mathcal{R}, P, \epsilon).$$

The variable ϵ defines the accumulated plastic strain which evolves according to

$$\frac{D\epsilon}{Dt} = \begin{cases} \dot{\epsilon}_{II}, & \text{if } 2\eta_v\dot{\epsilon}_{II} > \tau_y \\ 0, & \text{otherwise} \end{cases}$$

Momentum forcing f_u

- Constant

$$\boldsymbol{f}_u = \boldsymbol{f}(\mathcal{R})$$

- Bouyancy

$$\boldsymbol{f}_u = \rho \boldsymbol{g}$$

where the density is defined by either

$$\rho = f(\mathcal{R}),$$

or

$$\rho = \rho_0(1 - \alpha T + \beta P).$$

Continuity forcing f_p

- Typically we use $f_p = 0$.

Energy

Density

- Constant

$$\rho := f(\mathcal{R})$$

Conductivity

- Constant

$$k := f(\mathcal{R})$$

- Temperature dependent

$$k := f(\mathcal{R}, T)$$

Source

- Constant

$$f_T^c := f(\mathcal{R})$$

- Radiogenic

$$f_T^r := H_0(\mathcal{R}) \exp(-t\lambda(\mathcal{R}))$$

- Adiabatic

$$f_T^a := -T\alpha(\mathcal{R})\rho(\mathcal{R})(\boldsymbol{g} \cdot \boldsymbol{v})$$

The final source term f_T can be composed by summing the above terms

$$f_T = f_T^c + f_T^r + f_T^a,$$

where any of the individual terms may be zero.

Simplified conservation law
We also can solve the simpler equation

$$\frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \nabla T + \nabla \cdot (\kappa \nabla T) = f_T.$$

With this form, the following choices are valid:
Diffusivity

- Constant

$$\kappa := f(\mathcal{R})$$

Source

- Constant

$$f_T := f(\mathcal{R})$$

3.3 Numerics

- Mechanics: Q2-P1 mixed finite elements on hexahedral elements
- Energy: overlapping Q1 finite elements (hex) with SUPG stabilization
- Transport: Lagrangian particle (material point method)

4 Drivers

PTATIN3D in itself is a library which aims to enable users to build specific executables to solve a particular class of problems in lithospheric dynamics. Executables which utilize `libptatin3d` are referred to as drivers. For standard problems, e.g. time dependent linear (or non-linear) Stokes problems with optionally thermal coupling, we provide several default drivers. The most important (i.e. useful) of these are summarized below.

1. `ptatin_driver_ic.app` Specifically designed to confirm that the model is defined in accordance with what the developer had in mind. This driver does not solve any equations or perform any time stepping. Rather it only loads the model, loads the initial conditions and boundary conditions and calls the model output function.

2. `ptatin_driver_ts_init.app` A driver for initlalising and solving time dependent, *linear* or *non-linear* Stokes flow problems. The energy equation may be optionally activated within this model.

   To commence a new run, one must: (i) first execute the driver with the additional argument `-init`. With this argument, the solution associated with the initial condition is computed and checkpointed; (ii) execute the driver. When `-init` is not provided, the driver will load a checkpoint file and commence time-stepping. See Section 7 for further details.

   Checkpointing is enabled and restart capabilities are functional.

3. `ptatin_driver_energy.app` A basic driver used for testing the advection diffusion solver. It does not solve Stokes.

4. `ptatin_driver_linear_ts.app` A driver for solving time dependent, *linear* Stokes flow problems. Checkpointing is enabled, however no capabilities for restart is provided. This driver is deprecated and will soon be removed - users should use `ptatin_driver_ts_init.app`.

5. `ptatin_driver_nonlinear_ts.app` A driver for solving time dependent, *non-linear* Stokes flow problems. The energy equation may be optionally activated within this model. Checkpointing is enabled, however no capabilities for restart is provided. This driver is deprecated and will soon be removed - users should use `ptatin_driver_ts_init.app`.

4.1 Options for standard drivers

- `-nsteps`: Number of time steps to perform.
- `-output_frequency`: Interval to output data.
- `-output_path`: Directory where output should be written to.
- `-dt_max`: Upper bound on the time step permitted.
- `-dt_min`: Lower bound on the time step permitted.
- `-constant_dt`: Specifies a constant time step size which should be used.
- `-time_max`: Maximum time (in model units) which simulation should perform.
- `-checkpoint_every`: Indicates interval which check point files with a non-unique name should be written.

- `-checkpoint_every_nsteps`: Indicates interval which check point files with a unique name (associated with the current time step) should be written.
- `-checkpoint_every_ncpumins`: Indicates interval in terms of CPU time with which a check point files with a unique name (associated with the current time step) should be written.

5 Example Models and Tests

A small test suite is provided with the source code for PTATIN3D. These are used to ensure that core functionality is producing the correct results. The tests can be found in the directory `test/`. To execute the tests, ensure that you have Python and Numpy[1] and execute `make test`(a quick subset of tests) or `make testall`(all tests) from the PTATIN3D root directory. These wiill provide instructions on downloading a required test harness script. If on a batch system, after the queueing system has run the tests, you can use

```
make testcheck
```

or

```
make testallcheck
```

to verify the results.
Numerous example models are provided with the source distribution. These serve as a starting point to generate your own models. The most simple model is "viscous_sinker", with source files living in `src/models/viscous_sinker`. Several example options files are provided in `src/models/viscous_sinker/testjobs`. We summarize the examples below

1. `ex_sinker.opts`: Defines a dense, viscous rectangular block in a unit cube domain. The rectangular block sinks due to the buoyancy difference between the block and background material. "Free slip" boundaries (zero normal velocity, zero tangential stress) are applied along all boundaries. The model is time dependent with a linear viscous rheology, thus the driver `ptatin_driver_linear_ts.app` should be used.
2. `sinker-ex2.opts`: Defines a buoyant spherical viscous inclusion in a domain $\Omega = [0,1] \times [0,\frac{1}{4}] \times [0,1]$. Boundary conditions on the upper surface $\sigma_{ij} n_j = 0$ ("free surface") and "free slip" on all other sides. The upper surface is initialized to have a small amount of positive topography. Only a single time step is performed. The driver `ptatin_driver_linear_ts.app` should be used. This option file illustrates how flexible multi-grid coarsening and coarse grid operators can be used. See Sec. 8.1 for further details.
3. `sinker-mfscaling.opts`: Defines a dense, viscous spherical inclusion in a unit cube domain. A"free surface" upper boundary is used and all other boundaries are "free slip". The upper surface is initially flat. Only a single time step is performed. The driver `ptatin_driver_linear_ts.app` should be used. This option file is used for strong scaling experiments.

6 Generated Output

All output generated via PTATIN3D will be written in the directory specified via the option `-output_path`. If this option isn't provided, a default directory called `output` will be generated.

[1] To check, run `python` and then enter `import numpy`, which shouldn't produce an error. (Enter `exit()` to exit.)

6.1 Diagnostics

The time dependent drivers will generate a log file (`ptatin.log-XXX`) and a file containing all options (inputs) used for the current simulation (`ptatin.options-XXX`). Here, `XXX` will be replaced with a year/date/time stamp (e.g. `2013.10.31_11:00:54`) corresponding to when the executable was launched. The time stamp ensures unique files will be give to each model run, thus enabling them to reproduced at a later date. Log files contain information regarding the mesh resolution, memory usage, CPU times associated with each solve and diagnostics about the solver (number of iterations, residuals, etc) and the time step, current time associated with the simulation. The options file generated can be used to re-run PTATIN3D models simply via executing `./ptatin_DRIVER.app -options_file ptatin.options-XXX`.

