PSD v 2.3

August 25, 2021

Contents

Chapter 1

Introduction

1.1 Introduction

PSD acronym for Parallel Solid/Structural/Seismic Dynamics, is a finite elements-based solid mechanics solver with capabilities of performing High Performance Computing (HPC) simulations with billions of unknowns. The kernel of PSD is wrapped around FreeFEM for finite element discretization, and PETSc for linear algebra/Preconditioning. PSD solver contains straightforward supports for static or dynamic simulations with linear and nonlinear solid mechanics problems. Besides these hybrid-phase field fracture mechanics models have also been incorporated within PSD. For dynamics the generalized-α model for time discretization is used, this models enable straightforward use of Newmark-β, central difference, or HHT as time discretization. PSD uses sate-of-the art domain-decomposition paradigm via vectorial finite elements for parallel computing and all solvers are proven to scale quasi-optimally. PSD has proven scalability uptill 13,000 cores with largest problem solved containing over 5 Billion unknowns.

Besides the parallel suite, PSD also includes a sequential solver which does not require PETSc. PSD works for two and three dimensional problems only. Unstructured meshes (triangular for 2D and tetrahedral for 3D) are supported in MEDIT's .mesh format or Gmsh's .msh format. PSD post processing is done via .pvd ,.vtk and .vtu files of the ParaView platform.

1.2 Changelog

PSD has been maturing and evolving with time, following subsections present the highlights of some key changes made to each PSD version.

1.2.1 Rolling Release (2.3)

Added

- More tests for each plugin and physics, now added in their respective folders.
- Support for Mfront and Mgis interface for non-linear material laws
- New plugin `mfront` can now be loaded in PSD.
- `-withmaterialtensor` now uses Quadrature finite elements to build the material tensor.
- New variational formulation for handling Linear mechanics problems, Quadrature point wise material tensor is built.
- `pseudo_nonlinear` implementation of Linear Elasticity now works.
- New `Notes` section added in the source repo, to help with mathematical and algorithimic resoning.

Changed

-

Bug

- Bug in top-ii-vol meshing due to MPI communication removed.
- Bug in `pseudo_nonlinear` model for Elastodynamics/soildynamics fixed.

1.2.2 2.2 - 2021-07-28

Added

- New and more verbose tutorials on fracture mechanics, soil-dynamics.
- Fast and parallel post processing is now performed using `pvtu` files.
- New PETSc interface in plugins that supports pvtu output.
- Error mechanism to signify wrong PSD flags.

Changed

- Flag values now donot contain hyphens '-' use underscore instead '_', e.g, `linear-elasticity` is now `linear_elasticity`
- 4 CPU procs are now used for `make check`, user can control this by `make check NP=USER_PROCS`.
- Moved to FreeFEM 4.9 and PETSc 3.15.0 .
- Moved to GitLab for hosting the repository.
- New checks for wrong flag. Now if wrong flag or values is entered PSD will give error.
- Boolean flags now also accept value 1|0|yes|no|on|off|true|false for turning on or off.

1.2.3 2.1 - 2021-01-27

Added

- New accurate force calculations via matrix-vector product: new flag `-getreactionforce`.
- New flag `-reactionforce` variational-based | stress-based to get reaction force on a surface.
- New flag `-plotreactionforce` to activate real time pipe plotting using GnuPlot.
- More verbos info on `-help`
- New flag `-mesh` to provide the name of mesh to `PSD_Solve`.
- New flag `-probe` to postprocess FE variables at a point.
- New flag `-crackdirichletcondition` to include a pre-cracked Dirichlet in damage models.
- New tests for more advance top-ii-vol partitioning.
- New flag `-constrainHPF` to enable constrain conditions in hybrid phase-field (WIP).
- New developments in parallel interpolations.
- Tutorials added, use `make tutorials` to install

Changed

- Moved to FreeFEM 4.7-1.
- Moved to PETSc 3.14.
- New version of top-ii-vol v 1.3 support for exascale computing (includes new 2D 3D partitioning)
- `-fastmethod` now replaced by `-withmaterialtensor` (this is now inverse of `-fastmethod`)

Removed

- Flag `-pipegnu` not supported for damage mechanics (to be further deprecated from elsto/soildynamics)

Bugs removed

- Stain vector incorrect numbering in split function of GFP

(see hash 5ec7b882494f71984d07f468b518ec886e942d32)

- Hybrid phase-field with constrain with wrong update

(see has 20ebfbc3cc58b9f1407658543bf3b239e74bd089)

1.2.4 2.0 - 2020-08-18

Added

- New processing via C++, `PSD_PreProcess` binary (MAJOR CHANGE).
- New solving via shell wrapper `PSD_Solve` instead of `FreeFem++` or `FreeFem++-mpi`.
- New examples of using the solver.
- New Pdf documentation containing tutorials, function definitions, verbos on PSD C++ library.
- New option `-nonlinearmethod` (Picard|Newton-Raphsons).
- New time discretization option `-timediscretization`.
- New point boundary conditions.
- New dummy city mesh and analysis 2D for soil dynamics.
- Automatic identification of FreeFEM and Gmsh during `./configure`.
- New flags for `--with-FreeFEM=` and `--with-Gmsh=` during `./configure`.
- New flag `-bodyforceconditions` [int] to include body force.
- New flag `-problem` linear-elasticity | damage | elastodynamics | soildynamics to define physics.
- New flag `-model` to set approximation for damage mechanics hybrid-phase-field | Mazar | pseudo-nonlinear | Hujeux.
- Better energy splitting included Hybrid phase-field compressibility vs tensile energy condition.
- Introduce boundary conditions via `-dirichletconditions` [int] flag.
- Introduce point boundary conditions via `-dirichletpointcondition` [int] flag.
- Introduce traction boundary conditions via `-tractionconditions` [int] flag.
- New folder `tests` containing unit-tests for modules.
- New Hujeux law (nonlinear soil law) interface using C++ class (Thanks to Evelyne Foerster).
- New pseudo-nonlinear model for solving elastodynamics and soildynamics with nonlinear Newton-Raphsons.
- Introduced double couple earthquake source boundary condition for soildynamics.
- New flag `-doublecouple` force-based | displacement-based to use double couple source for soildynamics.
- New top-ii-vol parallel meshing via `-top2vol-meshing` flag (compatible with soildynamics).

Changed

- Moved to FreeFEM 4.6.
- Moved to PETSc 13.13.
- Moved to C++ for preprocessing.
- Moved to shell wrapper PSD_Solve for solving.
- Changes to GFP energydecopostion plugin 'DecompEnergy_Op', now includes compressibility history.
- Replaced GFPDecompEnergy2D/GFPDecompEnergy3D by a generic 2D/3D function GFPSplitEnergy(Eps1[],PsiPlus[],PsiMinus[],HistPlus[],HistMinus[],par);.
- Postprocessing flag `-postprocess` options now support `u` , `v`, `a` , `uv`, `ua`, `av` or `uav`.

Removed

- No more `FFINSTALLDIR` and `GMSH` variables during `make` and `make check`.
- No more `-plot` flag now handled by `-postprocess`.
- No more `-nonlinear` flag now handled by `-problem` and `-model`.
- No more `-bodyforce` flag now handled by an `int` valued `-bodyforceconditions`.
- No more `-dynamic` flag now handled by `-problem` and `-model`.
- No more `-soildynamic` flag now handled by `-problem` and `-model`.
- No more `-quasistatic` flag now handled by `-problem` and `-model`.
- No more `-dirichletbc` flag now handled by `-dirichletconditions`.

Bugs removed

- Bug in 3D paraxial loading (see hash 8fcbe7390e526423cd24b5f0ab1c06899b36c67f)

1.2.5 1.8 - 2020-01-21

Added

- New soil dynamic module `-soildynamics`
- New paraxial element support in 2D.
- New timeplotting support `-timepvd`
- New `-postprocess` option for postprocessing `u` , `v`, `a` , or `uav`.

Changed

- Moved to FreeFEM 4.4.2.
- Moved to PETSc 13.12.
- New simpler way of plotting `savevtk` in parallel with `append` flag for iterative solutions.
- VTU files get stored with a date and time stamp.
- New way of maintaining a logfile for all simulations (date,time,case,..) in `simulation-log.csv`.

1.2.6 1.7 - 2019-11-08

Added

- New mesh reordering via Reverse Cuthill-Mackee via `-useRCM`.
- New quasi-static parallel solver (Extension of B.Masseron & G.Rastiello sequential version).
- New GFP plugin for Mazar's damage update for 2D/3D problems `GFPMazarsDamageUpdate(...)`.
- New MPI plotting routine `plotJustMeshMPI()`.
- New option `-fastmethod` to switch back to default variational formulation.
- New make flag for compiling on supercomputer.

Changed

- Changed variational formulation now using $\epsilon(u) : A : \epsilon(v)$.
- Using GFP becomes optional `-useGFP`.
- Better documentation via `.md` and `.html` files.
- Better plotting support for `PlotMPI()`.
- Moved to FreeFEM 4.4.

1.2.7 1.6 - 2019-06-11

Added

- Dynamic linear solver in 2D and 3D parallel/sequential.
- New finite element variable for partition of unity– for fixing integrals.

Changed

- Better documentation via `.md` and `.html` files.
- Correct quadrature order for faster computations.
- Major changes/splitting of `.script` files.

Removed

- Removed the `BoundaryAndSourceConditions.script` merged with `ControlParameters.script`.

Bugs

- Bug in integrals fixed.

1.2.8 1.5 - 2019-05-29

Added

- Dynamic linear solver in 2D and 3D sequential.
- New meshes for dynamics tests `bar-dynamic.msh`.
- Checking modules `make check`.
- Faster sparsity pattern calculations.

Changed

- Better documentation via `.md` and `.html` files.
- Major restructuring of the codes.
- Moved to `automake` for solver installation.
- Mesh building via `make`.

Removed

- Removed the manufactured solution codes.

1.2.9 1.4 - 2019-05-14

Added

- Fully vectorial finite element solver for phase-filed `-vectorial`.
- New `-supercomp` for avoiding xterm issues on super computers.
- New `MatViz()` function for matrix sparsity visualization.
- Introduced `GFP` plugin support (Go Fast Plugins).

Changed

- Elastic energy decomposition is now optional `-energydecomp`.
- Force calculation using integrals (Thanks to G.Rastiello).

1.2.10 1.3 - 2019-04-08

Added

- New meshes in 2D/3D`Notched-plate` , `square-crack`, etc.
- New fracture mechanics module.
- New `-nonlinear` flag to activate phase-field model for brittle fracture.
- New `-timelog` for time logging the solver.
- New `-pipegnu` for GNUplot piping.

Changed

- Scripting now performed using `.script` files:
  - `BoundaryAndSourceConditions.script`
  - `LinearFormBuilderAndSolver.script`
  - `Macros.script`
  - `Main.script`
  - `VariationalFormulation.script`
  - ....
- Move to FreeFEM version 4.0.
- Move to PETSc version 3.11.

1.2.11 1.2 - 2019-03-18

Added

- Support for Gmsh's `.msh` or Medit's `.mesh` meshes in folder `Meshes`.
- Advance to 3D physics.
- New MPI based parallel solver linear elasticity.
- New approach for solver generation via scripting (PhD thesis MA Badri) with `scriptGenerator.edp`.
- Integrated Domain decomposition macro (PhD thesis MA Badri).
- Customized `.vtk` support for ParaView post-processing.
- New point boundary condition macro `pointbc(Real[int], fespace, matrix)`.
- New flags for communicating with the solver: `-dimension`, `-plot`, `-bodyforce`, `-lagrange`, etc.

Changed

- More advance README.MD.
- Sequential solver now merged within scripting via flag `-sequential`.
- Move to FreeFEM version 3.62.
- Moved manufactured solutions to `validation-test` folder.

1.2.12 1.1 - 2019-03-04

Added

- Initial FreeFEM files for sequential linear elasticity in 2D (case of constrained bar).
- More cases of manufactured solution for linear elasticity in 2D.
- Added README.MD for explaining the solver.
- ParaView plotting activated.

Changed

- Moved to Tuleap git hosting from CEA.
- Separate folder of manufactured solutions and the linear elastic solver.
- Move to FreeFEM version 3.61.

1.2.13 1.0 - 2019-02-15

Added

- Initial FreeFEM files Method of manufactured solution for linear elasticity in 2D.

1.2.14 Version git tags

Version	Git tag
1.0	8a8ecb2746b7da792073358c60df33bae647f788
1.1	a667e6085ba1f92f8dd619bd40e18f85c593bc0a
1.2	e48b7b3a30c05ad4c343efa6a17fee386031f437
1.3	39f4324550365849852c5264b8d4535aae05e30d
1.4	f51f678630eb9b2fed355e5cedf976ce8b5fa341
1.5	07293ba09a69d3d6a16278220a0b4a7a9f318f96
1.6	f359dd049fb1ddde376e8ad8e5177c663e430418
1.7	aee9bfec868a70b3d9974d7692bc19f9739ab7dc
1.8	2f26292636c7248133e31ae912ee58113de2ef71
2.0	1e1a4d7f10df30d106b52eba1c5caf69e8bc0f36
2.1	8b9d84f25aedbd684eb0f06cdd4ffbbf9a60a0e2
2.2	5e0368f990d505d3bf1960122cb99a23e08567b0

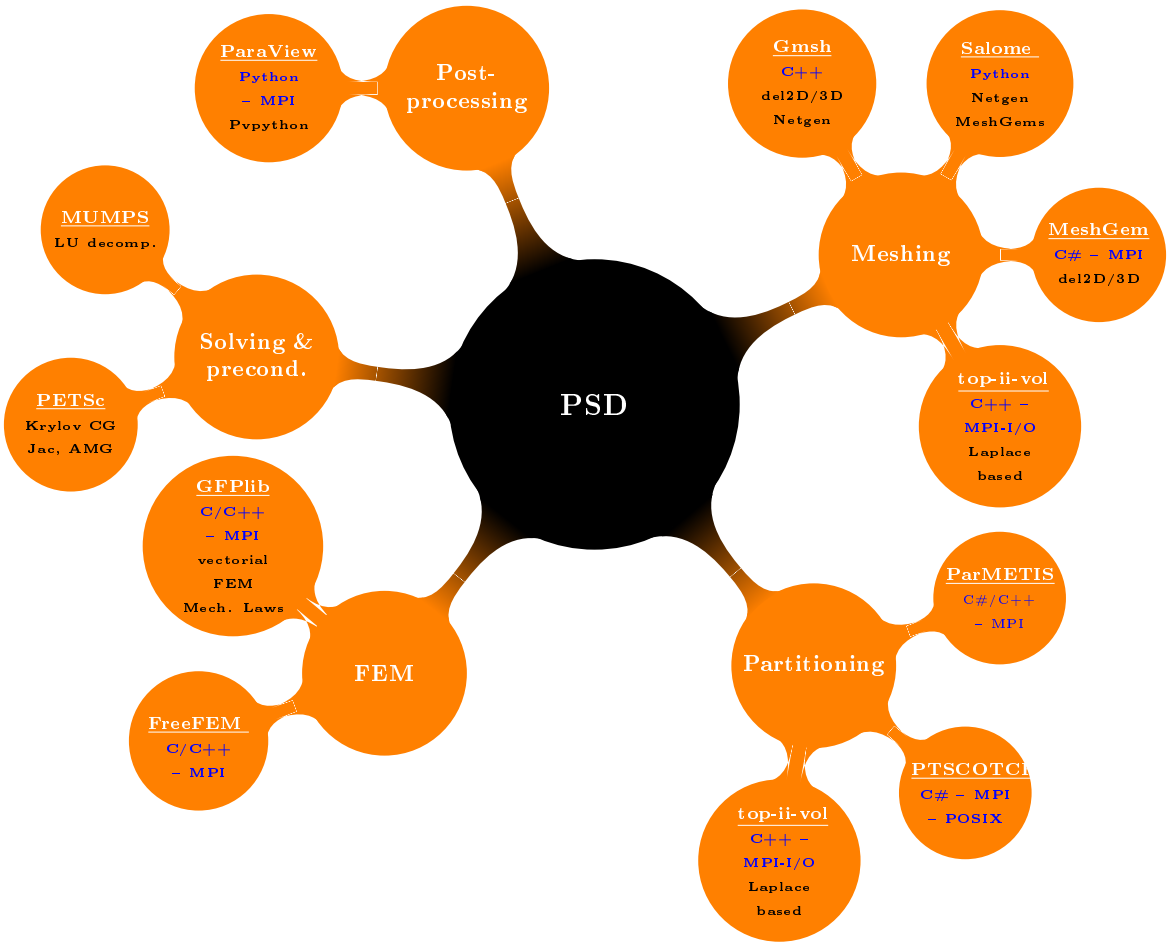


Figure 1.1: PSD app architecture

Chapter 2

Installation

2.1 Installation Procedure for PSD

PSD is a cross-platform solver built to work with Linux platforms (Windows comping up soon). PSD has successfully been deployed on the following platforms, CentOS 7/8, Ubuntu 16.04/18.04/20.04, Raspberry Pi, Fedora 32, etc. Before installing PSD please ensure that you have the following dependencies installed on your OS.

Dependencies

- automake
- FreeFEM
- PETSc
- Gmsh
- C++
- git
- MFront (optional)
- MGIS (optional)
- gnuplot (optional)

Now that I have all the dependencies what next

- Go ahead and grab the latest copy of PSD. The code is hosted on GitLab repository.

```
git clone https://gitlab.com/PsdSolver/psd_sources.git PSD-Sources
```

  Note: You can also use SSH protocol if your key has been added to the repo in that case use

```
git clone git@gitlab.com:PsdSolver/psd_sources.git
```

- Use automake PSD within the cloned folder

```
autoreconf -i
```

- Configure PSD within the cloned folder

```
./configure
```

  Note: `./configure` will install PSD in `/usr/local/bin` and you would need sudo rights to perform installation, for non sudo users or for local install consider changing directory of installation. To change this directory use `--prefix=Your/Own/Path` with `./configure`. Remember to add `Your/Own/Path` to your `$PATH` variable, you can do so by `export PATH=$PATH:Your/Own/Path`. Also `./configure` will try to look for installation of FreeFEM and Gmsh in `usr/bin/` or `usr/local/bin/` directories. If you have

these packages installed in some other directory this should be specified during `./configure` by using flags `--with-FreeFEM=` and `--with-Gmsh=`. For example, if FreeFEM is installed at `home/FreeFem/bin` and Gmsh in `home/Gmsh/bin` then one should use

```
./configure --with-FreeFEM=home/FreeFem/bin  --with-Gmsh=home/Gmsh/bin
```

- Make PSD directives

```
make
```

- Install PSD

```
sudo make install
```

Note : You should not use `sudo` if you have used `--prefix` during the `./configure`

- Install PSD tutorials

```
make tutorials
```

Now you should have the PSD solver installed on your machine. Note that, the solver will be installed at `usr/bin` or `usr/local/bin` directories if you used `sudo make install` or else it will be in your `--prefix` location. The PSD tutorials are installed in `$HOME/PSD-tutorials`.

Additional FreeFEM tweak for brittle fracture mechanics

Note that this procedure is only recommended if you are interested in using PSD for brittle fracture problems. In your FreeFEM source files (installation) go to `src/femlib/fem.cpp` , in this file replace the lines of code

```
R seuil=hm/splitmax/4.0;
```

by the following

```
R seuil=hm/splitmax/4.0/1000.0;
```

Additional Installation steps for experts and developers

- Check if installation is correct

```
make check
```

- Configure PSD with MFront support

```
./configure --prefix=$HOME/Install/local --with-mgis=$HOME/Install/local/mgis --with-mfront=$HOME/Install/local/mfront/bin/mfront  --w
```

Here in this example we assume that MFront, MGIS, Gmsh, and FreeFEM are installed in `$HOME/Install/local` directory, if that is not the case for you please improvise or you can always ask for help from Linux geeks around.

- Update PSD to the latest version. If you would like to update your old PSD source to a new one. Go to your`PSD-Sources` folder and

```
git pull origin master && ./reconfigure &&  make &&  make install
```

- If you would like to install a developers copy of PSD

```
make install-devel
```

2.1.1 A quick sneak-peek of a typical PSD simulation

PSD is a TUI (terminal user interface) based finite element solver. Parallel or sequential PSD simulations can run on Linux platforms. Command line options (flags) which user enters are used to control the PSD solver. In order to make your choice of physics, model, mesh, etc., command line options need to be typed right into the bash.

A typical PSD simulation is performed in three steps.

Step 1: Setting up the solver

Its time to set up the PSD solver. Open the `terminal` window at the location of the solver, i.e., `$HOME/PSD/Solver`. Then run the following command in the `terminal`.

```
PSD_PreProcess [Options-PSD]
```

Via the command line options you will embed the physics within the solver. This step generates a bunch of `.edp` files which are native to FreeFEM and additionally prints out instructions on what to do next. You then need to open and edit couple of these files via your favourite text editor, which could be `vim`, `gedit` ,`Notepad++`, etc. To facilitate the edit process for your will have to go through the instructions printed on the terminal.

For example to generate a sequential 2D elasticity solver for a problem with body force and one Dirichlet border use

```
PSD_PreProcess -dimension 2 -bodyforceconditions 1 -dirichletconditions 1
```

Step 2: Launching the solver

Now you are all set to run your simulation. To do so you will need to do the run the following in the `terminal`:

if you complied a parallel PSD version

```
PSD_Solve -np $N Main.edp -v 0 -nw
```

if you complied a sequential PSD version

```
PSD_Solve_Seq Main.edp -v 0 -nw
```

- In the parallel command **$N** is an `int` value, i.e., number of processes that you want to use for performing the simulation in parallel.
- Additional flag `-wg` may be required while launching the solver, this is in case debug mode is on.