6.2 Simulation results

PTATIN3D generates output in formats which can be visualized via the open source tool ParaView (www.paraview.org). A comprehensive ParaView tutorial can be found here at http://www.paraview.org/Wiki/The_ParaView_Tutorial. Additional tutorial information can be found here:

- www.bu.edu/tech/about/research/training/online-tutorials/paraview/
- www.cac.cornell.edu/education/Training/data10/VisualizationWithParaView.pdf

A summary of file extensions commonly produced by PTATIN3D is provided below:
`*.pvd`: Contains time dependent datasets.
`*.pvts`: Contains mesh data (nodal fields and cell fields) for the complete mesh.
`*.vts`: Contains mesh data (nodal fields and cell fields) for each sub-domain used in the parallel computation.
`*.pvtu`: Contains particle data for the complete set of material points.
`*.vtu`: Contains particle data for each sub-domain used in the parallel computation.

Depending on the particular output functions called in your model, numerous fields are generated during a PTATIN3D simulation. The term "sub-domain" used above refers to the piece of the mesh assigned to each core. In the mode of parallelism used in PTATIN3D, there is always one sub-domain per core.

6.2.1 Mesh data

Velocity and pressure: Both the velocity and pressure fields are contained in the following set of files:
`timeseries_vp.pvd`,
`step000000_vp.pvts`,
`step000000_vp-subdomain00000.vts`.
Floating point numbers are stored as double precision. All data is stored in a binary format. The function
`pTatin3d_ModelOutput_VelocityPressure_Stokes()`
generates these data files.

Velocity: The velocity field is contained in the following set of files:
`timeseries_v.pvd`,
`step000000_v.pvts`,
`step000000_v-subdomain00000.vts`.
Floating point numbers are stored as single precision. All data is stored in a binary format. The

function
`pTatin3d_ModelOutputLite_Velocity_Stokes()`
generates these data files.

Temperature: The temperature field is contained in the following set of files:
`timeseries_T.pvd`,
`step000000_T.pvts`,
`step000000_T-subdomain00000.vts`.
Floating point numbers are stored as single precision. All data is stored in a binary format. The function
`pTatin3d_ModelOutput_Temperature_Stokes()`
generates these data files.

6.2.2 Material point data

Material point data can be represented in two distinct manners:

1. As a point cloud, e.g. a set of unconnected points x_p with a set of material point properties (e.g η_p, ρ_p, ...) defined at each point coordinate x_p. We refer to this as a "point-wise" representation.

2. As cell constant values defined on the hex mesh (or sub-mesh hex) used to approximate the velocity field. We refer to this as a "cell-wise" representation.

Each form of representing material point data has several advantages and disadvantages. The point-wise representation of material point data has the following limitations: (i) the point-wise representation requires a large amount of storage, and (ii) investigation of the 3D data set is limited as operations such as iso-surfaces, clipping, etc cannot be defined on point clouds. Furthermore, points do not represent a finite volume, thus they do not define a surface. Hence no lighting/shading can be applied, thereby limiting the depth perception of the material point data set. However, the point-wise representation is useful for debugging model setups. The cell-wise representation of material point data overcomes both of the two limitations of the point-wise representation. However, to define the material properties on each cell we have to introduce an interpolation (or projection) between the data defined on the material points and the mesh. This means that the result stored in the cell-wise data representation is only an *approximation* of the true material point data. Cell-wise representation is useful for high-resolution simulations and production runs in which the correctness of model setups has been verified.

Point-wise representation: Material point data fields associated with the basic marker type (`MPField_Std`) are contained in the following set of files:
`timeseries_mpoints_std.pvd`,
`step000000_mpoints_std.pvtu`,
`step000000_mpoints_std-subdomain00000.vtu`.
Floating point numbers are stored in the precision associated with each material point data field. All data is stored in a binary format. The function
`pTatin3d_ModelOutput_MPntStd()`
generates these data files.

A more flexible mechanism to output material point data associated with multiple marker types is provided by the function `SwarmViewGeneric_ParaView()`. Note that this function will not generate a `*.pvd` file for you. All data is stored in a binary format. Marker types which can be output in

these files are identified via the the following names:
`MPField_Std, MPField_Stokes, MPField_StokesPl, MPField_Energy.`

Marker data can alternatively be represented on the mesh used for computing the velocity/-pressure solution. In this case, the material point data field is averaged over each element and represented as a constant value within each element.

Cell-wise representation: Material point data fields which are represented on the mesh are contained in the following set of files:
`timeseries_mpoints_cell.pvd`,
`step000000_mpoints_cell.pvts`,
`step000000_mpoints_cell-subdomain00000.vts`.
Floating point numbers are stored in the precision associated with each material point data field. All data is stored in a binary format. The function
`pTatin3d_ModelOutput_MarkerCellFields()`
generates these data files. This function is flexible and permits specific marker fields to be output into the file. Marker fields which can be output in these files are identified via the the following names: `MPV_region, MPV_viscosity, MPV_density, MPV_plastic_strain, MPV_yield_indicator, MPV_diffusivity, MPV_heat_source.`

7 Checkpointing and Restarting

7.1 General

- All drivers are capable of *generating* checkpoint files - however only `ptatin_driver_ts_init.app` is capable to restarting a job from a checkpoint file.

- All checkpoint files created will reside underneath the following directory:

```
<OUTPUTDIR>/checkpoints
```

  where `<OUTPUTDIR>` was specified via `-output_path`. Each checkpoint is associated with a unique textual name related to the specific stage of the computation, for example the initial condition, or the solution at a given time step. A checkpoint file associated with the initial condition data will be written here

```
<OUTPUTDIR>/checkpoints/initial_condition
```

  whilst a checkpoint associated with step 100 will be written here

```
<OUTPUTDIR>/checkpoints/step100
```

  Within each each of these directories a number of different files are created. The parent checkpoint file for the PTATIN3D is always called

```
ptatin3dctx.json
```

7.2 Controlling checkpoint file creation

Specifying when a checkpoint should occur can be controlled in several different ways. These are summarised below.

1. By enforcing that the code writes a checkpoint file every N steps, but the checkpoint files will will have a common name. These files are over written each time checkpointing occurs. Such functionality is useful when you simply need to restart the code on machine which allows a fixed wall time and you want to job chain. Since the checkpoint files have a common name, job chaining is straight forward. This type of checkpointing is activated via the command line flag
   `-checkpoint_every N`

2. By enforcing that the code writes a checkpoint file every M time steps. These files will have a file name with a unique extension, thus they do not get over-ridden. This type of checkpointing is activated via the command line flag
   `-checkpoint_every_nsteps M`

3. By enforcing that the code writes a checkpoint file every X CPU minutes. This can be useful if you have some time steps which require significantly more time to solve than others. In this case, requesting to check point every M time steps might not be appropriate and you might exceed the wall time before M time steps have completed. This type of checkpointing is activated via the command line flag
   `-checkpoint_every_ncpumins X`

All of the above checkpoint creation options can be used together to preserve the results obtained from a simulation. When they are used together, care is taken to ensure that time step based checkpoint files are not written twice. For example, if I asked to checkpoint every 30 mins, and this criterion is activated at step 10, and the `-checkpoint_every_nsteps` options was used with a value of 10, pTATIN3D will not checkpoint step10 twice.

7.3 Restarting via a checkpoint file

When ever a checkpoint file is written, the file `restart.default` is created into the directory specified via `-output_path`. This file contains a PETSc options specifying the path to the most recent checkpoint file created:

```
# restart.default
-restart_directory path/to/last_checkpoint_file_written
```

To restart from the last checkpoint file written, you just need to either execute

```
./ptatin_driver_ts_init.app
```

By default this driver will look in the directory <OUTPUTDIR> for `restart.default` and if found, it will use the path to the checkpoint file specified via the option `-restart_directory` to restart the run.
To restart from any checkpoint file written, you need to either execute

```
./ptatin_driver_ts_init.app \
  -restart_directory path/to/your/checkpoint_file
```

If no other options are provided, the restarted job will write output into the same directory specified by the value of `-output_path` given to the run which generated the checkpoint file.
By default it is assumed that the output of a restarted job will be written into the same location as the original job. If you wish to direct the output of you restarted run to a directory different from that set by `-output_path` in the original job, you are required to specify the following two options:

```
./ptatin_driver_ts_init.app \
  -restart_directory path/to/your/checkpoint_file \
  -output_path path/to/new/outputdirectory
```

8 Nonlinear and Linear Solvers

The discrete linear Stokes problem can be expressed as

$$\begin{bmatrix} A & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ h \end{bmatrix} \tag{1}$$

We solve Eq. (1) via a right preconditioned Krylov method using an Elman-Wathan style upper block triangular preconditioner. The preconditioned system is thus

$$\begin{bmatrix} A & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} A & B \\ 0 & -S \end{bmatrix}^{-1} \begin{bmatrix} \tilde{u} \\ \tilde{p} \end{bmatrix} = \begin{bmatrix} f \\ h \end{bmatrix} \tag{2}$$

where

$$\begin{bmatrix} \tilde{u} \\ \tilde{p} \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & -S \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} \tag{3}$$

The upper block triangular preconditioner is only required to be applied to vectors. Rather than assembling the inverse and multiplying with an arbitrary vector (x_1, x_2), we re-state the problem in the following manner

$$\begin{bmatrix} A & B \\ 0 & -S \end{bmatrix}^{-1} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad \longrightarrow \quad \text{Solve } \begin{bmatrix} A & B \\ 0 & -S \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ for } y_1, y_2 \tag{4}$$

The solve involving the upper triangular system is defined in two steps:

1. Solve $-S\, y_2 = x_2$ for y_2
2. Solve $A\, y_1 = x_1 - B\, y_2$ for y_1

The iterative solution strategy for the Stokes flow problem is a nested procedure, containing many sub-solves. The nested form of the method used is depicted in Fig. 1. We will refer to the outer most solver (i.e. that for Eq. (2)) as the “Stokes solve”. Any solver associated with the A matrix will be verified to as a “Viscous block” solve. Any solver related to S will be referred to as a “Pressure Schur complement” solve. In the following section we detail how to configure the solver applied to Eq. (2) to obtain (u, p), and how to configure the solvers in steps 1 (`-fieldsplit_u_ksp`) and 2 (`-fieldsplit_p_ksp`) of applying the upper triangular preconditioner.

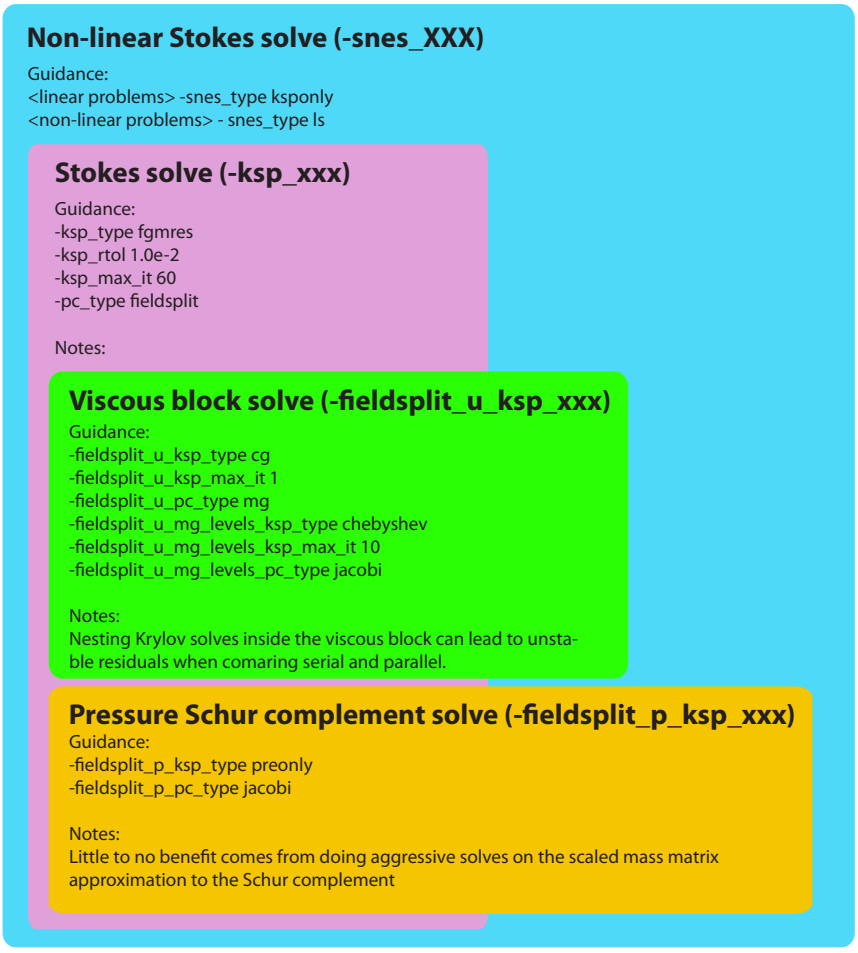


Figure 1: *Solver hierarchy*

8.1 Configuring solvers

8.1.1 Non-linear Stokes solver

Non-linear methods adopt the following sequence

1. Given x_0, set $k = 0$. Compute $\|F(x_0)\|$
2. Solve $J(x_k)\delta x = -F(x_k)$
3. $x_{k+1} = x_k + \delta x$
4. Compute $\|F(x_{k+1})\|$
5. $k = k + 1$, goto step 2 until converged.

The non-linear iteration sequence may be terminated for several different reasons. The reasons and the controlling parameters (modifiable from the command line are provided below):
`-snes_rtol 1.0e-3` : stops if $\|F(x_{k+1})\| \leq 10^{-3}\|F(x_0)\|$
`-snes_atol 1.0e-3` : stops if $\|F(x_{k+1})\| \leq 10^{-3}$
`-snes_stol 1.0e-3` : stops if $\|\delta x\| \leq 10^{-3}\|x_{k+1}\|$
`-snes_max_it 30` : stops if 30 non-linear iterations occur
`-snes_max_funcs 30` : stops if 30 non-linear residual evaluations occur

8.1.2 Linearized Stokes solve

`-ksp_rtol 1.0e-3` : stops if $\|J_k\delta x + F_k\| \leq 10^{-3}\|J_0\delta x + F_0\|$
`-ksp_atol 1.0e-3` : stops if $\|J_k\delta x + F_k\| \leq 10^{-3}$
`-ksp_max_it 30` : stops if 30 Stokes iterations occur

8.1.3 Viscous block solve

`-fieldsplit_u_ksp_rtol 1.0e-3` : stops if $\|Au_k - b\| \leq 10^{-3}\|Au_k - b\|$
`-fieldsplit_u_ksp_atol 1.0e-3` : stops if $\|Au_k - b\| \leq 10^{-3}$
`-fieldsplit_u_ksp_max_it 30` : stops if 30 viscous block iterations occur

8.2 Multi-grid sequencing

The number of multi-grid levels is set via the option:

```
-dau_nlevels X
```

If $X = 3$, this will imply that the mesh specified via `-mx 16 -my 16 -mz 16` will be coarsened twice, resulting (by default) in a mesh sequence with the number of elements $M_x = 16$ (fine), $M_x = 8$ and $M_x = 4$ (coarse). For clarity, level 0 is designated as the "coarsest" level, and level $X - 1$ is designated as the "finest" level.