Step 3: Result visualization Final step is to have a look at the results of the simulation. PSD can provides output results in the form of plots, finite element fields of interest, etc. ParaView's pvd, vtu, and pvtu files are used for postprocessing. ASCII data files that to trace certain quantities of interest like reaction forces, kinetic energies, etc can also be outputted.

2.1.2 PSD flags explained

These are a set of commandline flags/options that control your simulation. You can think of it as a way to talk to the solver. Here is a table that lists out some of the options that are available (for full list see documentation). It is advised to print these and have them around when performing a PSD simulation.

Flag	Type	Comment
Boolean flags		These flags accept values *1/0/yes/no/on/off/true/false* and are used to activate or deactivate any functionality of PSD.

Flag	Type	Comment
-help	[bool]	To activate helping message on the terminal. Gives description and list of available flags.
-debug	[bool]	To activate live plot while PSD runs. Development flag.
-useGFP	[bool]	To activate use of GoFastPlugins. A suite of C++ plugins.
-useRCM	[bool]	Activate mesh level renumbering: Reverse Cuthill Mckee.
-pipegnu	[bool]	Use to activate real time pipe plotting using gnuplot.
-timelog	[bool]	To activate time logging the different phases of the solver.
-supercomp	[bool]	Use when using a super computer without Xterm support
-bodyforce	[bool]	To activate volumetric source term (body force).
-vectorial	[bool]	To use vectorial finite element method.
-pointprobe	[bool]	To postprocess point fields.
-sequential	[bool]	To solve via a sequential solver.
-energydecomp	[bool]	To activate energy decomposition, only for phase-field.
-doublecouple	[bool]	To activate double couple source for soildynamics.
-constrainHPF	[bool]	To use constrain condition in hybrid phase-field model.
-top2vol-meshing	[bool]	Activate top-ii-vol point source meshing for soil-dynamics.
-getreactionforce	[bool]	Activate routine for extraction reactions at surface.
-plotreactionforce	[bool]	Activate realtime pipe plotting using GnuPlot.
-withmaterialtensor	[bool]	Activate material tensor for building stiffness matrix.
-crackdirichletcondition	[bool]	To activate pre-cracked surface Dirichlet.
Integer flags		These flags accept a integer value followed by the flag itself. These integer values are used in PSD simulations for various definitions.
-dirichletpointconditions	[int]	Number of Dirichlet points.
-dirichletconditions	[int]	Number of Dirichlet boundaries.
-bodyforceconditions	[int]	Number of regions acted upon by bodyforce.
-tractionconditions	[int]	Number of Neumann/traction boundaries.
-parmetis_worker	[int]	Number of parallel workers used by ParMetis for partitioning.
-lagrange	[int]	Lagrange order used for FE spaces. 1 for P1 or 2 for P2.
-dimension	[int]	Accepts values 2 or 3. Use 3 for 3D. and 2 for 2D problem.
String flags		These flags accept a string value followed by the flag itself. These string values are used in PSD simulations for various definitions.
-mesh	[sting]	Provide mesh to be solved by PSD_Solve.
-timediscretization	[sting]	Time discretization type. Use "generalized_alpha" or "newmark_beta" or "hht_alpha" or "central_difference"
-nonlinearmethod	[sting]	Nonlinear method type. Use "Picard" or "Newton_Raphsons".
-reactionforce	[sting]	Reaction force calculation method "stress_based" or "variational_based".
-doublecouple	[sting]	Soil dynamics double couple. Use "force_based" or "displacement_based".
-postprocess	[sting]	To communicate what to postprocess "u", "v", "a", "uv" , "ud", "ua", "d", "ud", or "uav".
-partitioner	[sting]	Mesh partitioner could be "metis" "parmetis" or "scotch".
-problem	[sting]	Interested problem. Use "linear_elasticity", "damage", "elastodynamics", or "soildynamics".

Flag	Type	Comment
`-model`	`[sting]`	Interested model. Use “hybrid_phase_field” or “Mazar”.

To report bugs, issues, feature-requests contact:

- **mohd-afeef.badri@cea.fr**
- **mohd-afeef.badri@hotmail.com**

Chapter 3

Theoretical background

3.1 Elastostatics

Let us consider d-dimensional domain $\Omega \in \mathbb{R}^d$ in a Euclidean referential $R(O, \mathbf{e}_i)$ (with $i = 1, \ldots, d$) submitted to a system of body forces $\mathbf{b}$. We denote $\partial\Omega$ the boundary of Ω and indicate with $\mathbf{n} = \mathbf{n}(\mathbf{x}) = n_i(\mathbf{x})\mathbf{e}_i$ its outer normal in any point $\mathbf{x} = x_i\mathbf{e}_i \in \partial\Omega$.

The problem to solve in order to characterize the dynamics equilibrium thus consists in finding a vector-valued displacement field $\mathbf{u} = \mathbf{u}(\mathbf{x}, t) : \Omega \times [0, T] \to \mathbb{R}^d$ regular "enough" and such that:

$$\begin{cases} \mathrm{div}\boldsymbol{\sigma} + \mathbf{b} = 0 & (\mathbf{x}, t) \in \Omega \times [0, T] \\ \boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{u}) & (\mathbf{x}, t) \in \Omega \times [0, T] \\ \mathbf{u} = \mathbf{u}^* & (\mathbf{x}, t) \in \partial_u\Omega \times [0, T] \\ \boldsymbol{\sigma} \cdot \mathbf{n} = \mathbf{t} & (\mathbf{x}, t) \in \partial_t\Omega \times [0, T] \end{cases} \tag{3.1}$$

where "div" denotes the divergence operator, symbol "$\cdot$" denotes the single contraction operation between tensors, $\rho = \rho(\mathbf{x}) : \Omega \to \mathbb{R}$ is the material density and $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{u})$ denotes a constitutive equation expressing the relationship between the second order Cauchy's stress tensor $\boldsymbol{\sigma} : \Omega \times [0, T] \to \mathbb{R}^{d\times d}$ and the displacement. Moreover, $\mathbf{u}^* = \mathbf{u}^*(\mathbf{x}, t) : \partial_u\Omega \times [0, T] \to \mathbb{R}^d$ is the imposed displacement field on $\partial_u\Omega$ (Dirichlet boundary condition) and $\mathbf{t} = \mathbf{t}(\mathbf{x}, t) : \partial_u\Omega \times [0, T] \to \mathbb{R}^d$ is the imposed traction vector on $\partial_t\Omega$ (Neumann boundary condition). The split of $\partial\Omega$ is such that $\partial\Omega = \overline{\partial_u\Omega \cup \partial_t\Omega}$ and $\partial_u\Omega \cap \partial_t\Omega = \emptyset$, with overline $\overline{\bullet}$ denoting the closure of set $\bullet$.

Let us now introduce the spaces of the admissible displacements fields ($\mathcal{U}$) and test functions ($\mathcal{V}$):

$$\begin{aligned} \mathcal{U} &= \left\{ \mathbf{u} = \mathbf{u}(\mathbf{x}, t) : \partial_u\Omega \times [0, T] \to \mathbb{R}^d \,|\, \mathbf{u} \in H^1(\Omega),\, \mathbf{u} = \mathbf{u}^*\, \partial_u\Omega \times [0, T] \right\} \\ \mathcal{V} &= \left\{ \mathbf{v} = \mathbf{v}(\mathbf{x}, t) : \partial_u\Omega \times [0, T] \to \mathbb{R}^d \,|\, \mathbf{v} \in H^1(\Omega),\, \mathbf{v} = 0\, \partial_u\Omega \times [0, T], \right\} \end{aligned} \tag{3.2}$$

The weak form of previous boundary value problem can be easily obtained by integrating by part the linear momentum balance equation using a test function $\mathbf{v} \in \mathcal{V}$, and imposing the Neumann boundary condition:

$$\underbrace{\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V}_{:=K(\mathbf{u},\mathbf{v})} = \underbrace{\int_\Omega \mathbf{b} \cdot \mathbf{v}\, \mathrm{d}V + \int_{\partial_t\Omega} \mathbf{t} \cdot \mathbf{v}\, \mathrm{d}S}_{:=b(\mathbf{v};\mathbf{b})+b(\mathbf{v};\mathbf{t})_{\partial_t\Omega}} \quad \forall\, \mathbf{v} \in \mathcal{V} \tag{3.3}$$

where symbol ":" is the double contraction operation between tensors, $K(\mathbf{u}, \mathbf{v})$ is the bi-linear symmetric form associated with the stiffness matrix and $b(\mathbf{v}; \mathbf{b}) + b(\mathbf{v}; \mathbf{t})_{\partial_t\Omega}$ are the linear forms associated with the external loading.[1]

[1] In the following of this document, given a known field $\mathbf{a}$, symbol $b(\mathbf{v}; \mathbf{a})$ will be used to denote the linear form $\int_\Omega \mathbf{a} \cdot \mathbf{v}\, \mathrm{d}V$, whereas $b(\mathbf{v}; \mathbf{a})_{Surf}$ will denote the linear form obtained form the surface integral $\int_{Surf} \mathbf{a} \cdot \mathbf{v}\, \mathrm{d}S$. Any linear form without down-script has to be interpreted as an integral over Ω. Only surface integrals will be defined explicitly.

The problem to solve can be finally written as:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}\in\mathcal{U} \text{ such that :}\\ K(\mathbf{u},\mathbf{v}) = b(\mathbf{v};\mathbf{b}) + b(\mathbf{v};\mathbf{t})_{\partial_t\Omega} \quad \forall\,\mathbf{v}\in\mathcal{V}\end{array}} \tag{3.4}$$

3.2 Elastodynamics

The problem to solve in order to characterize the dynamics equilibrium thus consists in finding a vector-valued displacement field $\mathbf{u} = \mathbf{u}(\mathbf{x},t) : \Omega\times[0,T]\to\mathbb{R}^d$ regular "enough" and such that:

$$\begin{cases}\mathrm{div}\boldsymbol{\sigma} + \mathbf{b} = \rho\ddot{\mathbf{u}} & (\mathbf{x},t)\in\Omega\times[0,T]\\ \boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{u}) & (\mathbf{x},t)\in\Omega\times[0,T]\\ \mathbf{u} = \mathbf{u}^* & (\mathbf{x},t)\in\partial_u\Omega\times[0,T]\\ \boldsymbol{\sigma}\cdot\mathbf{n} = \mathbf{t} & (\mathbf{x},t)\in\partial_t\Omega\times[0,T]\\ \mathbf{u} = \mathbf{u}_0 & \mathbf{x}\in\Omega,\ t=0\\ \dot{\mathbf{u}} = \dot{\mathbf{u}}_0 & \mathbf{x}\in\Omega,\ t=0\end{cases} \tag{3.5}$$

where "div" denotes the divergence operator, symbol "$\cdot$" denotes the single contraction operation between tensors, $\rho = \rho(\mathbf{x}) : \Omega\to\mathbb{R}$ is the material density, $\ddot{\mathbf{u}} = \ddot{\mathbf{u}}(\mathbf{x},t) = \mathbf{u}_{tt} : \Omega\times[0,T]\to\mathbb{R}^d$ is the acceleration field (i.e., the second time derivative of the field $\mathbf{u}$) and $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{u})$ denotes a constitutive equation expressing the relationship between the second order Cauchy's stress tensor $\boldsymbol{\sigma} : \Omega\times[0,T]\to\mathbb{R}^{d\times d}$ and the displacement. Moreover, $\mathbf{u}^* = \mathbf{u}^*(\mathbf{x},t) : \partial_u\Omega\times[0,T]\to\mathbb{R}^d$ is the imposed displacement field on $\partial_u\Omega$ (Dirichlet boundary condition) and $\mathbf{t} = \mathbf{t}(\mathbf{x},t) : \partial_u\Omega\times[0,T]\to\mathbb{R}^d$ is the imposed traction vector on $\partial_t\Omega$ (Neumann boundary condition). The split of $\partial\Omega$ is such that $\partial\Omega = \overline{\partial_u\Omega\cup\partial_t\Omega}$ and $\partial_u\Omega\cap\partial_t\Omega = \emptyset$, with overline $\overline{\bullet}$ denoting the closure of set $\bullet$. Finally, $\mathbf{u}_0 = \mathbf{u}_0(\mathbf{x},0) : \Omega\to\mathbb{R}^d$ and $\dot{\mathbf{u}}_0 = \dot{\mathbf{u}}_0(\mathbf{x},0) : \Omega\to\mathbb{R}^d$ are the displacement and velocity fields at time $t=0$ (initial conditions).

Let us now introduce the spaces of the admissible displacements fields ($\mathcal{U}$) and test functions ($\mathcal{V}$):

$$\begin{aligned}\mathcal{U} &= \left\{\mathbf{u} = \mathbf{u}(\mathbf{x},t) : \partial_u\Omega\times[0,T]\to\mathbb{R}^d \,|\, \mathbf{u}\in H^1(\Omega),\ \mathbf{u} = \mathbf{u}^*\,\partial_u\Omega\times[0,T],\ \mathbf{u}(\mathbf{x},0) = 0,\ \mathbf{u}(\dot{\mathbf{x}},0) = \dot{\mathbf{u}}_0\right\}\\ \mathcal{V} &= \left\{\mathbf{v} = \mathbf{v}(\mathbf{x},t) : \partial_u\Omega\times[0,T]\to\mathbb{R}^d \,|\, \mathbf{v}\in H^1(\Omega),\ \mathbf{v} = 0\,\partial_u\Omega\times[0,T],\right\}\end{aligned} \tag{3.6}$$

The weak form of previous boundary value problem can be easily obtained by integrating by part the linear momentum balance equation using a test function $\mathbf{v}\in\mathcal{V}$, and imposing the Neumann boundary condition:

$$\underbrace{\int_\Omega \rho\ddot{\mathbf{u}}\cdot\mathbf{v}\,\mathrm{d}V}_{:=M(\ddot{\mathbf{u}},\mathbf{v})} + \underbrace{\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\epsilon}(\mathbf{v})\,\mathrm{d}V}_{:=K(\mathbf{u},\mathbf{v})} = \underbrace{\int_\Omega \mathbf{b}\cdot\mathbf{v}\,\mathrm{d}V + \int_{\partial_t\Omega}\mathbf{t}\cdot\mathbf{v}\,\mathrm{d}S}_{:=b(\mathbf{v};\mathbf{b})+b(\mathbf{v};\mathbf{t})_{\partial_t\Omega}} \quad \forall\,\mathbf{v}\in\mathcal{V} \tag{3.7}$$

where symbol ":" is the double contraction operation between tensors, $M(\ddot{\mathbf{u}},\mathbf{v})$ is the bi-linear symmetric form associated with inertial terms (i.e., dependent on the mass matrix), $K(\mathbf{u},\mathbf{v})$ is the bi-linear symmetric form associated with the stiffness matrix and $b(\mathbf{v};\mathbf{b}) + b(\mathbf{v};\mathbf{t})_{\partial_t\Omega}$ are the linear forms associated with the external loading.[2]

The problem to solve can be finally written as:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}\in\mathcal{U} \text{ such that :}\\ M(\ddot{\mathbf{u}},\mathbf{v}) + K(\mathbf{u},\mathbf{v}) = b(\mathbf{v};\mathbf{b}) + b(\mathbf{v};\mathbf{t})_{\partial_t\Omega} \quad \forall\,\mathbf{v}\in\mathcal{V}\end{array}} \tag{3.8}$$

[2] In the following of this document, given a known field $\mathbf{a}$, symbol $b(\mathbf{v};\mathbf{a})$ will be used to denote the linear form $\int_\Omega \mathbf{a}\cdot\mathbf{v}\,\mathrm{d}V$, whereas $b(\mathbf{v};\mathbf{a})_{Surf}$ will denote the linear form obtained form the surface integral $\int_{Surf}\mathbf{a}\cdot\mathbf{v}\,\mathrm{d}S$. Any linear form without down-script has to be interpreted as an integral over Ω. Only surface integrals will be defined explicitly.

The only way for accounting form dumping effects in this formulation is through a proper definition of a constitutive law $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{u})$ modeling dissipative processes occurring at the material level. In some cases, however, it can be useful to account for damping effects in a more global way. This can be done by modifying the variational problem as follows:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}\in\mathcal{U} \text{ such that :}\\ M(\ddot{\mathbf{u}},\mathbf{v}) + C(\dot{\mathbf{u}},\mathbf{v}) + K(\mathbf{u},\mathbf{v}) = b(\mathbf{v};\mathbf{b}) + b(\mathbf{v};\mathbf{t})_{\partial_t\Omega} \quad \forall\,\mathbf{v}\in\mathcal{V}\end{array}} \tag{3.9}$$

where $C(\dot{\mathbf{u}},\mathbf{v})$ is an additional bi-linear symmetric form associated with damping/viscous effects.

3.3 Time discretization

Time discretized variational formulations are illustrate in this subsection, considering several implicit time integration schemes. Representative members of these algorithms are, among others, the N$-\beta$ method [**newmark1959method**], the HHT$-\alpha$ method [**hilber1977improved**], the WBZ$-\alpha$ method [**wood1980alpha**], the HP$-\theta_1$ method [**hoff1988development**] and the CH$-\alpha$ method [**chung1993time**]. These methods exhibit second order accuracy in linear dynamics and permit efficient variable step size techniques, being one-step methods. The CH$-\alpha$, the HHT$-\alpha$ and the WBZ$-\alpha$ methods, the so called $\alpha-$methods, are one-parameter schemes which can be considered as particular cases of a more general class of methods named generalized $-\ \alpha$ (G $-\ \alpha$). This class of methods corresponds to the CH$-\alpha$ scheme [**chung1993time**], where the algorithmic parameters α_m, α_f, β and γ are assumed to be independent of each other.

3.3.1 Generalized-α method

The Generalized $-\ \alpha$ (G $-\ \alpha$) is an implicit method that allows for high frequency energy dissipation, reduced unwanted low-frequency dissipation, and second order accuracy (i.e., Δt^2), both in linear and nonlinear regimes. Depending on choices of input parameters, unconditionally stability can be achieved for linear problems (as for all implicit schemes). Stability properties for nonlinear problem were studied in [**erlicher2002analysis**]. In the latter work, the second-order accuracy of this class of algorithms was proved also in the non-linear regime, independently of the quadrature rule for non-linear internal forces. Conversely, the G-stability notion which is suitable for linear multi-step schemes devoted to non-linear dynamic problems cannot be applied, as the non-linear structural dynamics equations are not contractive. Nonetheless, [**erlicher2002analysis**] proved that the G $-\ \alpha$ methods are stable in an energy sense, and guarantee energy decay for high-frequencies and asymptotic cancellation. However, overshoot and heavy energy oscillations in the intermediate-frequency range are exhibited.

Problem setting

Let introduce a time discretization of the time interval $[0,T]$ in an ordered sequence of $N+1$ time increments $(0,\dots,t_i,t_{i+1},\dots,T)$ such that $t_{i+1} = t_i + \Delta t$, with $\Delta t = T/N$ denoting the time step (here supposed constant). According to the (G $-\ \alpha$) method, the dynamic evolution equation is solved at intermediate time $t_{n+1-\alpha} \in [t_n, t_{n+1}]$. The following notation is used to denote the value of a generic variable z at time $t_{n+1-\alpha}$:

$$z_{n+1-\alpha} = (1-\alpha) z_{n+1} + \alpha z_n \quad \text{with} \quad \alpha \in [0,1] \tag{3.10}$$

Furthermore, the following approximations (standard for Newmark schemes) for the displacement and velocity fields at time t_{n+1} are used [**newmark1959method**]:

$$\begin{aligned}\mathbf{u}_{n+1} &= \overline{\mathbf{u}}_{n+1} + \beta\,\Delta t^2\,\ddot{\mathbf{u}}_{n+1}\\ \dot{\mathbf{u}}_{n+1} &= \dot{\overline{\mathbf{u}}}_{n+1} + \gamma\,\Delta t\,\ddot{\mathbf{u}}_{n+1}\end{aligned} \tag{3.11}$$

where $\overline{\mathbf{u}}_{n+1}$ and $\dot{\overline{\mathbf{u}}}_{n+1}$ are the following known contributions (predictions in predictor-corrector schemes):

$$\begin{aligned}\overline{\mathbf{u}}_{n+1} &= \mathbf{u}_n + \Delta t\,\dot{\mathbf{u}}_n + \Delta t^2\left(\frac{1}{2}-\beta\right)\ddot{\mathbf{u}}_n \\ \dot{\overline{\mathbf{u}}}_{n+1} &= \dot{\mathbf{u}}_n + \Delta t\,(1-\gamma)\ddot{\mathbf{u}}_n\end{aligned} \tag{3.12}$$

and (β,γ) are algorithmic parameters. By inverting the first equation of (3.11), one can express $\ddot{\mathbf{u}}_{n+1}$ as a function of $\mathbf{u}_{n+1}$ as:

$$\ddot{\mathbf{u}}_{n+1} = \frac{1}{\beta\,\Delta t^2}\left(\mathbf{u}_{n+1} - \overline{\mathbf{u}}_{n+1}\right) \tag{3.13}$$

3.3.2 Time discretized variational problem (no damping)

Neglecting damping effects, the problem to solve is written as:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}_{n+1} \in \mathcal{U} \text{ such that :} \\ M(\ddot{\mathbf{u}}_{n+1-\alpha_m},\mathbf{v}) + K(\mathbf{u}_{n+1-\alpha_f},\mathbf{v}) = b(\mathbf{v};\mathbf{b}) + b(\mathbf{v};\mathbf{t}_{n+1-\alpha_f})_{\partial_t\Omega} \quad \forall\,\mathbf{v}\in\mathcal{V}\end{array}} \tag{3.14}$$

where $\ddot{\mathbf{u}}_{n+1-\alpha_m}$ and $\mathbf{u}_{n+1-\alpha_f}$ can be written according to (3.10):

$$\begin{aligned}\ddot{\mathbf{u}}_{n+1-\alpha_m} &= \frac{1-\alpha_m}{\beta\,\Delta t^2}\left(\mathbf{u}_{n+1} - \overline{\mathbf{u}}_{n+1}\right) + \alpha_m\ddot{\mathbf{u}}_n \\ \mathbf{u}_{n+1-\alpha_f} &= (1-\alpha_f)\mathbf{u}_{n+1} + \alpha_f\mathbf{u}_n\end{aligned} \tag{3.15}$$

Furthermore, parameters β and γ read:

$$\gamma = \frac{1}{2} + \alpha_f - \alpha_m \qquad \beta = \frac{1}{4}\left(\gamma + \frac{1}{2}\right)^2 \tag{3.16}$$

Bilinear and linear operators. Using equation (3.13), one can easily write the bi-linear part associated with the mass matrix in terms of the unknown displacement $\mathbf{u}_{n+1}$ as follows:

$$M(\ddot{\mathbf{u}}_{n+1-\alpha_m},\mathbf{v}) = \frac{1-\alpha_m}{\beta\,\Delta t^2}M(\mathbf{u}_{n+1},\mathbf{v}) - \frac{1-\alpha_m}{\beta\,\Delta t^2}m(\mathbf{v};\overline{\mathbf{u}}_{n+1}) + \alpha_m m(\mathbf{v};\ddot{\mathbf{u}}_n) \tag{3.17}$$

where linear forms $m(\mathbf{v};\overline{\mathbf{u}}_{n+1})$ and $m(\mathbf{v};\ddot{\mathbf{u}}_n)$ read:[3]

$$m(\mathbf{v};\overline{\mathbf{u}}_{n+1}) = \int_\Omega \rho\overline{\mathbf{u}}_{n+1}\cdot\mathbf{v}\,\mathrm{d}V \qquad m(\mathbf{v};\ddot{\mathbf{u}}_n) = \int_\Omega \rho\ddot{\mathbf{u}}_n\cdot\mathbf{v}\,\mathrm{d}V \tag{3.19}$$

Term $m(\mathbf{v};\overline{\mathbf{u}}_{n+1})$ figuring in equation (3.17) can also be expanded as:

$$m(\mathbf{v};\overline{\mathbf{u}}_{n+1}) = m(\mathbf{v};\mathbf{u}_n) + \Delta t\,m(\mathbf{v};\dot{\mathbf{u}}_n) + \Delta t^2\left(\frac{1}{2}-\beta\right)m(\mathbf{v};\ddot{\mathbf{u}}_n) \tag{3.20}$$

As a consequence (3.17) can be rewritten as:[4]

$$M(\ddot{\mathbf{u}}_{n+1-\alpha_m},\mathbf{v}) = \frac{1-\alpha_m}{\beta\,\Delta t^2}M(\mathbf{u}_{n+1},\mathbf{v}) - \frac{1-\alpha_m}{\beta\,\Delta t^2}m(\mathbf{v};\mathbf{u}_n) - \frac{1-\alpha_m}{\beta\,\Delta t}m(\mathbf{v};\dot{\mathbf{u}}_n) + \left(1-\frac{1-\alpha_m}{2\beta}\right)m(\mathbf{v};\ddot{\mathbf{u}}_n) \tag{3.22}$$

[3] More in general, given a field $\mathbf{a}=\mathbf{a}(\mathbf{x}):\Omega\to\mathbb{R}^d$, $m(\mathbf{v};\mathbf{a})$ denotes the linear form:

$$m(\mathbf{v};\mathbf{a}) = \int_\Omega \rho\mathbf{a}\cdot\mathbf{v}\,\mathrm{d}V \tag{3.18}$$

[4] When summing up the terms depending on $\ddot{\mathbf{u}}_n$, coming from the definition of $\overline{\mathbf{u}}_{n+1}$ and from equation (3.17), we have:

$$-\left[(1-\alpha_m)\left(\frac{1-2\beta}{2\beta}\right)-\alpha_m\right] = -\frac{(1-\alpha_m)(1-2\beta)-2\beta\alpha_m}{2\beta} = -\frac{1-2\beta-\alpha_m+2\beta\alpha_m-2\beta\alpha_m}{2\beta} = 1-\frac{1-\alpha_m}{2\beta} \tag{3.21}$$

In a similar way, we can rewrite the bi-linear form associated with the stiffness matrix as:

$$K(\mathbf{u}_{n+1-\alpha_f}, \mathbf{v}) = (1-\alpha_f)K(\mathbf{u}_{n+1}, \mathbf{v}) + \alpha_f k(\mathbf{v}; \mathbf{u}_n) \tag{3.23}$$

where $k(\mathbf{v}; \mathbf{u}_n)$ is the linear form:[5]

$$k(\mathbf{v}; \mathbf{u}_n) = \int_\Omega \boldsymbol{\sigma}(\mathbf{u}_n) : \boldsymbol{\epsilon}(\mathbf{v}) \, \mathrm{d}V \tag{3.25}$$

Finally, the linear form $b(\mathbf{v}; \mathbf{t}_{n+1-\alpha_f})_{\partial_t\Omega}$ becomes:

$$b(\mathbf{v}; \mathbf{t}_{n+1-\alpha_f})_{\partial_t\Omega} = (1-\alpha_f) b(\mathbf{v}; \mathbf{t}_{n+1})_{\partial_t\Omega} + \alpha_f b(\mathbf{v}; \mathbf{t}_n)_{\partial_t\Omega} \tag{3.26}$$

Final variational problem. The time discretized variational formulation to solve becomes:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}_{n+1} \in \mathcal{U} \text{ such that :} \\ \widetilde{K}(\mathbf{u}_{n+1}, \mathbf{v}) = \widetilde{l}(\mathbf{v})\end{array}} \tag{3.27}$$

where $\widetilde{K}(\mathbf{u}_{n+1}, \mathbf{v})$ is the bi-linear form associated with the effective/equivalent stiffness matrix:

$$\widetilde{K}(\mathbf{u}_{n+1}, \mathbf{v}) = \frac{1-\alpha_m}{\beta \, \Delta t^2} M(\mathbf{u}_{n+1}, \mathbf{v}) + (1-\alpha_f) K(\mathbf{u}_{n+1}, \mathbf{v}) \tag{3.28}$$

and $\widetilde{l}(\mathbf{v}) = \widetilde{l}(\mathbf{v}; \{\mathbf{b}, \mathbf{t}_n, \mathbf{t}_{n+1}, \mathbf{u}_n, \dot{\mathbf{u}}_n, \ddot{\mathbf{u}}_n\})$ is the following linear form:

$$\begin{aligned}\widetilde{l}(\mathbf{v}) = b(\mathbf{v}; \mathbf{b}) + b(\mathbf{v}; \mathbf{t}_{n+1-\alpha_f})_{\partial_t\Omega} + \frac{1-\alpha_m}{\beta \, \Delta t^2} m(\mathbf{v}, \mathbf{u}_n) + \frac{1-\alpha_m}{\beta \, \Delta t} m(\mathbf{v}; \dot{\mathbf{u}}_n) \\ \cdots + \left(1 - \frac{1-\alpha_m}{2\beta}\right) m(\mathbf{v}; \ddot{\mathbf{u}}_n) - \alpha_f k(\mathbf{v}; \mathbf{u}_n)\end{aligned} \tag{3.29}$$

3.3.3 Time discretized variational problem (Rayleigh damping)

The problem to solve is now:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}_{n+1} \in \mathcal{U} \text{ such that :} \\ M(\ddot{\mathbf{u}}_{n+1-\alpha_m}, \mathbf{v}) + C(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v}) + K(\mathbf{u}_{n+1-\alpha_f}, \mathbf{v}) = b(\mathbf{v}; \mathbf{b}) + b(\mathbf{v}; \mathbf{t}_{n+1-\alpha_f})_{\partial\Omega} \quad \forall \mathbf{v} \in \mathcal{V}\end{array}} \tag{3.30}$$

where, following a simple Rayleigh formulation, the bi-linear form associated with the damping matrix can be written as:

$$C(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v}) = \eta_M M(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v}) + \eta_K K(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v}) \tag{3.31}$$

with (η_M, η_K) denoting two positive model parameters.

Now, using definitions (3.10), (3.11) and (3.12), $\dot{\mathbf{u}}_{n+1-\alpha_f}$ can be written as:[6]

$$\dot{\mathbf{u}}_{n+1-\alpha_f} = \frac{\gamma(1-\alpha_f)}{\beta \, \Delta t} \mathbf{u}_{n+1} + (1-\alpha_f)\dot{\overline{\mathbf{u}}}_{n+1} - \frac{\gamma(1-\alpha_f)}{\beta \, \Delta t} \overline{\mathbf{u}}_{n+1} + \alpha_f \dot{\mathbf{u}}_n \tag{3.33}$$

[5] More in general, given a field $\mathbf{a} = \mathbf{a}(\mathbf{x}) : \Omega \to \mathbb{R}^d$, $k(\mathbf{v}; \mathbf{a})$ denotes the linear form:

$$k(\mathbf{v}; \mathbf{a}) = \int_\Omega \boldsymbol{\sigma}(\mathbf{a}) : \boldsymbol{\epsilon}(\mathbf{v}) \, \mathrm{d}V \tag{3.24}$$

[6] Using definitions (3.10), (3.11) and (3.12), the velocity field at time $t_{n+1-\alpha_f}$ reads:

$$\begin{aligned}\dot{\mathbf{u}}_{n+1-\alpha_f} &= (1-\alpha_f)\dot{\mathbf{u}}_{n+1} + \alpha_f \dot{\mathbf{u}}_n \\ &= (1-\alpha_f)\dot{\overline{\mathbf{u}}}_{n+1} + \alpha_f \dot{\mathbf{u}}_n + \gamma \, \Delta t \, (1-\alpha_f)\ddot{\mathbf{u}}_{n+1} \\ &= \frac{\gamma(1-\alpha_f)}{\beta \, \Delta t} \mathbf{u}_{n+1} + (1-\alpha_f)\dot{\overline{\mathbf{u}}}_{n+1} - \frac{\gamma(1-\alpha_f)}{\beta \, \Delta t} \overline{\mathbf{u}}_{n+1} + \alpha_f \dot{\mathbf{u}}_n\end{aligned} \tag{3.32}$$

or, using definitions (3.12), as:[7]

$$\dot{\mathbf{u}}_{n+1-\alpha_f} = \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_{n+1} - \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_n - \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right]\dot{\mathbf{u}}_n - \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right)\ddot{\mathbf{u}}_n \tag{3.35}$$

Bilinear and linear operators. Operator $M(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v})$ reads:

$$\begin{aligned} M(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v}) = \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}M(\mathbf{u}_{n+1}, \mathbf{v}) - \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t}m(\mathbf{v}; \mathbf{u}_n) \\ \cdots - \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right] m(\mathbf{v}; \dot{\mathbf{u}}_n) - \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right) m(\mathbf{v}; \ddot{\mathbf{u}}_n) \end{aligned} \tag{3.36}$$

Similarly, the stiffness contribution becomes:

$$\begin{aligned} K(\dot{\mathbf{u}}_{n+1-\alpha_f}, \mathbf{v}) = \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}K(\mathbf{u}_{n+1}, \mathbf{v}) - \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}k(\mathbf{v}; \mathbf{u}_n) \\ \cdots - \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right] k(\mathbf{v}; \dot{\mathbf{u}}_n) - \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right) k(\mathbf{v}; \ddot{\mathbf{u}}_n) \end{aligned} \tag{3.37}$$

Final variational problem. Finally, the variational problem to solve reads:

$$\boxed{\begin{array}{l} \text{Find } \mathbf{u}_{n+1} \in \mathcal{U} \text{ such that :} \\ \widetilde{\widetilde{K}}(\mathbf{u}_{n+1}, \mathbf{v}) = \widetilde{\widetilde{l}}(\mathbf{v}) \quad \forall\, \mathbf{v} \in \mathcal{V} \end{array}} \tag{3.38}$$

where $\widetilde{\widetilde{K}}(\mathbf{u}_{n+1}, \mathbf{v})$ is the bi-linear form associated with the effective stiffness matrix:

$$\begin{aligned} \widetilde{\widetilde{K}}(\mathbf{u}_{n+1}, \mathbf{v}) &= \widetilde{K}(\mathbf{u}_{n+1}, \mathbf{v}) + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}C(\mathbf{u}_{n+1}, \mathbf{v}) \\ &= \frac{1-\alpha_m}{\beta\Delta t^2}M(\mathbf{u}_{n+1}, \mathbf{v}) + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}C(\mathbf{u}_{n+1}, \mathbf{v}) + (1-\alpha_f)\,K(\mathbf{u}_{n+1}, \mathbf{v}) \end{aligned} \tag{3.39}$$

with $C(\mathbf{u}_{n+1}, \mathbf{v})$ denoting the Rayleigh damping operator:

$$C(\mathbf{u}_{n+1}, \mathbf{v}) = \eta_M M(\mathbf{u}_{n+1}, \mathbf{v}) + \eta_K K(\mathbf{u}_{n+1}, \mathbf{v}) \tag{3.40}$$

and $\widetilde{\widetilde{l}}(\mathbf{v}) = \widetilde{\widetilde{l}}(\mathbf{v}; \{\mathbf{b}, \mathbf{t}_n, \mathbf{t}_{n+1}, \mathbf{u}_n, \dot{\mathbf{u}}_n, \ddot{\mathbf{u}}_n\})$ being the following linear form:

$$\begin{aligned} \widetilde{\widetilde{l}}(\mathbf{v}) = \widetilde{l}(\mathbf{v}) + \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right) c(\mathbf{v}; \ddot{\mathbf{u}}_n) \\ \cdots + \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right] c(\mathbf{v}; \dot{\mathbf{u}}_n) + \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t}c(\mathbf{v}; \mathbf{u}_n) \end{aligned} \tag{3.41}$$

In previous equation we introduced the following notation:

$$c(\mathbf{v}; \mathbf{a}) = \eta_M m(\mathbf{v}; \mathbf{a}) + \eta_K k(\mathbf{v}; \mathbf{a}) \tag{3.42}$$

[7] Using definitions (3.12) one obtains:

$$\begin{aligned} \dot{\mathbf{u}}_{n+1-\alpha_f} &= \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_{n+1} + (1-\alpha_f)\left[\dot{\mathbf{u}}_n + \Delta t\,(1-\gamma)\ddot{\mathbf{u}}_n\right] - \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t}\left[\mathbf{u}_n + \Delta t\,\dot{\mathbf{u}}_n + \Delta t^2\left(\frac{1}{2} - \beta\right)\ddot{\mathbf{u}}_n\right] + \alpha_f\dot{\mathbf{u}}_n \\ &= \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_{n+1} + \left[1 - \frac{\gamma(1-\alpha_f)}{\beta}\right]\dot{\mathbf{u}}_n + (1-\alpha_f)\Delta t\left\{1 - \gamma\left[1 + \left(\frac{1-2\beta}{2\beta}\right)\right]\right\}\ddot{\mathbf{u}}_n - \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_n \\ &= \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_{n+1} - \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t}\mathbf{u}_n - \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right]\dot{\mathbf{u}}_n - \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right)\ddot{\mathbf{u}}_n \end{aligned} \tag{3.34}$$

3.3.4 Implicit N—β and HHT—α method as special cases

Newmark. One can easily show that, the Newmark scheme is obtained by choosing $\alpha_m = \alpha_f = 0$. Without damping, the stiffness matrix becomes:

$$\widetilde{K}(\mathbf{u}_{n+1}, \mathbf{v}) = \frac{1}{\beta \Delta t^2} M(\mathbf{u}_{n+1}, \mathbf{v}) + K(\mathbf{u}_{n+1}, \mathbf{v}) \tag{3.43}$$

whereas the linear form simplifies as follows:

$$\begin{aligned} \widetilde{l}(\mathbf{v}) &= b(\mathbf{v}; \mathbf{b}) + b(\mathbf{v}; \mathbf{t}_{n+1})_{\partial_t \Omega} + \frac{1}{\beta\, \Delta t^2} m(\mathbf{v}; \overline{\mathbf{u}}_{n+1}) \\ &= b(\mathbf{v}; \mathbf{b}) + b(\mathbf{v}; \mathbf{t}_{n+1})_{\partial_t \Omega} + \frac{1}{\beta\, \Delta t^2} \left[m(\mathbf{v}; \mathbf{u}_n) + \Delta t\, m(\mathbf{v}; \dot{\mathbf{u}}_n) + \Delta t^2 \left(\frac{1}{2} - \beta \right) m(\mathbf{v}; \ddot{\mathbf{u}}_n) \right] \end{aligned} \tag{3.44}$$

When Rayleigh damping is considered, the bi-linear operator $\widetilde{\widetilde{K}}(\mathbf{u}_{n+1}, \mathbf{v})$ becomes:

$$\widetilde{\widetilde{K}}(\mathbf{u}_{n+1}, \mathbf{v}) = \frac{1}{\beta \Delta t^2} M(\mathbf{u}_{n+1}, \mathbf{v}) + \frac{\gamma}{\beta\, \Delta t} C(\mathbf{u}_{n+1}, \mathbf{v}) + K(\mathbf{u}_{n+1}, \mathbf{v}) \tag{3.45}$$

whereas the linear form simplifies as follows:

$$\widetilde{\widetilde{l}}(\mathbf{v}) = \widetilde{l}(\mathbf{v}) + \Delta t \left(\frac{\gamma}{2\beta} - 1 \right) c(\mathbf{v}; \ddot{\mathbf{u}}_n) + \left(\frac{\gamma}{\beta} - 1 \right) c(\mathbf{v}; \dot{\mathbf{u}}_n) + \frac{\gamma}{\beta\, \Delta t} c(\mathbf{v}; \mathbf{u}_n) \tag{3.46}$$

HHT. One can also show that HHT—α [**hilber1977improved**] method is recovered for $\alpha_m = 0$. Such formulation is not detailed in the following of this document, since it is less used than the classic Newmark approach.

3.3.5 Considerations on methods based upon operator splitting

In order to introduce predictor-correction, implicit-explicit and more in general schemes based upon operator splitting, one can rewrite displacement and velocity in a predictor-correction fashion as in (3.11) and (3.12), where (3.12) now defines predictors and (3.11) correctors (for more general information, the interested reader can refer to [**hughes2012finite**, **zienkiewicz1994finite**]). For instance, a simple explicit predictor-corrector method can be obtained through solving problem (3.30) with $K(\bar{\mathbf{u}}_{n+1}, \mathbf{v})$ and $C((1-\alpha_f)\dot{\bar{\mathbf{u}}}_{n+1} + \alpha_f \mathbf{u}_n)$. Mixed implicit-explicit predictor-corrector methods can also be obtained through splitting Ω into two subdomains and using different time-integration schemes for solving the dynamic equilibrium problem on each of them.

3.4 Space discretization

Space discretization is performed according to the standard finite element method. The computational domain Ω is thus discretized into a mesh Ω^h comprising a finite number (n_{el}) of subdomains, the finite element Ω_e^h, such that $\Omega \approx \Omega^h = \cup_{e=1}^{n_{el}} \Omega_e^h$. Inside each element, the displacement field is interpolated based on nodal displacements ($\mathbf{d}$) through the shape functions matrix ($\mathbf{N}$), i.e., $\mathbf{u} \approx \mathbf{u}^h = \mathbf{N}(\mathbf{x})\mathbf{d}$. As usual, gradient terms are interpolated using the derivatives of the shape functions, i.e., $\boldsymbol{\epsilon} \approx \boldsymbol{\epsilon}^h = \mathbf{B}(\mathbf{x})\mathbf{d}$.

In a standard matrix format, after spatial discretization of the displacement field, the problem to solve can

be written in the standard form as:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{d}_{n+1} \text{ such that :} \\ \left[\frac{1-\alpha_m}{\beta \Delta t^2}\mathbf{M} + (1-\alpha_f)\,\mathbf{K}\right]\mathbf{d}_{n+1} \\ = \mathbf{f}_{n+1-\alpha_f} + \mathbf{M}\left[\left(\frac{1-\alpha_m}{2\beta}-1\right)\ddot{\mathbf{d}}_n + \frac{1-\alpha_m}{\beta\,\Delta t}\dot{\mathbf{d}}_n + \frac{1-\alpha_m}{\beta\,\Delta t^2}\mathbf{d}_n\right] - \alpha_f \mathbf{K}\mathbf{d}_n\end{array}} \tag{3.47}$$

where $\mathbf{M}$ and $\mathbf{K}$ are now the mass and stiffness matrices. They are obtained through assembling (operator A) the corresponding elemental operators over the finite element mesh as:

$$\begin{aligned}\mathbf{M} &= \mathop{\mathsf{A}}_{e=1}^{n_{el}} \int_{\Omega_e} \rho \mathbf{N}^\top \mathbf{N} \mathrm{d}V \\ \mathbf{K} &= \mathop{\mathsf{A}}_{e=1}^{n_{el}} \int_{\Omega_e} \mathbf{B}^\top \mathbf{D} \mathbf{B} \mathrm{d}V\end{aligned} \tag{3.48}$$

where $\mathbf{D}$ is the material stiffness matrix defining the link between the stress and strain tensors (or between theirs rates of variation – more details are given in the next section).