Meshes can be coarsened by a constant factor on each level, or one can control the coarsening factor level-by level. In both cases, the coarsening applied is dependent on the coordinate direction (i, j, k). To define constant coarsening on every level, but varying in each direction, use the options `-da_refine_x`, `-da_refine_y`, and `-da_refine_z`, for example

```
-da_refine_x 4
-da_refine_y 2
-da_refine_z 1
```

Note that different coarsening can be specified in each direction - here denoted via `x,y,z` in the refinement command line option. Also note that a refinement factor of 1 implies that mesh will not be coarsened in that direction.

To define level specific coarsening, use the options `-stk_velocity_da_refine_hierarchy_x`, `-stk_velocity_da_refine_hierarchy_y`, and `-stk_velocity_da_refine_hierarchy_z`, for example

```
-stk_velocity_da_refine_hierarchy_x 4,1
```

The integer parameters are listed from coarse to fine levels (left to right). The above options will result result in the following hierarchy in the x direction (from coarse to fine), $M_x = \{4, 16, 16\}$. Note that if you have X levels, you are only required to specify $X - 1$ values to define the coarsening hierarchy.

Note that when using geometric multi-grid, all levels are spatial decomposed across all cores. Furthermore, it is required that every core contains at least one element. Thus, the total number of cores used to run a PTATIN3D job must not exceed the number of elements within the coarse grid. If you require additional levels in your multi-grid hierarchy, you may

- an algebraic multi-grid preconditioner as your coarse grid preconditioner, or
- use PETSC's `PCTELESCOPE` to reduce the number of processors in the coarse grid communicator.

8.3 Coarse grid operators

Within PTATIN3D, the style of coarse grid operator can be configured at run time. We provide support to enable four types of coarse grid operator; { assembled re-discretised = 0, matrix-free re-discretised = 1, Galerkin = 2, matrix-free Galerkin = 3 }. Re-discretised operators imply that the viscosity is projected through the mesh hierarchy and the finite element operators are re-evaluated using the standard FE weak form defined on the fine level. Galerkin coarsening implies that the coarse grid operator at level k A_k is defined via $A_k = R^T A_{k+1} R$, where R is the restriction operator for level $k+1$ to k. To define the triple matrix product required by the a Galerkin operator definition on level k, it is require that the operator on level $k+1$ be assembled, i.e. be of either type { assembled re-discretised = 0, Galerkin = 2 }. Configuration of the coarse grid operator type is define via the option

```
-A11_operator_type
```

, for example

```
-A11_operator_type 2,2,0,1
```

where valid indices are $\{0, 1, 2, 3\}$ and operators on each level are specified from coarse to fine (left to right).

8.4 Multi-grid performance profiling

To activate monitors which report CPU times on each multi-grid level, one must specify the following options

```
-dau_nlevels X -fieldsplit_u_pc_type mg -fieldsplit_u_pc_mg_levels X -fieldsplit_u_pc_mg_log -log_summary
```

Note that if the values for `-dau_nlevels X` and `-fieldsplit_u_pc_mg_levels X` do not match, logging will not be performed.

8.5 Understanding residuals

When solving linear problems, two types of residuals are important to monitor for both accuracy of the Stokes problem and efficiency of the solver. By default, all drivers report the Stokes residuals for the individual components of the residual associated with the u, v, w-momentum equations and the continuity equation. The complete residual associated with the residual of the full vector (u, v, w, p) can be monitored using the flag

```
-ksp_monitor
```

The residuals output to the screen will typically look like the following:

```
0 KSP Component U,V,W,P residual norm [ 0.000000000000e+00, 2.307691625955e-03, 0.000000000000e+00, 0.000000000000e+00 ]
      Residual norms for fieldsplit_u_ solve.
      0 KSP Residual norm 1.000000000000e+00
      1 KSP Residual norm 9.566218971472e-01
      2 KSP Residual norm 6.120179419948e-01
      3 KSP Residual norm 2.693954462485e-01
      4 KSP Residual norm 7.305148723409e-02
      5 KSP Residual norm 2.484961341074e-02
      6 KSP Residual norm 7.872576358279e-03
1 KSP Component U,V,W,P residual norm [ 1.931850895378e-07, 2.249650371310e-03, 1.931669353791e-07, 3.613480131486e-04 ]
      Residual norms for fieldsplit_u_ solve.
      0 KSP Residual norm 9.383540923375e-01
      1 KSP Residual norm 2.353199854638e-01
      2 KSP Residual norm 6.233285019473e-02
      3 KSP Residual norm 2.466199083002e-02
      4 KSP Residual norm 6.082964600394e-03
2 KSP Component U,V,W,P residual norm [ 2.862389390637e-05, 9.461315464033e-04, 2.873233657833e-05, 1.128984116051e-03 ]
      Residual norms for fieldsplit_u_ solve.
      0 KSP Residual norm 1.461862868221e+01
      1 KSP Residual norm 6.787824394358e-01
      2 KSP Residual norm 1.306125161069e-01
3 KSP Component U,V,W,P residual norm [ 1.614279704240e-04, 4.795734857592e-04, 1.613914106944e-04, 8.488885268501e-04 ]
      Residual norms for fieldsplit_u_ solve.
      0 KSP Residual norm 3.614067424860e+01
      1 KSP Residual norm 1.727513541857e+00
      2 KSP Residual norm 1.701193913599e-01
4 KSP Component U,V,W,P residual norm [ 1.182053444264e-04, 2.177408865879e-04, 1.181950335293e-04, 3.963394470286e-04 ]
```

The longer lines containing the "`U,V,W,P residual norm`" relate to the residuals of the Stokes problem, whilst the shorter lines under each instance of "`Residual norms for fieldsplit_u_ solve.`" relate to the viscous block solver which is performed as the Stokes preconditioner is applied.

8.6 Trouble shooting convergence problems

Convergence failure of Stokes solver

1. *Elements possess a high aspect ratio (AR)*. When using Q_2-P_1 elements, the inf-sup constant increases with increasing AR. Possible fixes: (1) Strengthen the Schur complement preconditioner. This could be achieved by `-fieldsplit_p_pc_type lu`, or switching to use an approximate form of $S = B^T A^{-1} B$ via say one application of a Krylov method preconditioned with the scaled lumped mass matrix. Increasing the strength of the Schur complement preconditioner can be achieved by making the solve on A^{-1} more accurate and making the solve on S more accurate; (2) modify the element resolution to make AR < 10.
2. *Convergence failure of the viscous block*. This can be diagnosed by running with these options; `-fieldsplit_u_ksp_monitor_true_residual -fieldsplit_u_ksp_converged_reason`. See below for more information.

Convergence failure of viscous block

1. *MG smoother is not strong enough*. Possible fixes: (1) Increase the number of smoothing applications via `-fieldsplit_u_mg_levels_ksp_max_it`; (2) completely re-configure the smoother for

example try `-fieldsplit_u_mg_levels_ksp_type fgmres -fieldsplit_u_mg_levels_ksp_max_it 4 -fieldsplit_u_mg_levels_pc_typ bjacobi`. Not that this will require assembling the coarse grid operators; (3) Use more robust coarse grid operators, e.g. assemble some levels and use Galerkin coarsening on the coarser levels; (4) Use less levels, or try aggressive coarsening.

2. *MG coarse grid solver fails to converge*. This can be diagnosed by running with these options; `-fieldsplit_u_mg_coarse_ksp_monitor_true_residual -fieldsplit_u_mg_coarse_ksp_converged_reason`. Possible fixes: (1) Improve the coarse grid solver. See below for more details.