When Rayleigh damping is considered the problem to solve is written as:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{d}_{n+1} \text{ such that :} \\ \left[\frac{1-\alpha_m}{\beta \Delta t^2}\mathbf{M} + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}\mathbf{C} + (1-\alpha_f)\,\mathbf{K}\right]\mathbf{d}_{n+1} \\ = \mathbf{f}_{n+1-\alpha_f} + \mathbf{M}\left[\left(\frac{1-\alpha_m}{2\beta}-1\right)\ddot{\mathbf{d}}_n + \frac{1-\alpha_m}{\beta\,\Delta t}\dot{\mathbf{d}}_n + \frac{1-\alpha_m}{\beta\,\Delta t^2}\mathbf{d}_n\right] - \alpha_f \mathbf{K}\mathbf{d}_n \\ + \mathbf{C}\left\{\Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta}-1\right)\ddot{\mathbf{d}}_n + \left[\frac{\gamma(1-\alpha_f)}{\beta}-1\right]\dot{\mathbf{d}}_n + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t}\mathbf{d}_n\right\}\end{array}} \tag{3.49}$$

where matrix $\mathbf{C}$ is now defined as:

$$\mathbf{C} = \eta_M \mathbf{M} + \eta_K \mathbf{K} \tag{3.50}$$

3.5 Linear and nonlinear dynamic solvers

Elastodynamics is the simplest case one can encounter in structural mechanics. In that case, the space-time discretized linear system of equations is linear and finding the solution at any time t_{n+1} is straightforward. In most applications, however, material behavior is nonlinear since structural materials often dissipate energy and exhibit damage, permanent strains, etc. In that case, the resulting discretized problem to solve is nonlinear, and Newton–Raphson procedures can be used. In that case, the solution is found iteratively through solving a series of linearized problems.

3.5.1 Linear case - linear elastic material behavior

Let us start from the linear case first. Under small strains conditions, if the material is assumed isotropic linear elastic, the Cauchy's stress tensor reads $\boldsymbol{\sigma} = \lambda \mathrm{tr}(\boldsymbol{\epsilon})\mathbf{I} + 2\mu\boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} = (\nabla\mathbf{u} + \nabla^\top\mathbf{u})/2$ is the small strain tensor (i.e., the symmetric part of the displacement gradient $\nabla\mathbf{u}$), $\lambda = \lambda(\mathbf{x})$ and $\mu = \mu(\mathbf{x})$ are the Lame's parameters and $\mathbf{I} = \delta_{ij}\mathbf{e}_i \otimes \mathbf{e}_j$ denotes the second order identity tensor. As a consequence, the bi-linear form

$K(\mathbf{u}_{n+1}, \mathbf{v})$ and the corresponding linear form $k(\mathbf{v}; \mathbf{u}_n)$ can be rewritten in a more explicit form as:

$$\begin{aligned} K(\mathbf{u}_{n+1}, \mathbf{v}) &= \int_\Omega \boldsymbol{\sigma}\left(\mathbf{u}_{n+1}\right) : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V = \int_\Omega \left[\lambda \operatorname{tr} \boldsymbol{\epsilon}(\mathbf{u}_{n+1})\mathbf{I} + 2\mu\boldsymbol{\epsilon}(\mathbf{u}_{n+1})\right] : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V = \int_\Omega \boldsymbol{\epsilon}(\mathbf{u}_{n+1}) : \mathbb{E} : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V \\ k(\mathbf{v}; \mathbf{u}_n) &= \int_\Omega \boldsymbol{\sigma}\left(\mathbf{u}_{n}\right) : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V = \int_\Omega \left[\lambda \operatorname{tr} \boldsymbol{\epsilon}(\mathbf{u}_{n})\mathbf{I} + 2\mu\boldsymbol{\epsilon}(\mathbf{u}_{n})\right] : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V = \int_\Omega \boldsymbol{\epsilon}(\mathbf{u}_{n}) : \mathbb{E} : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V \end{aligned} \tag{3.51}$$

where $\mathbb{E}$ is the fourth-order elastic stiffness tensor.

3.5.2 Nonlinear case - inelastic material behaviors (under implementation)

An iterative Newton–Raphson procedure is used to solve the nonlinear problem. The unknown displacement $\mathbf{u}_{n+1}$ at global iteration $k+1$ is written as $\mathbf{u}_{n+1}^{k+1} = \mathbf{u}_{n+1}^{k} + \delta\mathbf{u}_{n+1}^{k+1}$, where $\mathbf{u}_{n+1}^{k}$ is the solution at iteration k, and $\delta\mathbf{u}_{n+1}^{k+1}$ is the solution variation at iteration $k+1$. The latter is computed from the resolution of a linearized system of equations.

For this purpose, the variational formulation (3.39) is first written in residual form as:

$$\boxed{\begin{aligned} &\text{Find } \mathbf{u}_{n+1}^{k+1} \in \mathcal{U} \text{ such that :} \\ &R(\mathbf{u}_{n+1}^{k+1}, \mathbf{v}) = \widetilde{\widetilde{K}}(\mathbf{u}_{n+1}^{k+1}, \mathbf{v}) - \widetilde{\widetilde{l}}(\mathbf{v}) = 0 \quad \forall\, \mathbf{v} \in \mathcal{V} \end{aligned}} \tag{3.52}$$

The residual(i.e., the out-of-balance force) is then linearized around solution $\mathbf{u}_{n+1}^{k}$ as follows:

$$R(\mathbf{u}_{n+1}^{k+1}, \mathbf{v}) = r(\mathbf{v}; \mathbf{u}_{n+1}^{k}) + R'(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}; \mathbf{u}_{n+1}^{k}) \tag{3.53}$$

where $r(\mathbf{v}; \mathbf{u}_{n+1}^{k}) = r(\mathbf{v}; \{\mathbf{b}, \mathbf{t}_n, \mathbf{t}_{n+1}, \mathbf{u}_n, \dot{\mathbf{u}}_n, \ddot{\mathbf{u}}_n\}, \mathbf{u}_{n+1}^{k})$ is the linear form corresponding to the out-of-balance forces at iteration k:

$$r(\mathbf{v}; \mathbf{u}_{n+1}^{k}) = \frac{1-\alpha_m}{\beta \Delta t^2} m(\mathbf{v}; \mathbf{u}_{n+1}^{k}) + \frac{\gamma\,(1-\alpha_f)}{\beta\, \Delta t} c(\mathbf{v}; \mathbf{u}_{n+1}^{k}) + (1-\alpha_f)\, k(\mathbf{v}; \mathbf{u}_{n+1}^{k}) - \widetilde{\widetilde{l}}(\mathbf{v}) \tag{3.54}$$

and:

$$\begin{aligned} R'(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}; \mathbf{u}_{n+1}^{k}) = &\left[\frac{1-\alpha_m}{\beta\, \Delta t^2} + \frac{\gamma\,(1-\alpha_f)\eta_M}{\beta\, \Delta t}\right] M(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}) \\ &+ (1-\alpha_f)\left(1 + \frac{\gamma\eta_K}{\beta\, \Delta t}\right) K_t(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}; \mathbf{u}_{n+1}^{k}) \end{aligned} \tag{3.55}$$

with:

$$K_t(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}; \mathbf{u}_{n+1}^{k}) = \int_\Omega \boldsymbol{\epsilon}(\delta\mathbf{u}_{n+1}^{k+1}) : \mathbb{D}^k : \boldsymbol{\epsilon}(\mathbf{v})\, \mathrm{d}V \tag{3.56}$$

The fourth order stiffness tensor $\mathbb{D}^k = D_{ijkl}(\mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l)$ can be defined differently according to the chosen algorithm. For instance, if a standard Newton–Raphson formulation is chosen, $\mathbb{D}^k = \partial_{\boldsymbol{\epsilon}}\boldsymbol{\sigma}(\mathbf{u}_{n+1}^{k})$ is the tangent stiffness operator at iteration k.

Finally, the discretized variational formulation to solve becomes:

$$\boxed{\begin{aligned} &\text{Find } \delta\mathbf{u}_{n+1}^{k+1} \in \mathcal{U}_\delta \text{ such that :} \\ &\left[\frac{1-\alpha_m}{\beta\, \Delta t^2} + \frac{\gamma\,(1-\alpha_f)\eta_M}{\beta\, \Delta t}\right] M(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}) + (1-\alpha_f)\left(\frac{\gamma}{\beta\, \Delta t}\eta_K + 1\right) K_d(\delta\mathbf{u}_{n+1}^{k+1}, \mathbf{v}; \mathbf{u}_{n+1}^{k}) \\ &= -r(\mathbf{v}; \mathbf{u}_{n+1}^{k}) \quad \forall\, \mathbf{v} \in \mathcal{V} \end{aligned}} \tag{3.57}$$

where $\mathcal{U}_\delta$ is the admissibility space of the displacement variations, and $-r(\mathbf{v}; \mathbf{u}_{n+1}^{k})$ can now be interpreted as the difference between the pseudo-external forces (represented by the linear form $\widetilde{\widetilde{l}}(\mathbf{v})$) and the internal forces (first three terms of equation (3.54)).

3.6 Paraxial formulation for absorbing layers

When spatially unbounded (infinite) domains are represented through bounded computational domains, spurious wave reflections can be observed boundaries. Several techniques have been proposed in the literature to introduce proper treatments of the boundary conditions allowing to reproduce propagation processes in infinite one-phase and two-phase media artificially. Among the available formulations, one can cite the ones based upon using lumped dumpers [**lysmer1969finite**], Perfectly Matched Layers (PML) [**berenger1994perfectly**] and paraxial boundaries [**engquist1977absorbing**, **clayton1977absorbing**, **aubry1985local**, **MODARESSI1994117**].

3.6.1 Standard formulation

Paraxial approximation constitutes a local boundary condition which permits diffracting waves to be evacuated from the computational domain. To introduce the formulation, let us consider a split of the total domain Ω^∞ into two subdomains Ω and Ω^E separated by a surface $\Sigma \in \mathbb{R}^d$ of outer normal $\mathbf{m}$ (pointing from Ω to Ω^E). On surface Σ, the continuity condition of the displacement field read:

$$[|\mathbf{u}|] = \mathbf{u} - \mathbf{u}^E = 0 \qquad \Sigma \times [0,T] \tag{3.58}$$

whereas the continuity of the traction vector reads:

$$[|\boldsymbol{\sigma}|] \cdot \mathbf{m} = \left(\boldsymbol{\sigma} - \boldsymbol{\sigma}^E\right) \cdot \mathbf{m} = \boldsymbol{\sigma} \cdot \mathbf{m} + \boldsymbol{\sigma}^E \cdot (-\mathbf{m}) = \mathbf{t} + \mathbf{t}^E = 0 \qquad \Sigma \times [0,T] \tag{3.59}$$

In previous equations, symbol $[|\bullet|]$ is used to denote the jump of function $\bullet$ across surface Σ, $\mathbf{u}^E = \mathbf{u}^E(\mathbf{x},t) : \Omega^E \times [0,T] \to \mathbb{R}^d$ is the vector-valued displacement field on Ω^E, and $\boldsymbol{\sigma}^E = \boldsymbol{\sigma}^E(\mathbf{u}^E) : \Omega^E \times [0,T] \to \mathbb{R}^{d\times d}$ is the corresponding stress tensor.

Variational formulation

Given the traction continuity condition (3.59), the variational problem to solve on Ω reads:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u} \in \mathcal{U} \text{ such that:} \\ M(\ddot{\mathbf{u}}, \mathbf{v}) + C(\dot{\mathbf{u}}, \mathbf{v}) + K(\mathbf{u}, \mathbf{v}) = b(\mathbf{v}; \mathbf{b})_\Omega + b(\mathbf{v}; \mathbf{t})_{\partial_t\Omega} - b(\mathbf{v}; \mathbf{t}^E)_\Sigma \quad \forall\, \mathbf{v} \in \mathcal{V}\end{array}} \tag{3.60}$$

Using a zeroth-order paraxial approximation, the traction vector $\mathbf{t}^E$ can be written as:

$$\mathbf{t}^E = A_0(\dot{\mathbf{u}}) = \rho c_p \dot{u}_m \mathbf{m} + \rho c_s \dot{\mathbf{u}}_s \tag{3.61}$$

where (c_p, c_s) are the the propagation velocities of compressional and shear waves, $\dot{u}_m = \dot{\mathbf{u}} \cdot \mathbf{m}$ is the velocity normal to Σ and $\dot{\mathbf{u}}_s = \dot{\mathbf{u}} - \dot{u}_m \mathbf{m}$ its tangent counterpart.

More in general, given a vector-valued field $\mathbf{a}$ (e.g., the displacement, velocity and acceleration fields), we write:[8]

$$A_0(\mathbf{a}) = \rho c_p \left(\mathbf{m} \otimes \mathbf{m}\right) \cdot \mathbf{a} + \rho c_s (\mathbf{I} - \mathbf{m} \otimes \mathbf{m}) \cdot \mathbf{a} = \rho \left[(c_p - c_s)\, m_i m_j + c_s \delta_{ij}\right] a_j \mathbf{e}_i \tag{3.62}$$

where symbol $\otimes$ denotes the dyadic product between first order tensors (vectors).[9]

[8] We recall that given three Euclidean vectors $\mathbf{v} \in \mathbb{R}^d$, $\mathbf{w} \in \mathbb{R}^d$ and $\mathbf{z} \in \mathbb{R}^d$, the dyadic product $\mathbf{v} \otimes \mathbf{w} \in \mathbb{R}^{d\times d}$ is the second order tensor defined by: $(\mathbf{v} \otimes \mathbf{w}) \cdot \mathbf{z} = (\mathbf{w} \cdot \mathbf{z})\mathbf{v}$. In components: $(\mathbf{v} \otimes \mathbf{w})_{ij} = v_i\, w_j$.

[9] Denoting (m_x, m_y, m_z) the components of vector $\mathbf{m}$ in the reference system $R(O, \mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z)$, the components of $A_0(\mathbf{a})$ read:

$$(A_0(\mathbf{a}))_x = \rho\left[(c_p - c_s) m_x (m_x a_x + m_y a_y + m_z a_z) + c_s\, a_x\right] \tag{3.63}$$

$$(A_0(\mathbf{a}))_y = \rho\left[(c_p - c_s) m_y (m_x a_x + m_y a_y + m_z a_z) + c_s\, a_y\right] \tag{3.64}$$

$$(A_0(\mathbf{a}))_z = \rho\left[(c_p - c_s) m_z (m_x a_x + m_y a_y + m_z a_z) + c_s\, a_z\right] \tag{3.65}$$

Time-discretization

After time discretization, equation (3.61) becomes:

$$\begin{aligned} b(\mathbf{v};\mathbf{t}^E)_\Sigma = \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t} b(\mathbf{v};A_0(\mathbf{u}_{n+1}))_\Sigma - \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right] b(\mathbf{v};A_0(\dot{\mathbf{u}}_n))_\Sigma \\ \cdots - \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right) b(\mathbf{v};A_0(\ddot{\mathbf{u}}_n))_\Sigma - \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t} b(\mathbf{v};A_0(\mathbf{u}_n))_\Sigma \end{aligned} \tag{3.66}$$

The variational problem to solve thus reads:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}_{n+1} \text{ such that :} \\ \tilde{\tilde{K}}(\mathbf{u}_{n+1},\mathbf{v})(\mathbf{u}_{n+1},\mathbf{v}) = \tilde{\tilde{l}}(\mathbf{v}) \quad \forall\,\mathbf{v}\in\mathcal{V}\end{array}} \tag{3.67}$$

where $\tilde{\tilde{K}}(\mathbf{u}_{n+1},\mathbf{v})$ is:

$$\begin{aligned} \tilde{\tilde{K}}(\mathbf{u}_{n+1},\mathbf{v}) &= \tilde{\tilde{K}}(\mathbf{u}_{n+1},\mathbf{v}) + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t} b(\mathbf{v};A_0(\mathbf{u}_{n+1}))_\Sigma \\ &= \frac{1-\alpha_m}{\beta\Delta t^2} M(\mathbf{u}_{n+1},\mathbf{v}) + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t} C(\mathbf{u}_{n+1},\mathbf{v}) + (1-\alpha_f)\,K(\mathbf{u}_{n+1},\mathbf{v}) \\ &\cdots + \frac{\gamma\,(1-\alpha_f)}{\beta\,\Delta t} b(\mathbf{v};A_0(\mathbf{u}_{n+1}))_\Sigma \end{aligned} \tag{3.68}$$

and $\tilde{\tilde{l}}(\mathbf{v})$ is:

$$\begin{aligned} \tilde{\tilde{l}}(\mathbf{v}) = \tilde{\tilde{l}}(\mathbf{v}) + \left[\frac{\gamma(1-\alpha_f)}{\beta} - 1\right] b(\mathbf{v};A_0(\dot{\mathbf{u}}_n))_\Sigma \\ \cdots + \Delta t(1-\alpha_f)\left(\frac{\gamma}{2\beta} - 1\right) b(\mathbf{v};A_0(\ddot{\mathbf{u}}_n))_\Sigma + \frac{\gamma(1-\alpha_f)}{\beta\,\Delta t} b(\mathbf{v};A_0(\mathbf{u}_n))_\Sigma \end{aligned} \tag{3.69}$$

3.6.2 Accounting for incident waves

Let us now split the total displacement vector at Σ into its incident $\mathbf{u}_{in}$ and radiant $\mathbf{u}_r$ components:

$$\mathbf{u} = \mathbf{u}^E = \mathbf{u}_{in} + \mathbf{u}_r \tag{3.70}$$

and use the zeroth-order paraxial approximation for expressing the traction contribution to the traction vector $\mathbf{t}^E$ due to the radiant field. The traction continuity condition (3.59), together with the linearity hypotheses at the vicinity of Σ, enables us to write:

$$\mathbf{t} = -\mathbf{t}^E(\mathbf{u}^E) = -\mathbf{t}^E(\mathbf{u}_{in}) - \mathbf{t}^E(\mathbf{u}_r) = -\mathbf{t}^E(\mathbf{u}_{in}) - A_0(\dot{\mathbf{u}}_r) = -\mathbf{t}^E(\dot{\mathbf{u}}_{in}) - A_0(\dot{\mathbf{u}}) + A_0(\dot{\mathbf{u}}_{in}) \tag{3.71}$$

here $\dot{\mathbf{u}}_{in}$ is known, whereas $\dot{\mathbf{u}}$ is the unknown velocity field.

The variational equation to solve now reads:

$$\boxed{\begin{array}{l}\text{Find } \mathbf{u}\in\mathcal{U} \text{ such that :} \\ M(\ddot{\mathbf{u}},\mathbf{v}) + C(\dot{\mathbf{u}},\mathbf{v}) + K(\mathbf{u},\mathbf{v}) = b(\mathbf{v};\mathbf{b})_\Omega + b(\mathbf{v};\mathbf{t})_{\partial_t\Omega} - b(\mathbf{v};A_0(\dot{\mathbf{u}}))_\Sigma \\ \quad \cdots - b(\mathbf{v};\mathbf{t}_E(\dot{\mathbf{u}}_{in}))_\Sigma + b(\mathbf{v};A_0(\dot{\mathbf{u}}_{in}))_\Sigma \quad \forall\,\mathbf{v}\in\mathcal{V}\end{array}} \tag{3.72}$$

where the last two terms are the only novelty with respect to equation (3.60).

Chapter 4

Tutorials

Preliminaries
Before diving into the tutorials, here are some preliminaries that will help you guide easily through them.

- A PSD simulation is performed in three steps: preprocessing, solving, and postprocessing.
- Domain: denoted by Ω is a a n-dimensional solid body such that $\Omega \subset \mathbb{R}^n$ with $n = 2$ for 2D problems or $n = 3$ for 3D problems.
- Finite element mesh: denoted by Ω^h with mesh size h. Mesh can be triangular in 2D and tetrahedral in 3D.
- MPI processes for simulation: denoted by N_p these are the total MPI ranks that will work in parallel to solve the problem.
- Partitioned mesh: denoted by $\{\Omega_i^h\}_{i=1}^{N_\mathrm{p}}$ these are set of subdomains which are held by each MPI rank during a parallel simulation.

4.1 Linear Elasticity

Linear Elasticity is a mathematical approximation of solid object deformation caused by prescribed loading conditions. It is a simplification of the more general nonlinear theory of elasticity. PSD allows for solving Linear Elasticity problems both in sequential and in parallel. We shall discuss how to do so in details within this section.

PSD is a FEM based solver, to solve a given physics it heavily relies on the variational formulations of the underlying physics. Let us begin with writing the variational formulation of system of Elasticity in which the primary unknown is the displacements vector $\mathbf{u} = \{u_j\}_{j=1}^n$. In the Lagrangian FE framework for searching the unknown nodal displacements vector $\mathbf{u}^h = \{u_j^h\}_{j=1}^n$ the variational formulation of system of Elasticity reads,

$$\forall i \in [\![1; N_\mathrm{p}]\!], \int_{\Omega_i^h} \boldsymbol{\sigma}(\boldsymbol{u}^h) : \boldsymbol{\varepsilon}(\boldsymbol{v}^h) = \int_{\partial\Omega_{i,\mathrm{N}}^h} \mathbf{f} \cdot \boldsymbol{v}^h \quad \forall\, \boldsymbol{v}^h \in \mathbb{V}^h, \boldsymbol{u}^h \in \mathbb{V}^h, \tag{4.1}$$

here, $\boldsymbol{u}^h$ is in fact the FE trial function and $\boldsymbol{v}^h = \{v_j^h\}_{j=1}^n$ is the FE test function.

$$\forall i \in [\![1; N_\mathrm{p}]\!], \int_{\Omega_i^h} \lambda \nabla \cdot \boldsymbol{u}^h \nabla \cdot \boldsymbol{v}^h + \int_{\Omega_i^h} 2\mu \boldsymbol{\varepsilon}(\boldsymbol{u}^h) : \boldsymbol{\varepsilon}(\boldsymbol{v}^h) - \int_{\Omega_i^h} \mathbf{f} \cdot \boldsymbol{v}^h = 0, \quad \forall \boldsymbol{v}^h \in [H_0^1(\Omega_i^h)]^n \tag{4.2}$$

In these formulations λ and μ are the Lame's parameters, $\mathbf{f}$ is the body force vector.