Convergence failure of viscous block, GMG coarse grid solver

1. *MG coarse grid preconditioner isn't strong enough*. Possible fixes: (1) Use algebraic multigrid (ML, Hypre) as a preconditioner; (2) Use heavy sub-domain preconditioners, e.g. `-fieldsplit_u_mg_coarse_pc_type asm -fieldsplit_u_mg_coarse_pc_asm_overlap 2 -fieldsplit_u_mg_coarse_sub_pc_type ilu -fieldsplit_u_mg_coarse_sub_pc_factor_levels 2`; (3) Try using a semi-redundant approach where you accumulate the coarse grid problem on a sub-set of cores and use stronger sub-domain preconditioners.

Convergence failure of non-linear Stokes solver

1. *Coming soon....*

8.7 Using optimized operators

pTatin3D should automatically use optimized kernels for matrix-free application of the operators A and B in Equations 1 and 2. Additional control over the A operator (also known as the viscous block or the A_{11} operator)is achieved with a command line flag

```
-a11_op <ref,tensor,avx,opencl,cuda,subrepart>
```

The default is `avx` if your compiler detects that AVX is available, and otherwise `tensor`. See the initial pTatin3D output for a note about whether or not AVX was detected. Note that you may need to provide a special compiler flag in `makefile.arch`, such as `-march=native`, to enable AVX.
`opencl` and `cuda` kernels require that you have configured pTatin3D to use OpenCL and CUDA, respectively; see Section 2.3.2. Note that these kernels assume one GPU per rank.
If you have AVX, CUDA, and MPI-3 shared memory constructs [2] available, you can use `-a11_op subrepart`, which uses the CUDA and AVX kernels simultaneously, assuming 1 GPU per shared-memory domain. It requires specification of a parameter describing what fraction of the work (local elements) to allocate to the GPU,

```
-subrepart_frac <value 0-1>
```

This value should be tuned until good load balance is achieved between the single rank per node which uses the GPU, and the others which use only the CPU. Good balanced is achieved when a time "ratio" value of 1.0 is observed in the PETSc logs in the `MatMultMFA11_sub` row.
You may also experiment with using more than one OpenMP (hyper)thread per rank.

[2] In particular, a working `MPI_Comm_split_type()`

9 Geometry

9.1 Modelling domain

The underlying coordinate system used in pTatin3D is Cartesian. In Fig. 2 we identify the face labels used.

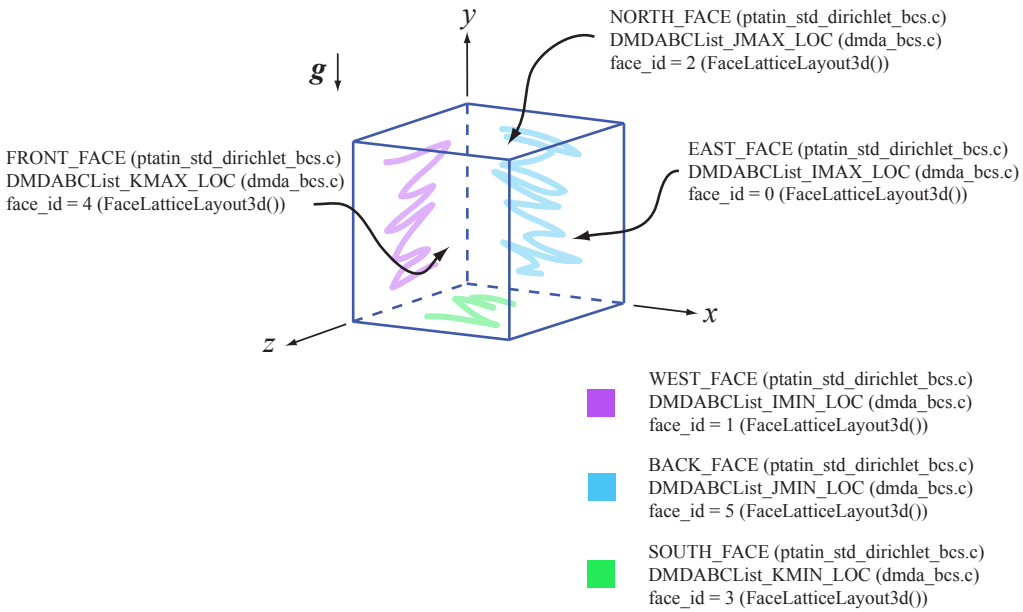


Figure 2: *Natural coordinate system used by the physical domain* Ω *in* pTatin3D*. Labels indicate various names given by different structures functionality.*

10 Building Models

10.1 Code structure

The idea we have for building models is that models should be separated as much as possible from the core functionality of pTatin3D. That is, we want to separate boundary conditions, mesh geometry, etc. as much as possible from the solver used to solve the underlying PDE (Stokes).

For this reason, we have adopted a code structure in which models are compiled and combined into a separate, stand alone library. When using a pTatin3D driver (a program which solves something), we link the model library against the solver library.

Models live under the directory

```
ptatin3d/src/models/
```

and the models are compiled into the static library `libptatin3dmodels.a` An example model called "template" is provided in `src/models`.

To add a model, you need to add the path to and the filename of *all files within your model*

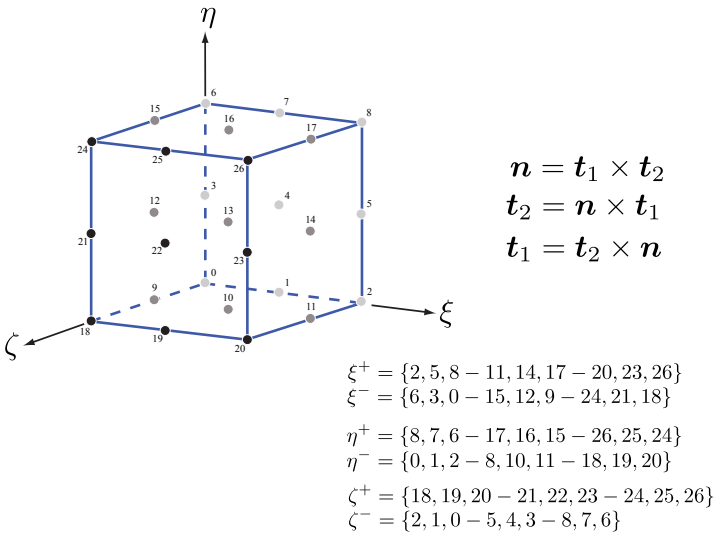


Figure 3: *Nomenclature and basis function ordering for the volume and surface basis functions. ξ, η, ζ are the spatial coordinates in local element coordinate system. The notation ξ^+, ξ^- indicates faces along which $\xi = 1, \xi = -1$ respectively. The convention associated with defining outward pointing surface normals $(\boldsymbol{n})$ and the two tangent vectors to the surface $(\boldsymbol{t}_1, \boldsymbol{t}_2)$ are indicated.*

directory into the file

```
ptatin3d/src/models/local.mk
```

A typical snippet of this file is shown below. Your files should replace the string `<new_model_files>`.

```
# ptatin3d/src/models/local.mk
libptatin3dmodels-y.c += $(call thisdir, \
  ptatin_models_reg.c \
  template/model_ops_template.c \
  <new_model_files> \
 )
```

Importantly, all lines must end with forward slash character (/).
Within the directory

```
ptatin3d/src/models/template
```

are the files which define any specific data structures needed for the model, (`model_template_ctx.h`) and a file (`model_ops_template.c`) containing a complete (yet empty) model description. A "model" description in pTatin3D consists of defining the following operations.