4.1.1 PSD simulation of bar problem bending under own body weight

To showcase the usage of Linear elasticity, we shall discuss here an example of a 2D bar, which bends under its own load. The bar 5×1 m^2 in area is made up of material with $\rho = 8 \times 10^3$, $E = 200 \times 10^9$, and $\nu = 0.3$.

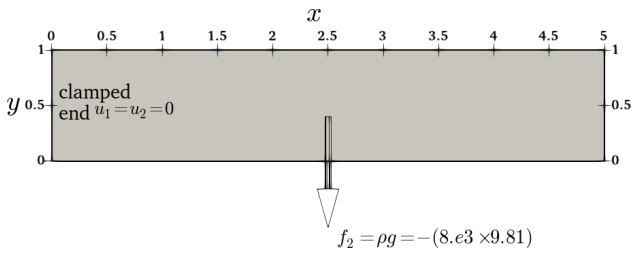


Figure 4.1: The 2D clamped bar problem.

Step 1: Preprocessing

First step in a PSD simulation is PSD preprocessing, at this step you tell PSD what kind of physics, boundary conditions, approximations, mesh, etc are you expecting to solve.

In the terminal cd to the folder /home/PSD-tutorials/linear-elasticity [1]. Launch PSD_PreProcess from the terminal, to do so run the following command.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u
```

After the PSD_PreProcess runs successfully you should see many .edp files in your current folder.

What do the arguments mean ?

- -problem linear_elasticity means that we are solving linear elasticity problem;
- -dimension 2 means it is a 2D simulation;
- -bodyforceconditions 1 with applied body force acting on the domain;
- -dirichletconditions 1 says we have one Dirichlet border;
- -postprocess u means we would like to have ParaView post processing files.

At this stage the input properties E, ν can be mentioned in ControlParameters.edp, use E = 200.e9, and nu = 0.3;. The volumetric body force condition is mentioned in the same file via variable Fbc0Fy -78480.0, i.e $(\rho * g = 8.e3 * (-9.81) = -78480.0)$. One can also provide the mesh to be used in ControlParameters.edp, via ThName = "../Meshes/2D/bar.msh" (*note that mesh can also be provided in the next step*) .In addition variable Fbc0On 1 has to be provided in order to indicate the volume (region) for which the body force is acting, here 1 is the integer volume tag of the mesh. Dirichlet boundary conditions are also provided in ControlParameters.edp. To provide the clamped boundary condition the variables Dbc0On 2, Dbc0Ux 0., and Dbc0Uy 0. are used, which means for Dirichlet border 2 (Dbc0On 2) where 2 is the clamped border label of the mesh Dirichlet constrain is applied and Dbc0Ux 0., Dbc0Uy 0 i.e., the clamped end condition $(u_x = u_y = 0)$.

Step 2: Solving

As PSD is a parallel solver, let us use 4 cores to solve the 2D bar case. To do so enter the following command:

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

[1] Note that one can perform these simulation in any folder provided that PSD has been properly installed. We use /home/PSD-tutorials/linear-elasticity for simplicity, once the user is proficient a simulation can be launch elsewhere.

Here -np 4 denote the argument used to enter the number of parallel processes (MPI processes) used while solving. -mesh ./../Meshes/2D/bar.msh is used to provide the mesh file to the solver. -v 0 denotes the verbosity level on screen. PSD_Solve is a wrapper around FreeFem++ or FreeFem++-mpi. Note that if your problem is large use more cores. PSD has been tested upto 13,000 parallel processes and problem sizes with billions of unknowns, surely you will now need that many for the 2D bar problem.

Step 3: Postprocessing

PSD allows postprocessing of results in ParaView. After the step 2 mentioned above finishes. Launch ParaView and have a look at the .pvd file in the VTUs... folder. Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.16 can be extracted.

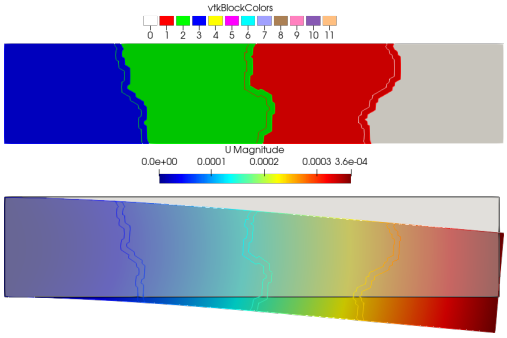


Figure 4.2: The 2D clamped bar problem: partitioned mesh and displacement field visualization in ParaView.

You are all done with your 2D linear-elasticty simulation.

2D bar is ok, but what about 3D ?

3D follows the same logic as 2D, in the preprocessing step

```
PSD_PreProcess -problem linear_elasticity -dimension 3 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u
```

note that all what has changed -dimension 3 instead of -dimension 2

Solving step remains exactly the same with -mesh flag now pointing towards the 3D mesh.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/3D/bar.msh -v 0
```

Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.3 can be extracted.

4.1.2 PSD simulation of of bar problem using a sequential solver (non parallel)

Same problem of Linear elasticity as in tutorial 1 – 2D bar which bends under its own load –, is discuss here. The bar 5×1 m^2 in area is made up of material with $\rho = 8 \times 10^3$, $E = 200 \times 10^9$, and $\nu = 0.3$. To avoid text repetition, readers are encouraged to go ahead with this tutorial only after tutorial 1.

As the problem remains same as tutorial 1, simply add -sequential flag to PSD_PreProcess flags from tutorial 1 for sequential solver. The flag -sequential signifies the use of sequential PSD solver. So the work flow for the 2D problem would be:

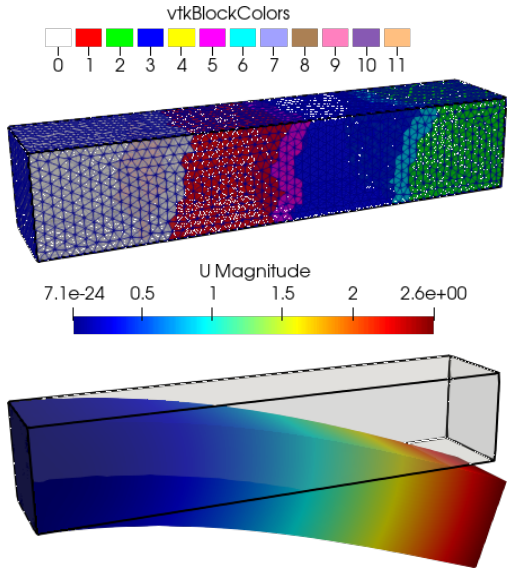


Figure 4.3: The 3D clamped bar problem: partitioned mesh and displacement field visualization in ParaView.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -sequential
```

Similar to tutorial 1, We solve the problem using the given mesh file bar.msh. However now we need to use PSD_Solve_Seq instead of PSD_Solve, as such:

```
PSD_Solve_Seq Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

Users are encouraged to try out the 3D problem with sequential solver.

Comparing CPU time

Naturally, since we are not using parallel PSD for solving, we lose the advantage of solving fast. To testify this claim checking solver timings can be helpful. PSD provides mean to time log your solver via -timelog flag. What this will do when you run your solver, on the terminal you will have information printed on what is the amount of time taken by each step of your solver. Warning, this will make your solver slower, as this action involves 'MPI_Barrier' routines for correctly timing operation.

An example work flow of 2D solver (parallel) with timelogging:

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -timelog
```

We solve the problem using four MPI processes, with the given mesh file bar.msh.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

The figure~4.4 shows the time logging output produced for parallel run on 4 processes using -timelog flag. Take note of timings produced for different operations of the solver.

Now let us repeat the procedure but this time use sequential solver:

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -timelog -sequential
```

```
-->Solver began....

-->Mesh Partitioning began....
finished in [ 4.790731e-01 ] seconds

-->matrix Assembly began....
finished in [ 1.469138e-01 ] seconds

-->RHS assembly began....
finished in [ 2.807666e-02 ] seconds

-->PETSc assembly began....
finished in [ 1.217071e-01 ] seconds

-->PETSc solving began....
finished in [ 3.263643e+00 ] seconds

-->Paraview Plotting began....
finished in [ 6.004273e-02 ] seconds

finished in [ 4.103547e+00 ] seconds
```

Figure 4.4: Time logging output produced for parallel run on 4 processes.

We solve the problem now in sequential, with the given mesh file bar.msh.

```
PSD_Solve_Seq Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

You should now see timings that are higher in comparison to the parallel solver. Approximately, for large meshes using 4 MPI processes should lead to 4 times fast solver.

4.1.3 PSD simulation of 2D bar problem clamped at both ends

To showcase the usage of Linear elasticity, we shall discuss here an example of a 2D bar, which bends under its own load. The bar 5×1 m^2 in area is made up of material with $\rho = 8 \times 10^3$, $E = 200 \times 10^9$, and $\nu = 0.3$. Contrary to tutorial 1, now both ends of the bar are clamped.

Step 1: Preprocessing

First step in a PSD simulation is PSD preprocessing , at this step you tell PSD what kind of physics, boundary conditions, approximations, mesh, etc are you expecting to solve.

In the terminal cd to the folder /home/PSD-tutorials/linear-elasticity. Launch PSD_PreProcess from the terminal, to do so run the following command.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 2 -postprocess u
```

After the PSD_PreProcess runs successfully you should see many .edp files in your current folder.

What do the arguments mean ?

- -problem linear_elasticity means that we are solving linear elasticity problem;
- -dimension 2 means it is a 2D simulation;
- -bodyforceconditions 1 with applied body force acting on the domain;
- -dirichletconditions 2 says we have two Dirichlet border;

- -postprocess u means we would like to have ParaView post processing files.

Since basic nature of both the problems (the one from tutorial 1 and 2) is same the almost the same command for preprocessing used in previous tutorial 1 is used here. The only difference,is that an additional Dirichlet condition needs to be supplied, notified to PSD by -dirichletconditions 2. To provide Dirichlet conditions of the left clamped end ($u_x = u_y = 0$) in ControlParameters.edp set Dbc0On 2, Dbc0Ux 0., and Dbc0Uy 0.. Similarly, for the right end set variables Dbc1On 4, Dbc1Ux 0., and Dbc1Uy 0. Each one of these is a clamped border respectively labeled as 2 (Dbc0On 2) and 4 (Dbc1On 4) in the mesh ../Meshes/2D/bar.msh.

Just like the previous tutorial the input properties E, ν should be mentioned in ControlParameters.edp, use E = 200.e9, and nu = 0.3;. The volumetric body force condition is mentioned in the same file via variable Fbc0Fy -78480.0, i.e $(\rho * g = 8.e3 * (-9.81) = -78480.0)$. One can also provide the mesh to be used in ControlParameters.edp , via ThName = "../Meshes/2D/bar.msh" (*note that mesh can also be provided in the next step*) .In addition variable Fbc0On 1 has to be provided in order to indicate the volume (region) for which the body force is acting, here 1 is the integer volume tag of the mesh.

Step 2: Solving

As PSD is a parallel solver, let us use 3 parallel processes to solve this 2D bar case. To do so enter the following command:

```
PSD_Solve -np 3 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

Here -np 3 denote the argument used to enter the number of parallel processes (MPI processes) used while solving. -mesh ./../Meshes/2D/bar.msh is used to provide the mesh file to the solver. -v 0 denotes the verbosity level on screen.PSD_Solve is a wrapper around FreeFem++ or FreeFem++-mpi. Note that if your problem is large use more cores. PSD has been tested upto 13,000 parallel processes and problem sizes with billions of unknowns, surely you will now need that many for the 2D bar problem.

Step 3: Postprocessing

PSD allows postprocessing of results in ParaView. After the step 2 mentioned above finishes. Launch ParaView and have a look at the .pvd file in the VTUs_DATE_TIME folder. Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.5 can be extracted.

Figure 4.5: The 2D clamped bar problem: partitioned mesh and displacement field visualization in ParaView.

You are all done with your 2D linear-elasticity simulation.

Try running the 3D problem. Keep in mind to rerun the PSD_PreProcess with -dimension 3 flag and using the appropriate mesh via -mesh flag with PSD_Solve. It goes without saying you will need to adjust the Dirichlet border labels in ControlParameters.edp.

Redoing the test on Jupiter and moon

Imagine, you wish to know how the test would compare if performed on Moon and Jupiter. Since gravity is the main force involved in the problem, try redoing the test with different gravitational constant. The

only thing that will change now is the gravitational pull, for Moon $g = 1.32$ and for Jupiter $g = 24.79$. To perform the moon test simply change Fbc0Fy -10560.0 in ControlParameters.edp and redo step 2 and step 3. Similarly, for the Jupiter test Fbc0Fy -198320.0 in ControlParameters.edp and redo step 2 and step 3.

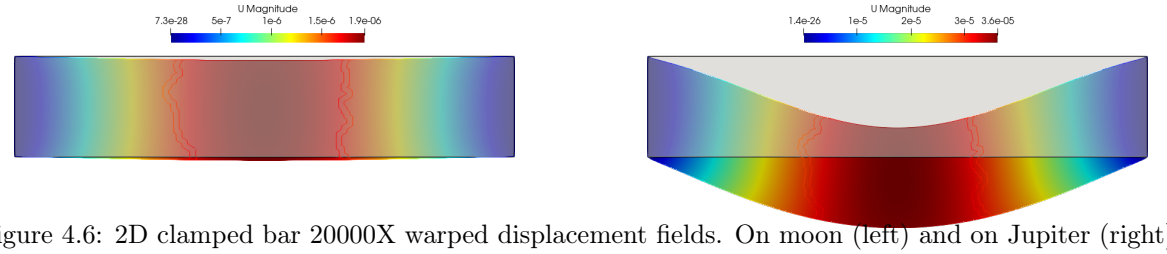


Figure 4.6: 2D clamped bar 20000X warped displacement fields. On moon (left) and on Jupiter (right).

4.1.4 PSD simulation of 2D bar problem clamped at one end wile being pulled at the other end (Dirichlet-Dirichlet case)

In this tutorial we showcase the 2D bar problem simulation with one end clamped wile being pulled at the other end. Body force is neglected and the non clamped ends pull is approximated with Dirichlet displacement $u_1 = 1$. If this simulation is compared to the previous one from tutorial 1 ans tutorial 2, the only difference now is that no body force is applied and an additional Dirichlet condition is applied at the free end of the bar. Here is how PSD simulation of this case can be performed. he bar 5×1 m^2 in area is made up of material with $\rho = 8 \times 10^3$, $E = 200 \times 10^9$, and $\nu = 0.3$.

Step 1: Preprocessing

First step in a PSD simulation is PSD preprocessing , at this step you tell PSD what kind of physics, boundary conditions, approximations, mesh, etc are you expecting to solve.

In the terminal cd to the folder /home/PSD-tutorials/linear-elasticity. Launch PSD_PreProcess from the terminal, to do so run the following command.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -dirichletconditions 2 \
-postprocess u
```

After the PSD_PreProcess runs successfully you should see many .edp files in your current folder.

What do the arguments mean ?

- -problem linear_elasticity means that we are solving linear elasticity problem;
- -dimension 2 means it is a 2D simulation;
- -dirichletconditions 2 says we have two Dirichlet border;
- -postprocess u means we would like to have ParaView post processing files.

In comparison to preprocessing from other two tutorials (tutorial 1 and 2), notice that the body force flag -bodyforceconditions 1 is missing. This is due to the fact that for this problem we assume null body force. -dirichletconditions 2, which notifies to PSD that there are two Dirichlet borders in this simulation i) the clamped end and ii) the pulled ends of the bar. To provide these Dirichlet conditions of the two ends in ControlParameters.edp set the variables Dbc0On 2, Dbc0Ux 0., and Dbc0Uy 0. signifying the clamped end ($u_x = 0, u_y = 0$ on mesh label 2) and Dbc1On 4, Dbc1Ux 1., and Dbc1Uy 0. signifying the pulled end ($u_x = 1, u_y = 0$ on label 4). Note that here at border 4 we have explicitly set $u_2 = 0$ this means the bar is not allowed to shrink (compress) in y direction, however you might wish to allow the bar to compress. For such a simulation simply use Dbc1On 4 and Dbc1Ux 1., and remove the term Dbc1Uy 0. therefor asking PSD not to apply constrain in y direction on the pulled end.

Just like the previous tutorial the input properties E, ν should be mentioned in ControlParameters.edp, use E = 200.e9, and nu = 0.3;. The volumetric body force condition is mentioned in the same file via variable Fbc0Fy -78480.0, i.e $(\rho * g = 8.e3 * (-9.81) = -78480.0)$. One can also provide the mesh to be used in ControlParameters.edp , via ThName = "../Meshes/2D/bar.msh" (*note that mesh can also be provided in the next step*) .In addition variable Fbc0On 1 has to be provided in order to indicate the volume (region) for which the body force is acting, here 1 is the integer volume tag of the mesh.

Step 2: Solving

As PSD is a parallel solver, let us use 2 parallel processes to solve this 2D bar case. To do so enter the following command:

```
PSD_Solve -np 2 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

Here -np 2 denote the argument used to enter the number of parallel processes (MPI processes) used while solving. -mesh ./../Meshes/2D/bar.msh is used to provide the mesh file to the solver. -v 0 denotes the verbosity level on screen. PSD_Solve is a wrapper around FreeFem++ or FreeFem++-mpi. Note that if your problem is large use more cores. PSD has been tested upto 13,000 parallel processes and problem sizes with billions of unknowns, surely you will now need that many for the 2D bar problem.

Step 3: Postprocessing

PSD allows postprocessing of results in ParaView. After the step 2 mentioned above finishes. Launch ParaView and have a look at the .pvd file in the VTUs_DATE_TIME folder. Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.7 can be extracted.

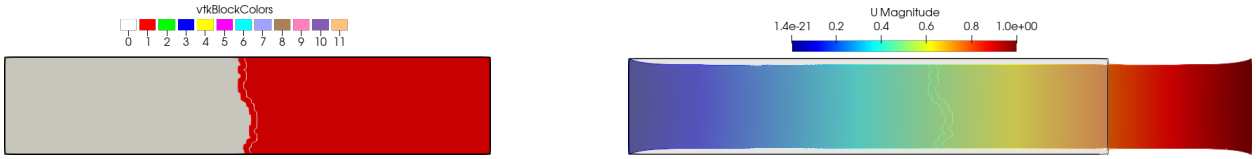


Figure 4.7: The 2D clamped bar problem: partitioned mesh and displacement field visualization in ParaView.

You are all done with your 2D linear-elasticity simulation.

4.1.5 PSD simulation of 2D bar problem clamped at one end wile being pulled at the other end (Dirichlet–Neumann case)

Similar simulation, as in the prvious tutorial is presented in this section. We showcase the 2D bar problem simulation with one end clamped wile being pulled at the other end. Just like the previous simulation the body force is neglected, However now the non clamped ends pull is approximated with Neumann force $\int_{\partial\Omega^h_N}(\mathbf{t}\cdot\mathbf{v}^h)$. To simulate the pull we assume traction vector $\mathbf{t} = [t_x, t_y] = [10^9., 0]$ acting on the non clamped right end of the bar, i.e., force in x direction is 10 units. Here is how PSD simulation of this case can be performed.

Step 1: Preprocessing

For "PSD preprocessing" go to any folder, launch the terminal there and run the following command.

```
PSD_PreProcess -problem linear-elasticity -dimension 2 -dirichletconditions 1 -tractionconditions 1 -postprocess u
```

the comandline flag -dirichletconditions 1, notifies to PSD that there is one Dirichlet border —the clamped end of the bar— in this simulation. And the flag -tractionconditions 1 notifies to PSD that there is one traction border —the right end of the bar— in this simulation. To provide the clamped boundary condition $(u_1 = 0, u_2 = 0)$ set the variables Dbc0On 2, Dbc0Ux 0., and Dbc0Uy 0. in ControlParameters.edp. In the same file traction boundary conditions are provided via the variables Tbc0On 4 and Tbc0Tx 1.e9, which mean apply traction force $\mathbf{t} = [t_x, t_y] = [10^9., 0]$ on label number 4 (right) of the mesh. If user wishes to add traction force ,for instance $t_y = 100.$, simply add the missing macro macro Tbc0Tx 1.e9 //.

Step 2: Solving

Let us now use 5 cores to solve this problem. To do so enter the following command:

```
PSD_Solve -np 5 Main.edp
```

Notice, that this is the exact same command used in solving the previous bar problems from other sections, with only difference that we now use -np 5.

Step 3: Postprocessing

Launch ParaView and have a look at the .pvd file in the PSD/Solver/VTUs_DATE_TIME folder.

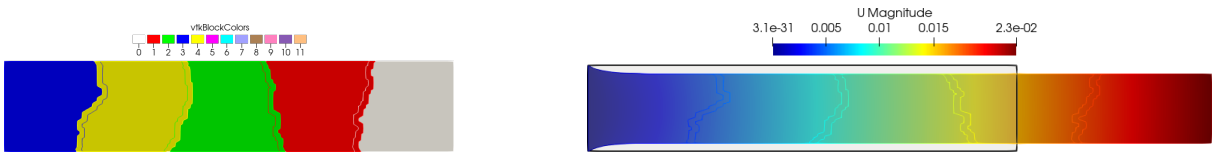


Figure 4.8: 2D bar results. Partitioned mesh (left) and 100X warped displacement field (right).

Note now in˜4.8 there are five subdomains in the partitioned mesh since five cores were used. Contrary to previous tutorial, as expected, we see that the right end of the bar which is being pulled now contract in y direction. This is due to the fact that there is no Dirichlet condition at this end now.

4.1.6 PSD simulation of 2D bar problem clamped at one end wile being pulled at the other end (Dirichlet-Neumann-Point boundary conditions case)

Similar simulations, as in the prvious tutorial is presented in this section. We showcase the 2D bar problem simulation with one end clamped wile being pulled at the other end. Contrary to simulation in the previous tutorial, the clamped end just restricts x movement, i.e, $u_x = 0$. Just like simulation from the prvious tutorial the body force is neglected. Just like simulation in the prvious tutorial , the non clamped ends pull is approximated with Neumann force $\int_{\partial\Omega^h_N}(\mathbf{t}\cdot\mathbf{v}^h)$. To simulate the pull we assume traction vector $\mathbf{t} = [t_x, t_y] = [10^9, 0]$ acting on the non clamped right end of the bar, i.e., force in x direction is 10^9 units. Here is how PSD simulation of this case can be performed.

Step 1: Preprocessing

For "PSD preprocessing" go to any folder, launch the terminal there and run the following command.