1. `FP_pTatinModel_Initialize` (non essential) Initialize any model specific options, and or model specific parameters in your user defined model context.

2. `FP_pTatinModel_ApplyBoundaryCondition` Define boundary conditions for the PDE (Stokes, energy).

3. `FP_pTatinModel_ApplyBoundaryConditionMG` Define boundary conditions for velocity on each multi-grid level.

4. `FP_pTatinModel_ApplyMaterialBoundaryCondition` (non essential) Define the influx of material points if you have prescribed boundary conditions for velocity which are such that $\boldsymbol{u} \cdot \boldsymbol{n} < 0$ (e.g. inflow boundary conditions).

5. `FP_pTatinModel_ApplyInitialSolution` (non essential for Stokes) Define initial values in the velocity, pressure (non essential) and temperature (essential) vectors. For the Stokes variables $(\boldsymbol{u}, p)$, specifying an initial value may improve convergence of the Stokes solve on the first time step (e.g. by introducing a hydrostatic pressure gradient in the pressure vector).

6. `FP_pTatinModel_ApplyInitialMeshGeometry` Define the geometry of the mesh. Typically this is done simply by described a hexahedral domain via the PETSc function `DMDASetUniformCoordinates()`.

7. `FP_pTatinModel_ApplyInitialMaterialGeometry` Define the initial geometry of the lithology on the markers. (e.g. specify rheology)

8. `FP_pTatinModel_UpdateMeshGeometry` (non essential) Define how the mesh should evolve with time. In some models the mesh remains fixed in space through out time, thus this function need not be defined. Other models may wish to deform the mesh with the velocity vector, or may wish to advect the free surface and then apply remeshing within the interior of the domain. Such prescription of ALE mesh movement should be specified here.

9. `FP_pTatinModel_Output` (non essential) Specify what mesh fields (e.g. velocity, pressure, temperature) and marker fields will be outputted. Numerous methods to output objects from pTatin3D are provided. Any model specific output functions should be called here.

10. `FP_pTatinModel_Destroy` Upon completion of a pTatin3D job, this function will be called to release the memory allocated which is associated with this particular model. If no data structures were defined, then this function does not need to be defined.

Following the definition of the above functions, to complete the model definition we have to perform the following steps :

1. `pTatinModelCreate()` Calling this creates a little structure to hold your function pointers and other model related information.

2. `pTatinModelSetName(...,MODELNAME);` The variable in `MODELNAME`, will used as the command line argument used to select the model `-ptatin_model MODELNAME`

3. `pTatinModelSetUserData()` Set any data structures required by the model.

4. Assign the function pointers. This is done via `pTatinModelSetFunctionPointer()`. The second argument indicates which function pointer is used. These are defined via `typedef enum { } pTatinModelOperation;` and are declared in `ptatin_models.h` To assign the operations which your model will perform, a helper function (`pTatinModelSetFunctionPointer()`) is provided. Example usage:

```
pTatinModelSetFunctionPointer(...,PTATIN_MODEL_APPLY_BC,my_apply_bc_function);

pTatinModelSetFunctionPointer(...,PTATIN_MODEL_APPLY_INIT_MAT_GEOM,my_apply_init_material_geom_function);
```

5. Register the model via the call `pTatinModelRegister()` This will add your model definition into a list which pTatin3D will have access to. Steps 1-5 are provided within the function `pTatinModelRegister_Template()`.

6. Finally, you need to edit `ptatin_models_reg.c` and add the function call to register your model. This should be done within `pTatinModelRegisterAll()` as you'll note, you will see the "template" model registration function `pTatinModelRegister_Template()`;

   So you don't have the compiler warning about "implicitly defined function", simply add the prototype as an `extern`, e.g.

   ```
   extern PetscErrorCode pTatinModelRegister_Template(void);
   ```

10.2 Compiling your model

Models must be compiled at the root level of the pTatin3D. That is, to make a model, you must run "make all" from `ptatin3d` You *should not* run make from within the model directory `ptatin3d/src/models` or `ptatin3d/src` or `ptatin3d/src/models/template`.

10.3 Implementing Dirichlet boundary conditions

pTatin3D uses "iterators" to assist you in defining boundary and initial conditions. Iterators allow you to apply user defined functions to set boundary and initial conditions into the data structures and solution vectors used by pTatin3D without requiring the user to see the underlying data structures used, or understand parallelism.

To apply boundary conditions, the main function used is:

```
   PetscErrorCode DMDABCListTraverse3d(
BCList list,
DM da,
DMDABCListConstraintLoc doflocation,
PetscInt dof_idx,
PetscBool (*evaluate_boundary_condition)(PetscScalar*,PetscScalar*,void*),
void *ctx)
```

which is declared in `dmda_bcs.c`, where the input arguments are identified as;

`BCList list` - data structure used to store boundary conditions.

`DM da` - data structure used to represent the mesh and quantity (e.g. velocity).

`DMDABCListConstraintLoc doflocation` - index identifying which region of the mesh you wish to attach a boundary condition to. The value used here should be one of { `DMDABCList_INTERIOR_LOC`, `DMDABCList_IMIN_LOC`, `DMDABCList_IMAX_LOC`, `DMDABCList_JMIN_LOC`, `DMDABCList_JMAX_LOC`, `DMDABCList_KMIN_LOC`, `DMDABCList_KMAX_LOC` }. Refer to Fig. 2 for the geometric interpretation of these names.

`PetscInt dof_idx` - integer identifying which degree of freedom the constraint should be applied to (.e.g v_x would correspond to 0, v_y would correspond to 1).

`PetscBool (*evaluate_boundary_condition)(PetscScalar*,PetscScalar*,void*)` - the user provided function which defines the boundary condition.

`void *ctx` - any data structure, or parameters which are required by the user provided function to evaluate the boundary condition.

The calling sequence for evaluate_boundary_condition is,

```
PetscBool evaluate_boundary_condition( PetscScalar position[], PetscScalar *value, void *ctx )
```

where the arguments are;

PetscScalar position[] - coordinates in space where the boundary condition is evaluated.

void *ctx - user defined data require to evaluate the boundary condition.

PetscScalar *value - the actual value of the boundary condition [OUTPUT].

PetscBool - the function must return { PETSC_TRUE, PETSC_FALSE } to indicate whether the boundary condition should be applied at this location [OUTPUT].

For a simple example of a user defined boundary condition function, refer to

```
PetscBool BCListEvaluator_constant(PetscScalar position[],PetscScalar *value,void *ctx)
```

in file dmda_bcs.c.

A standard set of velocity Dirichlet boundary conditions for Stokes are defined in ptatin_std_dirichlet_boundary_conditions. Please refer to ptatin_std_dirichlet_boundary_conditions.h for a summary of the functions available. Notes:

1. Boundary conditions can be applied within the interior of the domain using the value DMDABCList_INTERIOR_LOC.
2. Boundary conditions do not have to be applied along an entire face. Prescribing boundary conditions internal to a face is controlled by whether your user provided function returns PETSC_TRUE or PETSC_FALSE. For example, if the bc evaluator was called with the argument DMDABCList_IMIN_LOC we will traverse along the bottom boundary. The user defined function could return PETSC_TRUE if $x < 0.5$ and PETSC_FALSE otherwise. When PETSC_FALSE is returned the parameter PetscScalar *value is ignored.
3. When defining time dependent boundary conditions, the variable time should be included within the user defined structured (void *ctx) which is passed into DMDABCListTraverse3d().