```
PSD_PreProcess -problem linear-elasticity -dimension 2 -dirichletconditions 1 -tractionconditions 1 \
-dirichletpointconditions 1 -postprocess u
```

Additional flag -dirichletpointconditions 1 now appears, this notifies to PSD that there is one Dirichlet point boundary condition. Edit the ControlParameters.edp to communicate the desired point boundary conditions, set the variables Pbc0Ux 0. and Pbc0Uy 0. to specify $u_x = 0, u_y = 0$, and variable PbcCord = [[0. , 0.]]; to specify the point coordinates $(x, y) = (0, 0)$. Via the flags we specified that -dirichletconditions 1, i.e., there is one Dirichlet border. To provide the Dirichlet condition $(u_x = 0)$ set the variables Dbc0On 2 and Dbc0Ux 0. in ControlParameters.edp. PSD understands that 4 is the mesh border label on which Dirichlet is applied and $(u_x = 0)$ is the condition to be applied.

Step 2: Solving

Let us now use 6 cores to solve this problem. To do so enter the following command:

```
PSD_Solve -np 6 Main.edp
```

% Notice, that this is the exact same command used in solving the previous bar problems from other sections, with only difference that we now use -np 6.

Step 3: Postprocessing

Launch ParaView and have a look at the .pvd file in the PSD/Solver/VTUs_DATE_TIME folder.

Figure 4.9: 2D bar results. Partitioned mesh (left) and 100X warped displacement field (right).

Note now in˜fig. 4.9 there are six subdomais in the partitioned mesh. As expected, we see that the right and the left end of the bar which is being pulled now contract in y direction, and the bar elongates in x direction.

4.1.7 PSD simulation of 3D bar problem clamped at one end wile being pulled at the other end (Dirichlet-Neumann case)

In this section we present a 3D PSD simulation of a clamped bar which his being loaded in vertical direction at the non-clamped end. This simulation is like the one presented in previous tutorials, however in 3D. The material properties are same as before, and at the non-clamped end traction $t_y = -10^9$ units. The 3D bar is $1 \times 1 \times 5$ m^3.

Here is how PSD simulation of this case can be performed.

Step 1: Preprocessing

For "PSD setup" go to any folder, launch the terminal there and run the following command.

```
PSD_PreProcess  -problem linear-elasticity -dimension 3 -dirichletconditions 1 -tractionconditions 1 -postprocess u
```

% the comandline flag -dirichletconditions 1 notifies to PSD that there is one Dirichlet border —the clamped end of the bar— in this simulation; -dimension 3 means the simulation is 3D. And the flag -tractionconditions 1 notifies to PSD that there is one traction border —the right end of the bar— in this simulation. To provide Dirichlet conditions of the clamped end ($u_x = 0, u_y = 0, u_z = 0$) in ControlParameters.edp set Dbc0On 1, Dbc0Ux 0., Dbc0Uy 0., and Dbc0Uz 0., where 1 being the surface mesh label of the clamped end. To add the traction boundary condition set Tbc0On 2 and Tbc0Ty -1.e9, here the mesh label number of the right end is 2. For this end $\mathbf{t} = [t_x, t_y, t_z] = [0., 10^9, 0.]$, hence in ControlParameters.edp we only use Tbc0Ty -1.e9.

Step 2: Solving

Let us now use 4 cores to solve this problem. To do so enter the following command:

```
PSD_Solve -np 4 Main.edp
```

% Notice, that this is the exact same command used in solving the previous bar problems from other sections.

Step 3: Postprocessing

Launch ParaView and have a look at the .pvd file in the PSD/Solver/VTUs_DATE_TIME folder.

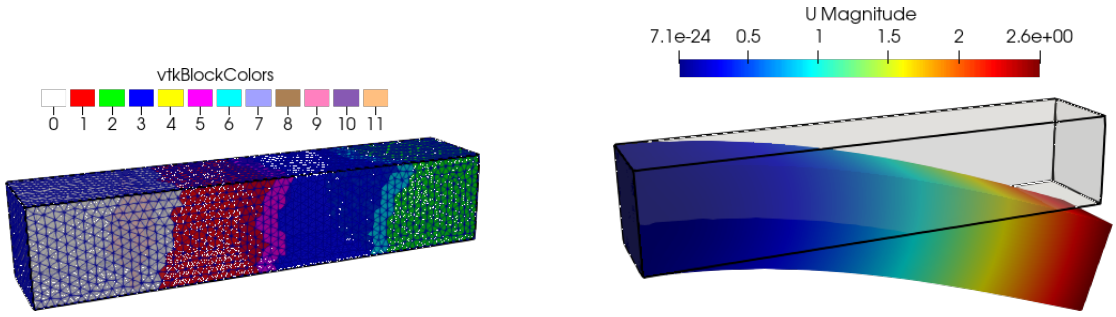


Figure 4.10: 3D bar results. Partitioned mesh (left) and 0.5X warped displacement field (right).

In˜fig. 4.10 there are four subdomais in the partitioned mesh since four cores were used.

4.1.8 PSD simulation of 3D mechanical piece (Dirichlet-Neumann case) with complex mesh

So far in the previous cases we only concentrated on bar simulations, which were more or less trivial cases. Moreover, the bar meshes are provided with the PSD solver. In this section we now turn towards 3D simulation of a mechanical piece, the geometry of which is shown in˜fig. 4.11. The left (small) hole is fixed: $u_1 = u_2 = u_3 = 0$, while as traction force $t_x = 10^9$ is applied on the large hole.

You can grab a copy of CAD geometry for the mechanical piece (the Gmsh .geo}) your local Gmsh installation folder gmsh/share/doc/gmsh/demos/simple_geo/{piece}.geo}. The listing of the file is also given in @. To generate the mesh piece.msh} simply do

```
gmsh -3 piece.geo
```

Place the generated mesh piece.msh} in /PSD/Meshes/3D/piece.msh}. Now the PSD simulation can be performed.

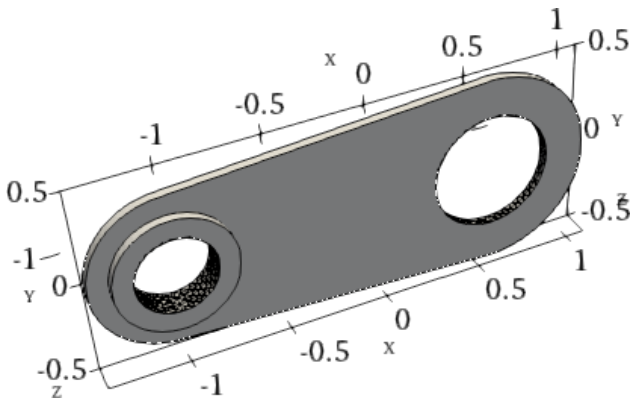


Figure 4.11: 3D mechanical piece.

Step 1: Preprocessing

For "PSD setup" go to any folder, launch the terminal there and run the following command.

```
PSD_PreProcess  -problem linear-elasticity -dimension 3 -dirichletconditions 1 -tractionconditions 1 -postprocess u
```

Here, by using these parameters we have generated one Dirichlet condition and one traction condition, respectively to be applied to the small and the large holes in the mesh. Further, by using -dimension 3} we have let PSD know that the problem is 3D .In the /PSD/Meshes/3D/piece.msh} generated, the label 4 (resp.˜3) corresponds to the Dirichlet (resp.˜traction) border. To provide Dirichlet conditions on label number 4 ($u_x = 0, u_y = 0, u_z = 0$) in ControlParameters.edp} use set Dbc0On 4}, Dbc0Ux 0.}, Dbc0Uy 0.}, and Dbc0Uz 0.}. To add the values and label numbers of the traction borders edit the ControlParameters.edp}, set Tbc0On 3} and Tbc0Ty -1.e9}. For this end $\mathbf{t} = [t_x, t_y, t_z] = [0., 10^9, 0.]$. Finally we use steel properties for the material, so in ControlParameters.edp} the parameters real E = 200.e9;} and real nu = 0.3;} should be used. These represent E and ν, respectively. With all the properties and boundary conditions set we now use string ThName = "../Meshes/3D/piece";} in the ControlParameters.edp} file, this notifies PSD about the name of the mesh used for this simulation.

Step 2: Solving

Let us now use 2 cores to solve this problem. To do so enter the following command:

```
PSD_Solve -np 2 Main.edp
```

Step 3: Postprocessing

Launch ParaView and have a look at the .pvd} file in the PSD/Solver/VTUs_DATE_TIME} folder.

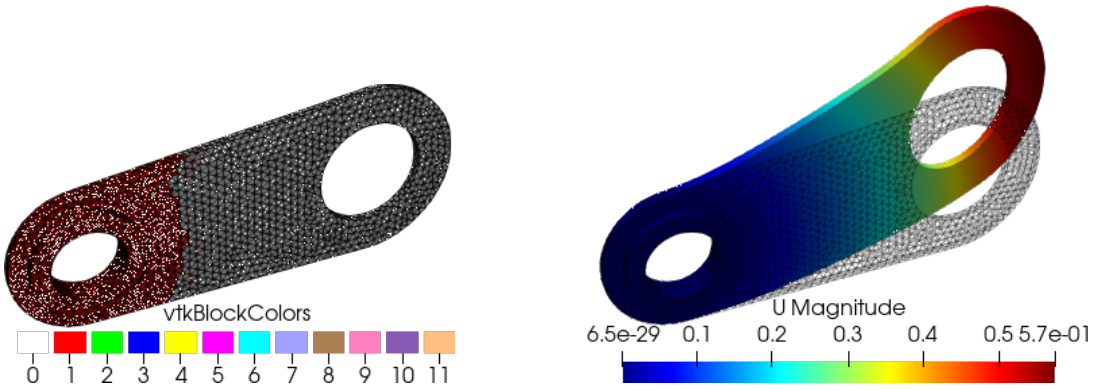


Figure 4.12: Mechanical piece test results. Partitioned mesh (left) and warped displacement field (right).

Redoing the test with different conditions

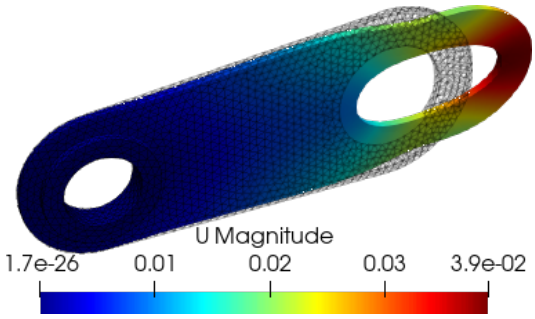


Figure 4.13: Mechanical piece test results: real tx0=1.e9, ty0=0, tz0=0.,;

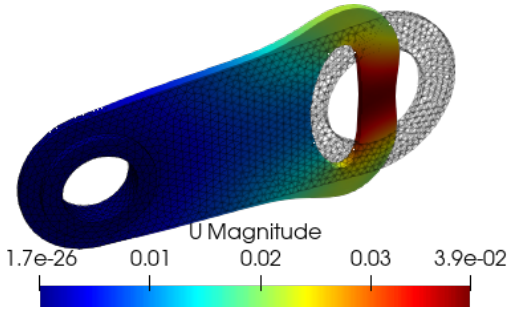


Figure 4.14: Mechanical piece test results:real tx0=1.e9, ty0=0, tz0=0.,;

4.1.9 PSD linear elasticity tutorial using Mfront-PSD interface

We reintroduce the problem from tutorial 1, an example of a 2D bar which bends under its own load – typical case of linear elasticity. The bar 5×1 m^2 in area is made up of material with $\rho = 8 \times 10^3$, $E = 200 \times 10^9$, and $\nu = 0.3$.

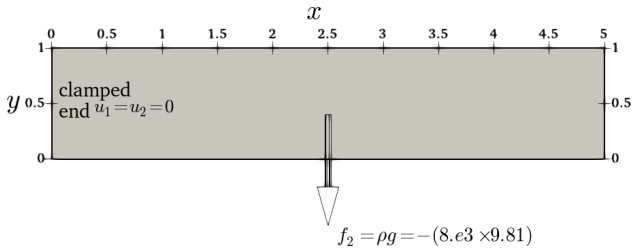


Figure 4.15: The 2D clamped bar problem.

Step 1: Preprocessing

First step in a PSD simulation is PSD preprocessing, at this step you tell PSD what kind of physics, boundary conditions, approximations, mesh, etc are you expecting to solve. More importantly for this tutorial we will signify to PSD that MFront has to be used.

In the terminal cd to the folder /home/PSD-tutorials/linear-elasticity. Launch PSD_PreProcess from the terminal, to do so run the following command.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -useMfront
```

After the PSD_PreProcess runs successfully you should see many .edp files in your current folder.

What do the arguments mean ?

- -problem linear_elasticity means that we are solving linear elasticity problem;
- -dimension 2 means it is a 2D simulation;
- -bodyforceconditions 1 with applied body force acting on the domain;
- -dirichletconditions 1 says we have one Dirichlet border;
- -postprocess u means we would like to have ParaView post processing files.
- -useMfront activates MFront interface for PSD.

At this stage the input properties E, ν can be mentioned in ControlParameters.edp, use E = 200.e9, and nu = 0.3. In contrast to tutorial 1, notice that these values of E and nu are fed to a vector PropertyValues = [E, nu]; verbosed by PropertyNames = "YoungModulus PoissonRatio";. We also signify that we will be solving linear elasticity via MfrontMaterialBehaviour = "Elasticity"; and also MaterialHypothesis = "GENERALISEDPLANESTRAIN"; which signifies the hypothesis to be used for the Linear elasticity [2]. PropertyValues, PropertyNames, and MaterialHypothesis will eventually be provided to MFront in FemParameters.edp file via mfrontElasticityHandler(...) function [3]. The volumetric body force condition is mentioned in the same file via variable Fbc0Fy -78480.0, i.e ($\rho * g = 8.e3 * (-9.81) = -78480.0$). One can also provide the mesh to be used in ControlParameters.edp, via ThName = "../Meshes/2D/bar.msh" (*note that mesh can also be provided in the next step*) .In addition variable Fbc0On 1 has to be provided in order to indicate the volume (region) for which the body force is acting, here 1 is the integer volume tag of the mesh. Dirichlet boundary conditions are also provided in ControlParameters.edp. To provide the clamped boundary condition the variables Dbc0On 2, Dbc0Ux 0., and

[2] The MaterialHypothesis accepts "GENERALISEDPLANESTRAIN", "PLANESTRAIN", "PLANESTRESS", and "TRIDIMENSIONAL" as arguments.

[3] User is encouraged to have a look at FemParameters.edp file.

Dbc0Uy 0. are used, which means for Dirichlet border 2 (Dbc0On 2) where 2 is the clamped border label of the mesh Dirichlet constrain is applied and Dbc0Ux 0., Dbc0Uy 0 i.e., the clamped end condition ($u_x = u_y = 0$).

Step 2: Solving

As PSD is a parallel solver, let us use 4 cores to solve the 2D bar case. To do so enter the following command:

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

Here -np 4 denote the argument used to enter the number of parallel processes (MPI processes) used while solving. -mesh ./../Meshes/2D/bar.msh is used to provide the mesh file to the solver. -v 0 denotes the verbosity level on screen. PSD_Solve is a wrapper around FreeFem++ or FreeFem++-mpi. Note that if your problem is large use more cores. PSD has been tested upto 13,000 parallel processes and problem sizes with billions of unknowns, surely you will now need that many for the 2D bar problem.

Step 3: Postprocessing

PSD allows postprocessing of results in ParaView. After the step 2 mentioned above finishes. Launch ParaView and have a look at the .pvd file in the VTUs... folder. Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.16 can be extracted.

Figure 4.16: The 2D clamped bar problem: partitioned mesh and displacement field visualization in ParaView.

You are all done with your 2D linear-elasticty simulation with Mfront interface.

How and what is being done in PSD-MFront interface?

To explain how PSD-MFront interface works we will compare how a PSD solver acts when using MFront or without. In other words what is different when -useMfront is used at preprocessing. Note that ultimately the problem results (displacement fields, stresses, strains) will be the same.

To put it briefly, what MFront does for linear elasticity problem here is build the Material tensor (stiffness matrix) at each quadrature point. So, there are two points

- We need to communicate to Mfront the nature of the problem and the material involved.
- We need to provide Mfornt the stiffness matrix at each quadrature point so that it can fill it up.

The two raised points are handled using mfrontElasticityHandler(...) in FemParameters.edp file.

Firstly, the arguments E = 200.e9, nu = 0.3, MforntMaterialBehaviour = "Elasticity";, PropertyValues = [E, nu];, PropertyNames = "YoungModulus PoissonRatio";, PropertyValues = [E, nu]; and MaterialHypothesis = "GENERALISED-PLANESTRAIN"; form ControlParameters.edp takes care of the first point (the nature of the problem and the material involved). The latter three arguments well define that we have a 2D problem, with given values of properties (E, ν). The snippet from ControlParameters.edp (produced after using -useMfront argument for PSD_PreProcess) file shows these variables which define the nature of the problem and characteristics of material involved

```
//=========================================================================
//                  ------- Material parameters -------
// ----------------------------------------------------------------------
//  E, nu : Modulus of Elasticity and Poisson ratio of the material
//  PropertyNames : String of material property names (space seperated)
//                  that are provided to Mfront.
//  PropertyValues : Values of material properties provided to Mfront
//
// ----------------------------------------------------------------------
//  NOTE:     Please note that PropertyNames should be the same as
//            as in the Elasticity.mfront file
// ----------------------------------------------------------------------
//=========================================================================

 macro E()  200.e9  //
 macro nu() 0.3    //

 string   MaterialBehaviour  = "Elasticity";
 string   MaterialHypothesis = "GENERALISEDPLANESTRAIN";
 string   PropertyNames      = "YoungModulus PoissonRatio";
 real[int] PropertyValues    = [ E, nu ];
```

Secondly, the to get the stiffness matrix from Mfornt we use a quadrature finite element space with vector finite elements built on it (6 components) that represent the 6 components of symmetric material tensor ($\mathbb{R}^{3\times3}$). The snippet from MeshAndFeSpace.edp file shows the 6 component Quadrature finite element space for building material tensor.

```
//=========================================================================
// ------- The finite element space  -------
// -------------------------------------------------------------------------
//  Qh       : Quadratur finite element space  for material tensor
//             FEQF2 implies 3 dof for a triangular cell in the mesh
//             A vectorial FEM space is built with 6 components
//=========================================================================

fespace Qh  ( Th ,[ FEQF2, FEQF2, FEQF2,
                    FEQF2, FEQF2,
                         FEQF2] );
```

Finally in file FemParameters.edp the mfrontElasticityHandler() is called to build the material tensor Mt provided with the previously built material properties and nature of problem. Please see the snippet below

```
//=========================================================================
// ------- Material Tensor using Quadrature FE space -------
// ----------------------------------------------------------------
// Mt[int]  : is an array of finite element variable belonging to quadratu
//            re space Qh. This array is used  to define components of the
//            material tensor. 3X3 in 2D and 6X6 in 3D
//            In 2D the material tensor looks like
//
//         [ 2*mu+lambda ,  lambda      , 0 ]    [ Mt11 , Mt12 , Mt13 ]
//   Mt =  [ lambda      ,  2*mu+lambda , 0 ] =  [ Mt12 , Mt22 , Mt23 ]
//         [   0         ,     0        , mu]    [ Mt13 , Mt23 , Mt33 ]
//
// mfrontElasticityHandler : is a function in mfront interface that helps
//                           building the material tensor  Mt  given with
//                           material prpts.  from  ControlParameters.edp
//=========================================================================
```

```
17
18 Qh [ Mt11 , Mt12 , Mt13 ,
19         Mt22 , Mt23 ,
20              Mt33 ];
21
22
23 mfrontElasticityHandler( MforntMaterialBehaviour                    ,
24                   mfrontBehaviourHypothesis = MaterialHypothesis    ,
25                   mfrontPropertyNames       = PropertyNames         ,
26                   mfrontPropertyValues      = PropertyValues        ,
27                   mfrontMaterialTensor      = Mt11[]
28                  );
```

Note that in the snippet above you might be seeing Mt11[] being provided as mfrontMaterialTensor, in fact the Mt11[] calls the full matrial tensor not just the first component, so user should not get confused [4].

The material tensor Mt built is used in the finite element variational formulation to build the bilinear $a(\mathbf{u}, \mathbf{v})$ which is used to assemble the finite element matrix $\mathbf{A}$ for the linear system $\mathbf{Ax} = \mathbf{b}$

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} (\varepsilon(\mathbf{u}) : \mathbf{Mt} : \varepsilon(\mathbf{v}))$$

Here, $\mathbf{u} : \mathbf{Mt}$ is nothing but the stress $\sigma(\mathbf{u})$ operator. User is encourage to have a look at the VariationalFormulation.edp file that contains the variational formulation (weak form) of the problem described.

4.1.10 Additional exercises on linear elasticity

Advance exercise 1

There is a solver run level flag for mesh refinement [5]. This flag is called -split [int] which splits the triangles (resp. tetrahedrons) of your mesh into four smaller triangles (resp. tetrahedrons). As such -split 2 will produce a mesh with 4 times the elements of the input mesh. Similarly, -split n where n is a positive integer produces 2^n times more elements than the input mesh. You are encouraged to use this -split flag to produce refined meshes and check, mesh convergence of a problem, computational time, etc. Use of parallel computing is recommended. You could try it out with PSD_Solve or PSD_Solve_Seq, for example:

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0 -split 2
```

for splitting each triangle of the mesh bar.msh into 4.

Advance exercise 2

There is a preprocess level flag -debug, which as the name suggests should be used for debug proposes by developers. However, this flag will activate OpenGL live visualization of the problems displacement field. You are encouraged to try it out

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -timelog -debug
```

Then to run the problem we need additional -wg flag

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0 -wg
```

[4] This is more technical note, Mt11[] is the cast of Mt vector to a single array for memory optimization. One can also simply use Mt12[], Mt13[], Mt22[], ... all these are acceptable and are simply aliases to material tensor.

[5] Mesh refinement is performed after partitioning.

Advance Exercise 3

One interesting way of solving a linear Elasticity problem is to solve it via a pseudo nonlinear model. There is a preprocess level flag -model pseudo_nonlinear, which introduces pseudo nonlinearity into the finite element variational formulation of linear elasticity. You are encouraged to use this flag and see how the solver performs. Indeed, now you should see some nonlinear iterations (1 or 2) are taken for convergence.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -timelog -model pseudo_nonlinear
```

Then to run the problem

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

To understand what the flag does, try to find out the difference between the files created by PSD_PreProcess when used with and without -model pseudo_nonlinear flag. Especially, compare LinearFormBuilderAndSolver.edp and VariationalFormulations.edp files produced by PSD_PreProcess step. You will see Newton–Raphsons iterations are performed for solving the linear problem. However, the nonlinear iterations loop converges very rapidly (in 1 iteration) due to linear nature of the problem. **Note:** This flag is exclusive for parallel solver.

Advance exercise 4

There is a preprocess level flag -withmaterialtensor, which introduces the full material tensor into the finite element variational formulation. You are encouraged to use this flag and see how the solver performs.

```
PSD_PreProcess -problem linear_elasticity -dimension 2 -bodyforceconditions 1 \
-dirichletconditions 1 -postprocess u -timelog -withmaterialtensor
```

Then to run the problem

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/bar.msh -v 0
```

To understand what the flag does, try to find out the difference between the files created by PSD_PreProcess when used with and without -withmaterialtensor flag. Especially, compare FemParameters.edp, MeshAndFeSpace and VariationalFormulations.edp files produced by PSD_PreProcess step.

4.2 Damage mechanics

4.2.1 Hybrid phase-field for damage

On a meshed domain $\Omega^h \in \Omega \subset \mathbb{R}^n$, for damage mechanics the mixed finite element variational formulation in the Lagrangian framework for searching the unknown nodal displacements vector $\mathbf{u}^h = [u_1, u_2, u_3]^\mathsf{T}$ reads,

$$\begin{aligned} &\text{search } \boldsymbol{u}^h \in \mathbb{V}^h \text{ that satisfies } \forall\, t \in [0,T]: \\ &\int_{\Omega^h} \left[(1-d^h)^2 + \kappa\right] \boldsymbol{\sigma}(\boldsymbol{u}^h) : \boldsymbol{\varepsilon}(\boldsymbol{v}^h)\, \mathrm{dv} = \int_{\partial\Omega^h_\mathrm{N}} \overline{\mathbf{t}} \cdot \boldsymbol{v}^h\, \mathrm{ds} \quad \forall\, \mathbf{v}^h \in \mathbb{V}^h, \end{aligned} \tag{4.3}$$

where $\kappa \ll 1$ is a model parameter to prevent numerical singularity when $d \to 1$. In this formulation, the notation ":" is used for the double contraction between tensors (i.e., component-wise tensor product) and $\mathbb{V}^h$ is a mixed third order vector valued finite element functional space to approximate vector test function $\boldsymbol{v}^h$ and vector trial function $\boldsymbol{u}^h$:

$$\mathbb{V}^h = \left\{ \mathbf{u}^h \in [H^1(\Omega^h)]^3 \;\; \forall t \in [0,T] \mid \forall \mathbf{x} \in \partial\Omega^h_\mathrm{D} \;\; \boldsymbol{u}^h = \overline{\mathbf{u}} \right\}, \tag{4.4}$$

with $H^1(\Omega^h)$ denoting a square integrable Sobolev functional space. Similarly, for the phase-field the standard finite element variational formulation for the unknown damage scalar d^h reads,

$$\begin{aligned} &\text{search } d^h \in V^h \text{ that satisfies } \forall\, t \in [0,T]: \\ &\int_{\Omega^h} \left[\frac{G_\mathrm{c}}{l_0} + 2\mathcal{H}^+(\boldsymbol{u}^h)\right] d^h \theta^h\, \mathrm{dv} + \int_{\Omega^h} G_\mathrm{c} l_0 \nabla d^h \cdot \nabla \theta^h\, \mathrm{dv} = \int_{\Omega^h} 2\mathcal{H}^+(\boldsymbol{u}^h)\theta^h\, \mathrm{dv} \quad \forall\, \theta^h \in V^h, \end{aligned} \tag{4.5}$$

where, V^h denotes the scalar finite element functional space to approximate scalar test function θ^h and scalar trial function d^h:

$$V^h = \left\{ d^h \in H^1(\Omega^h) \;\; \forall t \in [0,T] \;\middle|\; d^h \in [0,1] \right\}. \tag{4.6}$$

4.2.2 Tensile cracking of a pre-cracked plate: A 2D example of PSD parallel solver

A two dimensional test is introduced. The problem of interest is the typical single notch square plate cracking test under tensile loading. A unit square with a pre existing crack is clamped at the bottom $u_1 = u_2 = 0$ (first boundary condition) and is loaded quasi-statically $u_2 = u_2 + \Delta u_2$ on its top surface till the crack propagates through its walls. So there are two Dirichlet conditions one on the top border and one on the bottom one.

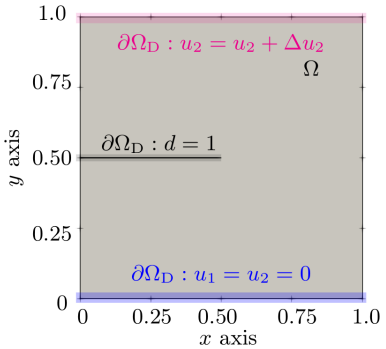


Figure 4.17: Domain of the single notch square cracking problem under tensile loading.

To model this test PSD provides hybrid phase-field modelling technique. We use ParaView post-processing of displacement u and phase-field d to visualise the cracking process. A PSD simulation is a two step process, with step one being the PSD_PreProcess :

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid_phase_field \
-dirichletconditions 2 -postprocess ud
```

A note on flags.

- This is a two-dimensional problem, so we use the flag -dimension 2.
- This problem indeed falls under the category of damage-mechanics, hence the flag -problem damage.
- We wish to solve this problem by invoking the hybrid phase-field problem, which is signified by the flag -model hybrid_phase_field.
- Versed in the description above the problem contains two Dirichlet conditions, we signal this via the flag -dirichletconditions 2.
- Finally for this problem we use the flag -postprocess ud which enables post-processing of displacement u and damage (phase-field) d fields.

Once the step above has been performed, we solve the problem using four MPI processes, with the given mesh file tensile-crack.msh. This is step two of the PSD simulation PSD_Solve.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/tensile-crack.msh -v 0
```

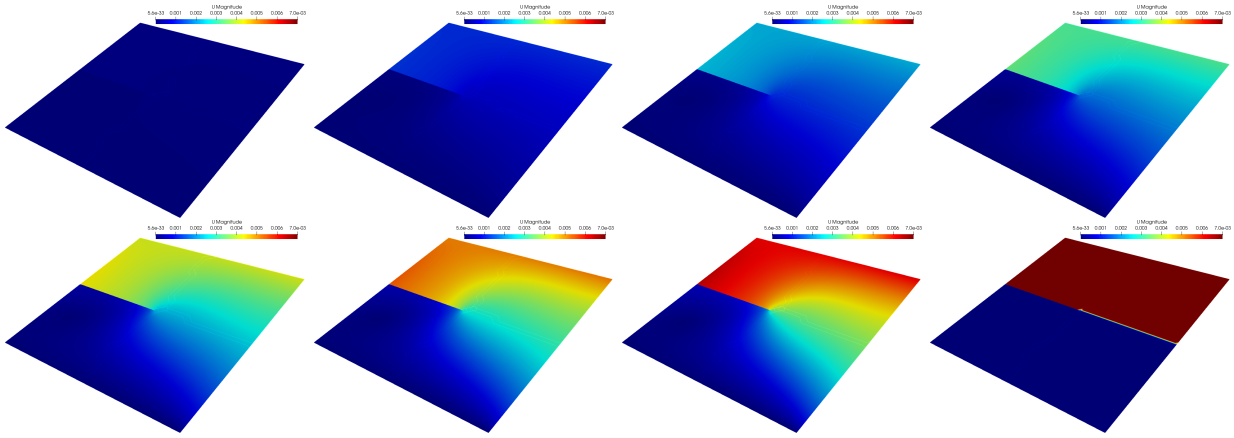


Figure 4.18: Finite element displacement visualised for the 2D problem with ParaView at different timesteps (quasi-statics). Time progresses from left to right in a row and top to bottom when comparing rows.

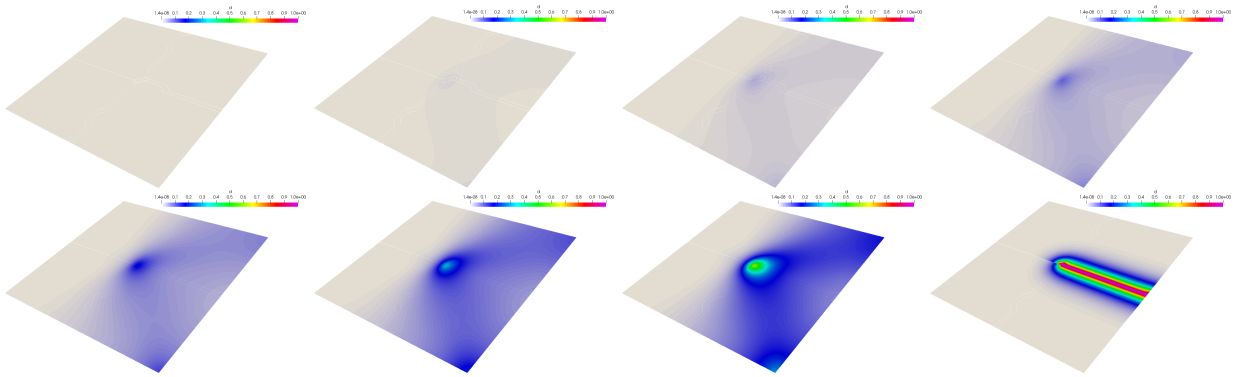


Figure 4.19: Finite element damage visualised for the 2D problem with ParaView at different timesteps (quasi-statics). Time progresses from left to right in a row and top to bottom when comparing rows.

Figures 4.18 and 4.19 present the finite element displacement and damage field, which enable us to visualise the cracking of the square plate.

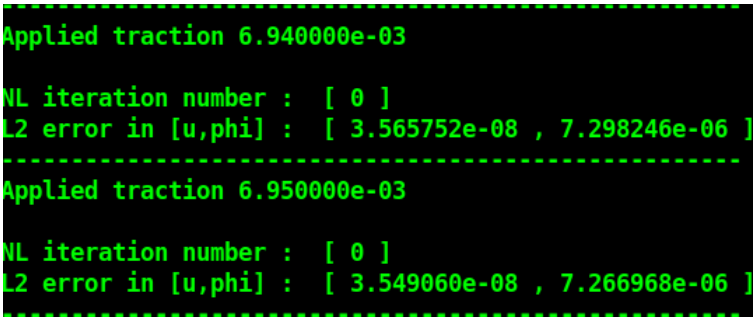


Figure 4.20: Applied traction, non-linear iterations to convergence, and residual being casted onto the terminal shell.

While this test runs, you will see on your screen the amount of traction updated, non-linear iterations taken to converge per-quasi-time-step and residue of u and d. See figure 4.20 that shows the screenshot of the terminal while the test was running. In order to construct your own test case try editing the ControlParameters.edp file.

4.2.3 Tensile cracking of a pre-cracked cube: A 3D example of PSD parallel solver

A three-dimensional test synonymous to its two-dimensional counterpart introduced above is used here as an tutorial example. The problem of interest is now a unit extrusion (along z-axis) of the 2D case above. Cracking is initiated and propagated under tensile loading. The unit cube with its pre existing crack is clamped at the bottom $u_1 = u_2 = u_3 = 0$ (first boundary condition) and is loaded quasi-statically $u_2 = u_2 + \Delta u_2$ on its top surface till the crack propagates through its walls. So there are two Dirichlet conditions one on the top border and one on the bottom one.

Just like in the 2D case, to model this test PSD's' hybrid phase-field modelling technique is used. We will again use ParaView post-processing of displacement u and phase-field d to visualise the cracking process. A PSD simulation is a two step process, with step one being the PSD_PreProcess :

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid_phase_field  \
-dirichletconditions 2 -postprocess ud
```

Notice that the flags used here are almost similar except for the -dimension 3 flag, which indeed specifies three-dimensional problem.

Once the step above has been performed, we solve the problem using four MPI processes, with the given mesh file tensile-crack.msh. This is step two of the PSD simulation PSD_Solve.

```
PSD_Solve -np 3 Main.edp -mesh ./../Meshes/3D/tensile-crack.msh -v 0
```

Figures 4.21 and 4.22 present the finite element displacement and damage field of the 3D problem, which enable us to visualise the cracking of the cubic specimen.

4.2.4 Parallel 3D and calculate reactionforce

Parallel 2D tensile cracking and calculate-ploting reaction-force

In solid mechanics often the quantities of interest includes plots such as reaction-force on a surface vs. the applied force. Often times these are experimental outputs and are used for validation.

PSD provides routines to calculate the reaction force on a surface and also provides means of live plotting (run-time) of these results. Imagine the test case of tensile cracking of plate (2D) as discussed above.

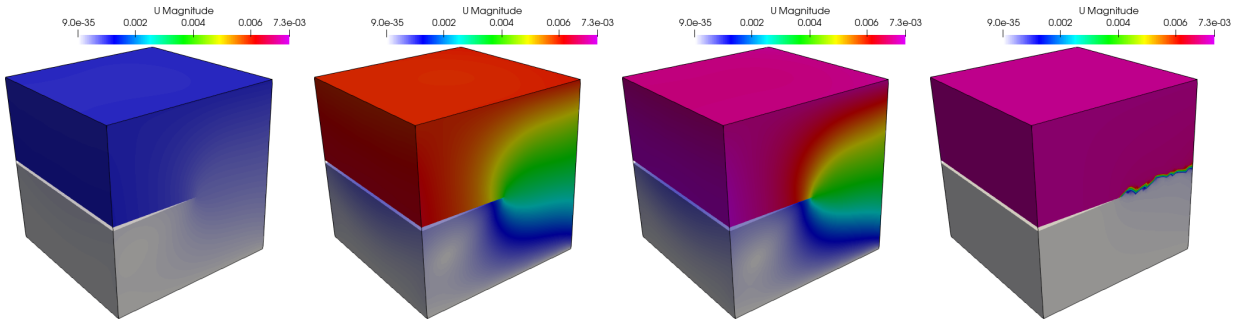


Figure 4.21: Finite element displacement visualised for the 3D problem with ParaView at different timesteps (quasi-statics). Time progresses from left to right in a row and top to bottom when comparing rows.

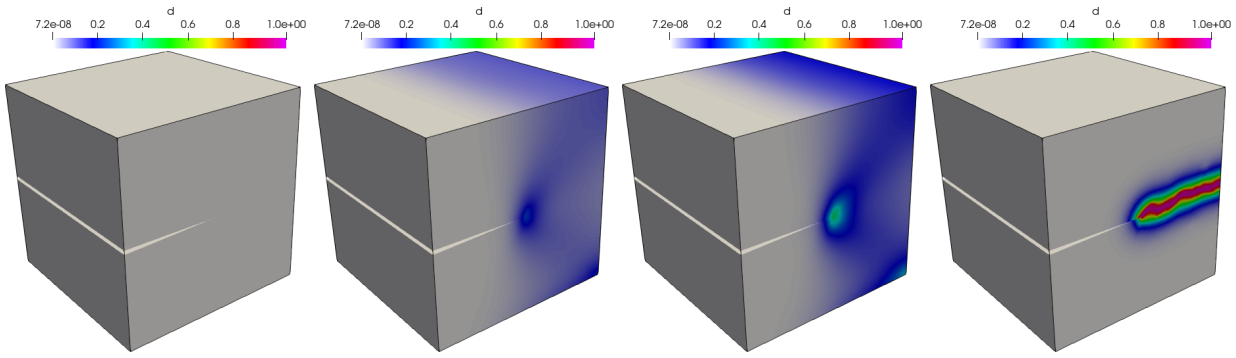


Figure 4.22: Finite element damage visualised for the 3D problem with ParaView at different timesteps (quasi-statics). Time progresses from left to right in a row and top to bottom when comparing rows.

Considering we are now interested in seeing the plot of reaction force at surface vs. the applied tensile displacement, we would need to use two extra flags in the PSD_PreProcess step. These flags are -getreactionforce and -reactionforce stress_based as read below:

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid_phase_field \
-dirichletconditions 2 -getreactionforce -reactionforce stress_based
```

The flag -getreactionforce directs PSD to include the routines to get the reaction force and -reactionforce stress_based is the method by which we get reaction force, in this case reaction force is calculated using integral of stress in y direction $F_y = \int_{\partial\Omega_{top}} \sigma_y$. Other method -reactionforce variational_based also exists within PSD, which is more accurate but slower, this method calculates reaction force based on matrix vector multiplication $F_x, F_y = \mathbf{A}u_1, u_2$.

Run the problem in the usual way bu using PSD_Solve and appropriate number of processes and mesh. While the PSD solver runs it will create a file force.data that contains the reaction force and the applied traction.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/tensile-crack.msh -v 0
```

You can then go ahead and plot force.data to see how F_y and F_x evolve with Δu_2. Within the file the first column is the loading Δu_2, the second and the third columns are the forces F_x and F_y.

Optionally if you have GnuPlot configured with PSD you can see live ploting of this curve if you use option -plotreactionforce during the PSD_PreProcess.

Parallel 3D and calculate reactionforce

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid_phase_field \
-dirichletconditions 2 -getreactionforce -reactionforce stress_based
```

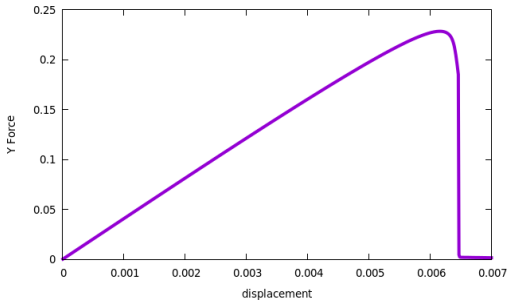


Figure 4.23: Applied traction vs. force in y direction.

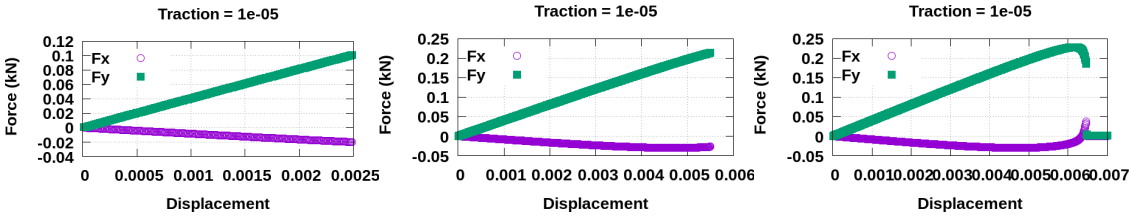


Figure 4.24: Applied traction vs. force in y direction plotted live using PSD.