10.4 Implementing initial conditions

Iterators are provided to allow to define the contents of PETSc vectors via user defined functions. This is particularly useful to define initial conditions, e.g. for the temperature field. The procedure is similar to that used to define boundary conditions described above. The two iterators used to fill in values within a vector are DMDAVecTraverse3d and DMDAVecTraverseIJK which are declared in dmda_iterator.c. Below we describe the calling pattern of these two functions.

The standard vector iterator is given by

```
PetscErrorCode DMDAVecTraverse3d(
DM da,
Vec X,
PetscInt dof_idx,
PetscBool (*evaluate_function)(PetscScalar*,PetscScalar*,void*),
void *ctx),
```

where the input arguments are identified as;

DM da - data structure used to represent the mesh and quantity (e.g. velocity).

`Vec X` - PETSc vector into which you want to insert values (e.g. velocity).

`PetscInt dof_idx` - integer identifying which degree of freedom the constraint should be applied to (.e.g v_x would correspond to 0, v_y would correspond to 1).

`PetscBool (*evaluate_function)(PetscScalar*,PetscScalar*,void*)` - the user provided function.

`void *ctx` - any data structure or parameters which are required to evaluate the the user provided function.

The calling sequence for `evaluate_function` is,

```
PetscBool evaluate_function( PetscScalar position[], PetscScalar *value, void *ctx )
```

where the arguments are

`PetscScalar position[]` - coordinates in space where the user defined function will be evaluated.

`void *ctx` - user defined data require to evaluate the boundary condition.

`PetscScalar *value` - the result of the user provided function [OUTPUT].

`PetscBool` - the function must return $\{$ `PETSC_TRUE`, `PETSC_FALSE` $\}$ to indicate whether the value should be inserted into the vector [OUTPUT].

The standard ijk vector iterator is given by

```
PetscErrorCode DMDAVecTraverseIJK(
DM da,
Vec X,
PetscInt dof_idx,
PetscBool (*evaluate_functionIJK)(PetscScalar*,PetscInt*,PetscInt*,PetscScalar*,void*),
void *ctx),
```

where the input arguments are identified as;

`DM da` - data structure used to represent the mesh and quantity (e.g. velocity).

`Vec X` - PETSc vector into which you want to insert values (e.g. velocity).

`PetscInt dof_idx` - integer identifying which degree of freedom the constraint should be applied to (.e.g v_x would correspond to 0, v_y would correspond to 1).

`PetscBool (*evaluate_functionIJK)(PetscScalar*,PetscInt*,PetscInt*,PetscScalar*,void*)` - the user provided function.

`void *ctx` - any data structure or parameters which are required to evaluate the the user provided function.

The calling sequence for `evaluate_functionIJK` is,

```
PetscBool evaluate_functionIJK( PetscScalar position[], PetscInt global_index[], PetscInt local_index[],
PetscScalar *value, void *ctx )
```

where the arguments are

`PetscScalar position[]` - coordinates in space where the user defined function will be evaluated.

`PetscInt global_index[]` - i, j, k values of the node in the global numbering system.

`PetscInt local_index[]` - i, j, k values of the node in the local numbering system.

`void *ctx` - user defined data require to evaluate the boundary condition.

`PetscScalar *value` - the result of the user provided function [OUTPUT].

`PetscBool` - the function must return { `PETSC_TRUE`, `PETSC_FALSE` } to indicate whether the value should be inserted into the vector [OUTPUT].

11 Passive Swarms

11.1 Introduction

The PSwarm object provides a convenient way to track deformation and other quantities of interest (e.g. temperature) at discrete locations in the model domain. The PSwarm object defines a "passive swarm", a set of discrete particles with no assumed connectivity to any other particle in the swarm. The particles are contained within the model domain and do not influence the flow field, or interact with each other (hence the usage of the word passive).

11.2 Defining passive swarms

A single passive swarm can be created with the following code

```
PSwarm pswarm;

PSwarmCreate(PETSC_COMM_WORLD,&pswarm);
```

Any arbitrary number of passive swarms can be created at run-time with the following code:

```
PSwarm *pswarm_array;

PSwarmCreateMultipleInstances(PETSC_COMM_WORLD,&pswarm_array);
```

together with the the command line option

```
        -pswarm_list xx,yy
```

where `xx` and `yy` define the unique names you wish to assign to each passive swarm. In this example, since two names are provided, two passive swarms are created. If the option `-pswarm_list` was not found, `pswarm_array` will be returned as an array of NULL pointers.

11.3 Options

The definition and behaviour of the PSwarm object is configurable at run-time via command line options. When only a single PSwarm object is created, a unique name is not required to identify the PSwarm. In this case, all options take take the form

```
-pswarm_<OPTION_NAME> <VALUE>
```

When multiple PSwarm objects have been defined, the options will be denoted via

```
-###_pswarm_<OPTION_NAME> <VALUE>
```

where ### is be the name used to identify each passive swarm (see Section 11.2)
For the options to take effect, one is required to call

```
    PSwarmSetFromOptions(pswarm);
```

An example of how to create and traverse through multiple PSwarm objects is shown below.

```
PSwarm *pswarm_array, *pswarm;

PSwarmCreateMultipleInstances(PETSC_COMM_WORLD,&pswarm_array);
pswarm = &pswarm_array[0];
while (*pswarm && pswarm_array) {
  PSwarmSetFromOptions(*pswarm);
  pswarm++;
}
```

11.4 Defining initial coordinates

Several choices are possible to define the initial coordinates of the points in each passive swarm. The method used to define the coordinate is controlled via the command line argument

```
-###_pswarm_coord_layout
```

Valid choices for `-###_pswarm_coord_layout` are `1`, `2`, `3` The default value is `1`. The methods of defining the coordinates are described below

11.4.1 Filling a sub-set of the velocity mesh (`-###_pswarm_coord_layout 1`)

Within each Q2 element, the user can define a number of points in each i, j, k direction via

```
-###_pswarm_lattice_nx 3,4,5
```

By default, all elements are filled with points. If desired, a box shaped clipping mask can be applied. Points located within the box are kept whilst all other points are ignored. The dimensions of the box are defined via

```
-###_pswarm_lattice_min 0.0,0.0,0.0
-###_pswarm_lattice_max 1.0,2.0,3.0
```

Options example:

```
-###_pswarm_coord_layout 1
-###_pswarm_lattice_nx 3,4,5
-###_pswarm_lattice_min 0.0,0.0,0.0
-###_pswarm_lattice_max 1.0,2.0,3.0
```

Note that in general, this point creation method does not allow you to precisely control the total number of points which will exists within the passive swarm as it depends on the resolution of the finite element mesh.

11.4.2 Filling a box (-###_pswarm_coord_layout 2)

This method allows you to define a box geometry with a fixed number of points in each i, j, k direction.
Options example:

```
-###_pswarm_coord_layout 2
-###_pswarm_box_nx 3,4,5
-###_pswarm_box_min 0.3,0.4,0.5
-###_pswarm_box_max 0.6,0.8,1.0
```

The above defines a domain $[0.3, 0.6] \times [0.4, 0.8] \times [0.5, 1.0]$ and fills this domain with 3 points in the x- direction, and $4, 5$ points in the y-, z- directions respectively. The number of points specified in each direction must be greater than 1. **Note: this design limitation could be easily fixed.** Point coordinates generated which reside outside the background finite element mesh are ignored and not included within the passive swarm.

11.4.3 Using a user defined coordinate list (-###_pswarm_coord_layout 3)

This method allows you to exactly specify the initial location of the passive swarm points by providing a list of x, y and z coordinates.
Options example:

```
-###_pswarm_coord_layout 3
-###_pswarm_coor_n 4
-###_pswarm_coor_x 0.4,0.6,0.4,0.6
-###_pswarm_coor_y 0.1,0.1,0.3,0.3
-###_pswarm_coor_z 0.5,0.5,0.5,0.5
```

will define 4 particles with the coordinates $(0.4, 0.1, 0.5)$, $(0.6, 0.1, 0.5)$, $(0.4, 0.3, 0.5)$ and $(0.6, 0.3, 0.5)$. Point coordinates specified which reside outside the background finite element mesh are ignored and not included within the passive swarm.

11.5 Visualisation

Assuming we have a PSwarm declared as follows

```
PSwarm pswarm;
```

and where `slab` defines the unique names assigned to each passive swarm.
Calling the viewer function

```
PSwarmViewInfo(pswarm);
```

will report meta-data about your swarm to stdout. Calling

```
PSwarmView(pswarm,PSW_VT_SINGLETON);
```

will create a single VTU file across all MPI-rank in the given communicator. Given that passive swarms may be sparsely distributed throughout the finite element mesh, this output generation style is preferred (even though it may not be the fastest approach). This method will generate the following output:

- timeseries_slab_pswarm.pvd
- step****/step****_slab_pswarm.vtu

Lastly, setting the last arg to the value PSW_VT_PERRANK, e.g.

```
PSwarmView(pswarm,PSW_VT_PERRANK);
```

will create a VTU file per MPI-rank. This method will generate the following output:

- timeseries_slab_pswarm.pvd
- step****/step****_slab_pswarm-subdomain00000.vtu
- step****/step****_slab_pswarm.pvtu

11.6 Defining the transport mode

Passive swarms can be defined to remain stationary over time, or to be advected with the fluid velocity. We refer to this choice as a "transport mode". By default, all passive swarms are assumed to utilize a Lagrangian transport mode (e.g. they move with the fluid velocity). If you require swarms to be stationary, use the command line option

```
-###_pswarm_transport_mode_eulerian
```

11.7 Defining the field update methods

A number of quantities can optionally be computed, or updated on each passive swarm. For example we can compute the pressure, temperature or track the finite strain. Such tasks are referred to as "field update" methods. By default, swarms do not have any field update methods specified.
On any given swarm, the user may request any number of field update methods be applied. For instance, swarm "xx" may have no field updates specified, whilst swarm "yy" may be asked to track pressure and finite strain. This would be invoked via the following command line arguments:

```
-yy_pswarm_field_update_pressure
-yy_pswarm_field_update_finite_strain
```

NOTE: Only a field update method for pressure is implemented

11.8 Defining the region index of a passive swarm

For convenience, we optionally allow users to set a region index on all points within a passive swarm. This is achieved via the command line option

```
-###_pswarm_region_index 3
```

which would assign a region index of 3 to each point in this passive swarm. The default region index is set to 0.

11.9 Post-processing finite strain

A simple recipe to post-process finite strain is described below. Here will define an initial set of points within a box, and track the deformation of the box over time.
The script requires ParaView, and furthermore the build of ParaView must have Python support enabled. We will be using the python wrapper provided by ParaView called `pvpython`. For binary builds on OSX, this can be located here:

```
/Applications/paraview.app/Contents/bin/pvpython
```

Procedure

1. Use the coordinate layout option

   ```
   -pswarm_coord_layout 2
   ```

   Upon executing of the PSwarm coordinate generation code, the following file will be created in your output directory

   ```
   deformation_grid_ref.vts
   ```

2. Within your model output function, call either `PSwarmView(pswarm,PSW_VT_SINGLETON)` or `PSwarmView(pswarm,PSW_VT_PERRANK)` to generate snapshots of the passive swarm.

3. To generate a deformation grid for a given time step, you need to execute the following script

   ```
   pvpython utils/python-post-proc/extract-dgrid.py \
     -i path/to/inputfile.vtu \
     -b path/to/deformation_grid_ref.vts \
     -o outfilename.vts
   ```

   The file created `outfilename.vts`, contains a hex-mesh representation of your deformed box which can be loaded and rendered by ParaView.

In step 3., the file specified by the `-i` must be a vtu file and not a pvtu file. Either format can be generated by PSwarm object (e.g. `PSW_VT_SINGLETON` vs. `PSW_VT_PERRANK`). A script is provided to convert a pvtu file associated with your passive swarm into a vtu file. To do this, use

```
pvpython utils/filters/convert_pvtu2vtu.py \
  -i path/to/inputfile.pvtu
```

This will create the following file

```
path/to/inputfile.vtu
```

NOTE: deformation grid reference file is not PREFIXED

12 Non-dimensional scaling

The preferred choice of scaling is the following; the users chooses a characteristic velocity, length and viscosity v^*, x^*, η^*. These choices define the following non-dimensional variables, $v' = v/v^*$, $x' = x/x^*$, $\eta' = \eta/\eta^*$. Given the Stokes problem

$$\nabla \cdot \eta \left(\nabla v + (\nabla v)^T \right) - \nabla p = \rho g, \qquad \nabla \cdot u = 0,$$

the non-dimensional form thus becomes

$$\nabla' \cdot \eta' \left(\nabla' v' + (\nabla' v')^T \right) - \frac{x^*}{\eta^* v^*} \nabla' p = \frac{{x^*}^2}{\eta^* v^*} \rho g, \qquad \nabla' \cdot u' = 0,$$

where $\nabla' = \frac{1}{x^*} \nabla$. From this the non-dimensional pressure is chosen as

$$p = p^* p', \qquad p^* = \frac{\eta^* v^*}{x^*}.$$

Thus the final non-dimensional form of the Stokes problem is given by:

$$\nabla' \cdot \eta' \left(\nabla' v' + (\nabla' v')^T \right) - \nabla' p' = \frac{{x^*}^2}{\eta^* v^*} \rho g \qquad \nabla' \cdot u' = 0.$$

The right hand side should be interpreted as a non-dimensional force per volume. If desired, we can express the right hand side in terms of a non-dimensional density $\rho = \rho^* \rho'$ and gravity $g = g^* g'$.

$$\nabla' \cdot \eta' \left(\nabla' v' + (\nabla' v')^T \right) - \nabla' p' = \left(\frac{{x^*}^2}{\eta^* v^*} \right) \left(\rho^* \rho' \right) \left(g^* g' \right), \qquad \nabla' \cdot u' = 0.$$

12.1 Additional notes

1. The ideal choice of x^*, v^*, η^* scaling should result in values of velocity and pressure which follow $\bar{v} = \max_k(|v'_k|) \sim 1$ and $\bar{p} = \max_k(|p'_k|) \sim 1$. If your scaling doesn't produce this type of behaviour, then one should be careful in choosing the stopping condition applied to both the non-linear and linear solvers. For example, if $\bar{v} \sim 10^6$, it doesn't make much sense to solve the non-linear equations to a tolerance of $F_u, F_p \sim 10^{-9}$.

2. Convergence of both linear and non-linear solver will appear slow if the scaling if the characteristic size of the velocity and pressure coming out of your model are orders of magnitude different. For instance, if your choice of scaling produces $F_u \sim 10^6$ and $F_p \sim 10^{-1}$ and you stop on $\delta_{\text{abs}} = 10^{-6}$, then a large number of iterations will be required to reduce the 2-norm of (F_u, F_p) to be less than 10^{-6}.

3. Adjusting the scaling parameters to force $\bar{v}, \bar{p} \to 1$: Assuming a model with a linear rheology and a free surface boundary condition, the following observations can be made: (i) increasing v^* by a factor of X will cause v' to *decrease* by a factor of X and p' will be unaffected; (ii) increasing x^* by a factor of X will cause v' to *increase* by a factor of X^2 and p' to *increase* by a factor of X^2; (iii) increasing η^* by a factor of X will not affect v', however p' will *decrease* by a factor of X.