```
PSD_Solve -np 3 Main.edp -mesh ./../Meshes/3D/tensile-crack.msh -v 0
```

4.2.5 L-shape cracking with point loading

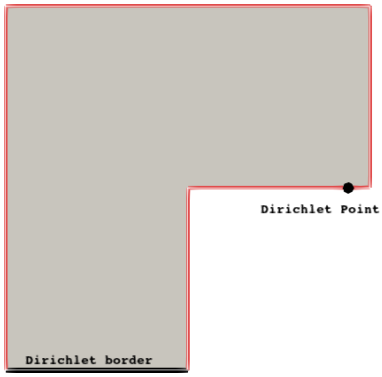


Figure 4.25: Geometry of the L-shaped test used in this tutorial.

Preprocessing

You can either solver the problem using vectorial approach (recommended) or using staggered approach. To generate the solver use either from below.

Generation of solver (vectorial)

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid_phase_field \
-dirichletconditions 1 -dirichletpointconditions 1 -debug -postprocess ud \
-energydecomp -constrainHPF -vectorial -getreactionforce -plotreactionforce \
-reactionforce variational_based
```

Generating solver (staggered)

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid_phase_field \
-dirichletconditions 1 -dirichletpointconditions 1 -debug -postprocess ud \
-energydecomp -constrainHPF -getreactionforce -plotreactionforce \
-reactionforce variational_based
```

Edit Cycle

Edit ControlParameter.edp:

- Update physical parameter, change

```
real lambda = 121.15e3 ,
    mu     = 80.77e3  ,
    Gc     = 2.7      ;
```

to

```
real lambda = 6.16e3 ,
    mu     = 10.95e3 ,
    Gc     = 8.9e-2  ;
```

- Update solver parameter , change

```
real lfac  = 2.0  ,
    maxtr = 7e-3 ,
    tr    = 1e-5 ,
    dtr   = 1e-5 ,
    lo          ;
```

to

```
real lfac  = 2.0  ,
    maxtr = 1    ,
    tr    = 1e-2 ,
    dtr   = 1e-2 ,
    lo          ;
```

- Enter the correct Point boundary condition, change

```
 real[int,int] PbcCord = [
//-------------------- [  x   , y   ] --------------------//
                [ 0. , 0. ]   // point 0
//--------------------------------------------------------//
               ];

 macro Pbc0Ux  -0. //
 macro Pbc0Uy  -0. //
```

to

```
 real[int,int] PbcCord = [
//-------------------- [  x   , y   ] --------------------//
                [ 470., 250. ]   // point 0
//--------------------------------------------------------//
               ]
;
 macro Pbc0Uy  tr //
```

Edit LinearFormBuilderAndSolver.edp:

- To postprocess correct reaction forces in LinearFormBuilderAndSolver.edp for vectorial solver, change

```
for(int i=0; i < Th.nv; i++){
  if(abs(Th(i).y-1.)<.000001){
    forcetotx = forcetotx + F[][i*3]*DP[i*3];
    forcetoty = forcetoty + F[][i*3+1]*DP[i*3+1];
  }
}
```

to

```
if(mpirank==mpirankPCi[0]){
  forcetotx = forcetotx + F[][PCi[0]*3+0]*DP[PCi[0]*3+0];
  forcetoty = forcetoty + F[][PCi[0]*3+1]*DP[PCi[0]*3+1];
}
```

- To postprocess correct reaction forces in LinearFormBuilderAndSolver.edp for staggered solver, change

```
for(int i=0; i < Th.nv; i++){
  if(abs(Th(i).y-1.)<.000001){
    forcetotx = forcetotx + F[][i*2]*DP[i*2];
    forcetoty = forcetoty + F[][i*2+1]*DP[i*2+1];
  }
}
```

to

```
if(mpirank==mpirankPCi[0]){
  forcetotx = forcetotx + F[][PCi[0]*2+0]*DP[PCi[0]*2+0];
  forcetoty = forcetoty + F[][PCi[0]*2+1]*DP[PCi[0]*2+1];
}
```

- Finally to include cyclic loading, change

```
//-----------------updating traction----------------//

tr += dtr;
```

to

```
//-----------------updating traction----------------//

if(iterout<50)
  tr += dtr;
if(iterout>=51 && iterout<110)
  tr -= dtr;
if(iterout>=111)
  tr += dtr;
```

Solving

Irrespective of weather vectorial or staggered mode is used solve the problem using PSD_Solve

```
PSD_Solve -np 4 Main.edp -wg -v 0 -mesh ./../Meshes/2D/L-shaped-crack.msh
```

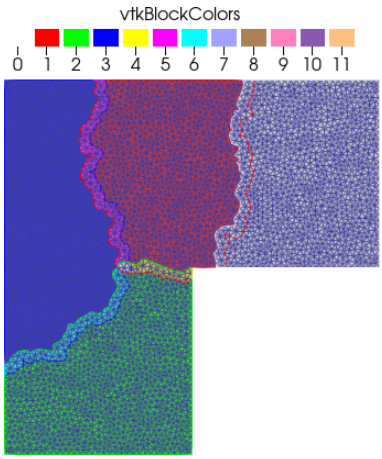


Figure 4.26: Finite element mesh of the L-shaped test.

Postprocessing

Use ParaView to post process results.

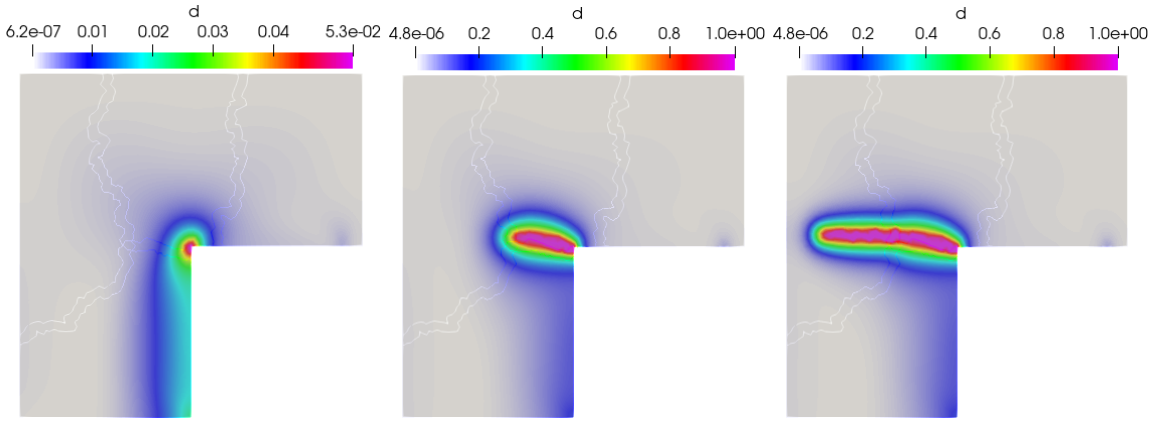


Figure 4.27: Finite element solution showing: Crack initiation, movement, and development.

On you screen, the force displacement curve which plots force.data should look something like this

4.2.6 Additional exercises on damage

Exercise 1

When calculating the reaction force produced on a surface, optionally try changing -reactionforce stress_based to -reactionforce variational_based for changing the method to extract reaction force, note that stress based method is way faster.

Exercise 2

Optionally try using -useGFP flag with PSD_PreProcess optimized solver. GFP acrynom for GoFast Plugins is a suite of C++ based fuctions built for PSD that are speed optimal.

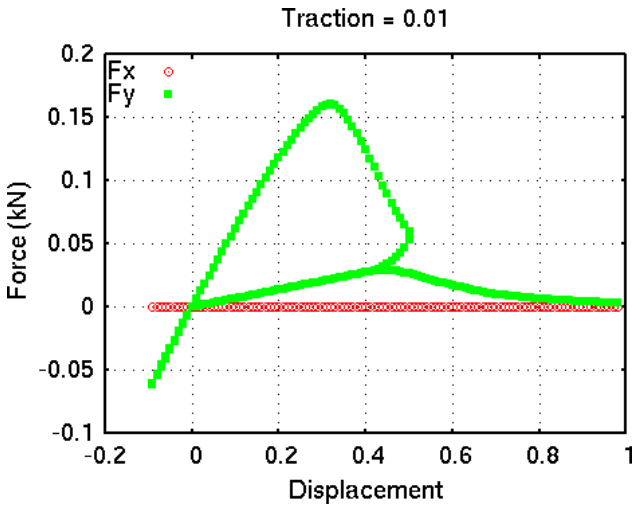


Figure 4.28: Force-displacement curve with cyclic loading.

Exercise 3

Add -sequential flag to PSD_PreProcess for sequential solver, but remember to use PSD_Solve_Seq instead of PSD_Solve

Advanced Exercise 1

try the -vectorial flag for vectorial finite element method

Advanced Exercise 2

try the -energydecomp flag for using split of tensile energy

Advanced Exercise 3

try using -constrainHPF flag for using the constrain condition in hybrid phase field model

4.3 Elastodynamics

4.3.1 Parallel 2D

The problem of interest is a single Dirichlet condition (clamped end bar) and traction loading. For this example we use Newmark-β time discretization. Additionally postrocessing is demanded for displacement, acceleration, and velocity (u, a, v).

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization newmark_beta -postprocess uav
```

Once the step above has been performed, we solve the problem using two MPI processes, with the given mesh file bar-dynamic.msh.

```
PSD_Solve -np 2 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0
```

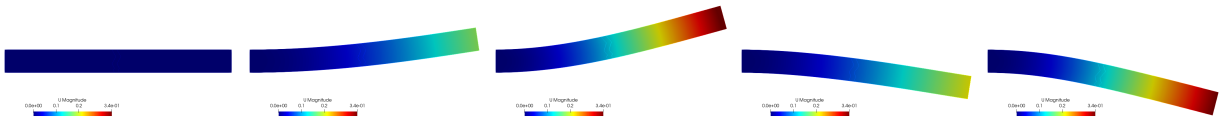


Figure 4.29: Finite element displacement field on warped mesh shown at different time steps.

Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.29 can be extracted.

4.3.2 Parallel 3D

The problem of interest is a single Dirichlet condition (clamped end bar) and traction loading. For this example we use Newmark-β time discretization. Additionally postrocessing is demanded for displacement, acceleration, and velocity (u, a, v).

```
PSD_PreProcess -dimension 3 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization newmark_beta
```

Once the step above has been performed, we solve the problem using four MPI processes, with the given mesh file bar-dynamic.msh.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/3D/bar-dynamic.msh -v 0
```

4.3.3 Sequential problems

To the same problems above Add -sequential flag to PSD_PreProcess for sequential solver, but remember to use PSD_Solve_Seq instead of PSD_Solve. So the work flow for the 2D problem would be:

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization newmark_beta -postprocess uav -sequential
```

Once the step above has been performed, we solve the problem using the given mesh file bar-dynamic.msh.

```
PSD_Solve_Seq -np 2 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0
```

Similarly try out the 3D problem as well.

4.3.4 Different time discretization

PSD offers different time discretization techniques for solving time dependent problems. For this example instead of using Newmark-β time discretization let us switch to more advanced Generalized-α one. This can be done by -timediscretization generalized_alpha, so for example for a 2D problem we use:

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization generalized_alpha -postprocess uav
```

Once the step above has been performed, we solve the problem using three MPI processes, with the given mesh file bar-dynamic.msh.

```
PSD_Solve -np 3 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0
```

Similarly try out the 3D problem as well.

4.3.5 Comparing CPU time

PSD provides mean to time log your solver via -timelog flag. What this will do when you run your solver, on the terminal you will have information printed on what is the amount of time taken by each step of your solver. Warning, this will make your solver slower, as this action involves MPIbarrier routines for correctly timing operation.

An example work flow of 2D solver with timelogging:

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization newmark_beta -postprocess uav -timelog
```

Once the step above has been performed, we solve the problem using two MPI processes, with the given mesh file bar-dynamic.msh.

```
PSD_Solve -np 2 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0
```

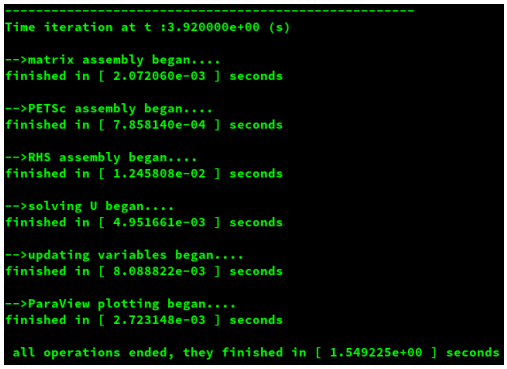


Figure 4.30: Time logging output produced for parallel run on 2 processes.

The figure~4.30 shows the time logging output produced for parallel run on 2 processes using -timelog flag. Similar output is produced for sequential solver of the same problem shown in figure~4.31. Take note of the speed up, which should be two folds - parallel solver solves the full problem in half the time (1.5 sec) than that of sequential solver (3.3 sec). This is due to the fact we used 2 MPI processes.

Also take note of timings produced for different operations of the solver. Note that in figures~4.30, 4.31, we only see the final time step of the solved problem.

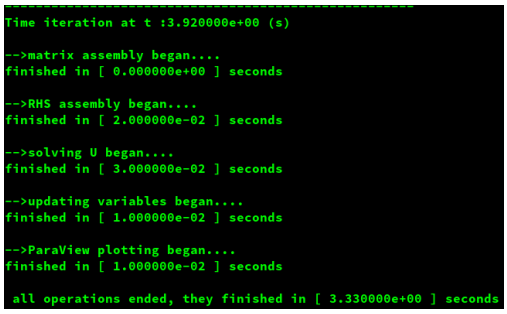


Figure 4.31: Time logging output produced for parallel run on 2 processes.

4.3.6 Additional exercises on damage

Exercise 1

You are encouraged to try out timelogging and find out if the code (parallel/sequential) is any faster when we use Newmark-β or Generalized-α. Read the documentation for other types of time discretizations that can be performed with PSD, try each one out with -timelog and compare.

Exercise 2

There is a solver run level flag for mesh refinement [6]. This flag is called -split [int] which splits the triangles (resp. tetrahedrons) of your mesh into four smaller triangles (resp. tetrahedrons). As such -split 2 will produce a mesh with 4 times the elements of the input mesh. Similarly, -split n where n is a positive integer produces 2^n times more elements than the input mesh. You are encouraged to use this -split flag to produce refined meshes and check, mesh convergence of a problem, computational time, etc. Use of parallel computing is recommended. You could try it out with PSD_Solve or PSD_Solve_Seq, for example:

```
PSD_Solve -np 2 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0 -split 2
```

for splitting each triangle of the mesh bar-dynamic.msh into 4.

Exercise 3

There is a preprocess level flag -debug, which as the name suggests should be used for debug proposes by developers. However, this flag will activate OpebGL live plotting of the problem, with displaced mesh. You are encouraged to try it out

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization newmark_beta -postprocess uav -timelog -debug
```

Then to run the problem we need aditional -wg flag

```
PSD_Solve -np 2 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0 -wg
```

Exercise 4

PSD comes with additional set of plugins/functions that are highly optimized for performing certain operations during solving. These operations are handled by GoFast Plugins (GFP) kernel of PSD (optimize C++ classes/templates/structures), by default this functionality is turned off and not used. You are encouraged

[6] Mesh refinement is performed after partitioning.

to try out using GFP functions in a solver by using -useGFP flag flag to PSD_PreProcess For example, the PSD solver workflow for the first 2D example in this tutorial would be:

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1 \
-timediscretization newmark_beta -postprocess uav -useGFP
```

Once the step above has been performed, we solve the problem using, with the given mesh file bar-dynamic.

```
PSD_Solve -np 2 Main.edp -mesh ./../Meshes/2D/bar-dynamic.msh -v 0 -wg
```

Try it out for other problems of this tutorial. -useGFP should lead to a faster solver, it might be a good idea to always use this option. To go one step further, use -timelog flag and determine if you have some speed up.

4.4 Soil dynamics

The problem of interest is a single Dirichlet condition problem of soildynamics in 2D. For this problem we use Newmark-β time discretization. Additionally postrocessing is demanded for displacement, acceleration, and velocity (u, a, v).

```
PSD_PreProcess -dimension 2 -problem soildynamics -dirichletconditions 1 -timediscretization newmark_beta \
-postprocess uav
```

Once the step above has been performed, we solve the problem using four MPI processes, with the given mesh file soil.msh.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/soil.msh -v 0
```

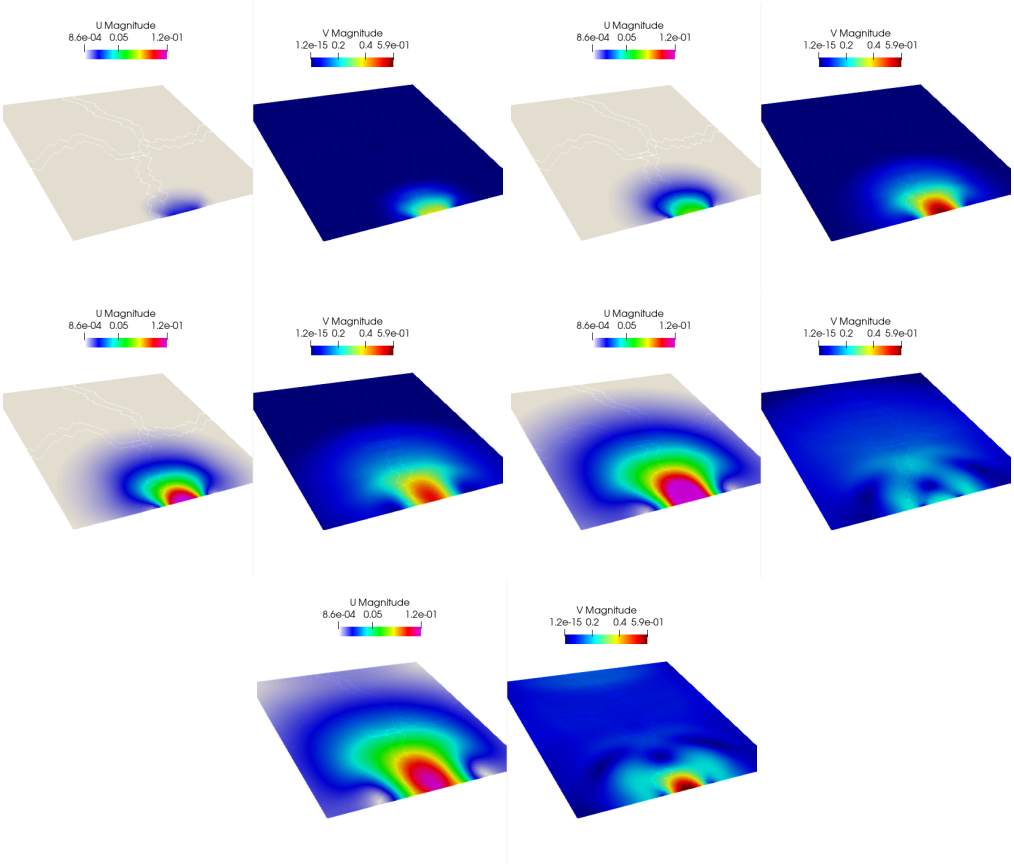


Figure 4.32: Finite element displacement and velocity fields visualized for the 2D problem with ParaView at different timesteps.

Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.32 can be extracted.

4.4.1 Parallel 3D

The problem of interest is a single Dirichlet condition problem of soildynamics in 3D. For this problem we use Newmark-β time discretization. Additionally postrocessing is demanded for displacement, acceleration, and velocity (u, a, v).

```
PSD_PreProcess -dimension 3 -problem soildynamics -dirichletconditions 1 -timediscretization newmark_beta \
-postprocess uav
```

Once the step above has been performed, we solve the problem using three MPI processes, with the given mesh file soil.msh.

```
PSD_Solve -np 3 Main.edp -mesh ./../Meshes/3D/soil.msh -v 0
```

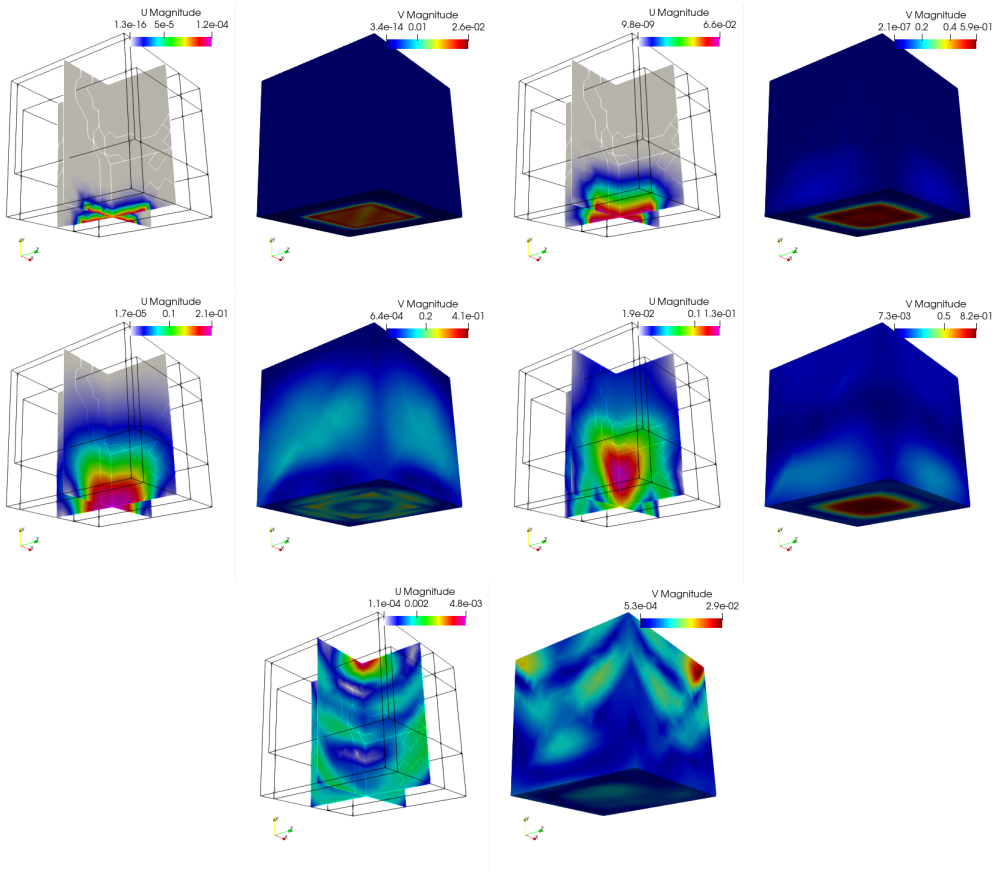


Figure 4.33: Finite element displacement and velocity fields visualized for the 3D problem with ParaView at different timesteps.

Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.33 can be extracted.

4.4.2 Parallel 2D with double couple

In the 2D problem above seismic sources was supplied on the border, in the current one the source is more realistic and comes from a double couple (point Dirichlet condition). The double couple boundary condition is a way to impose moments caused by faults that create earthquakes, here in this problem double couple is imposed using displacement based.

```
PSD_PreProcess -dimension 2 -problem soildynamics -model linear -timediscretization newmark-beta \
-useGFP -doublecouple displacement-based -postprocess uav
```

Once the step above has been performed, we solve the problem using two MPI processes, with the given mesh file soil-dc.msh.

```
PSD_Solve -np 2 Main.edp -v 1 -ns -nw -mesh ./../Meshes/2D/soil-dc.msh
```

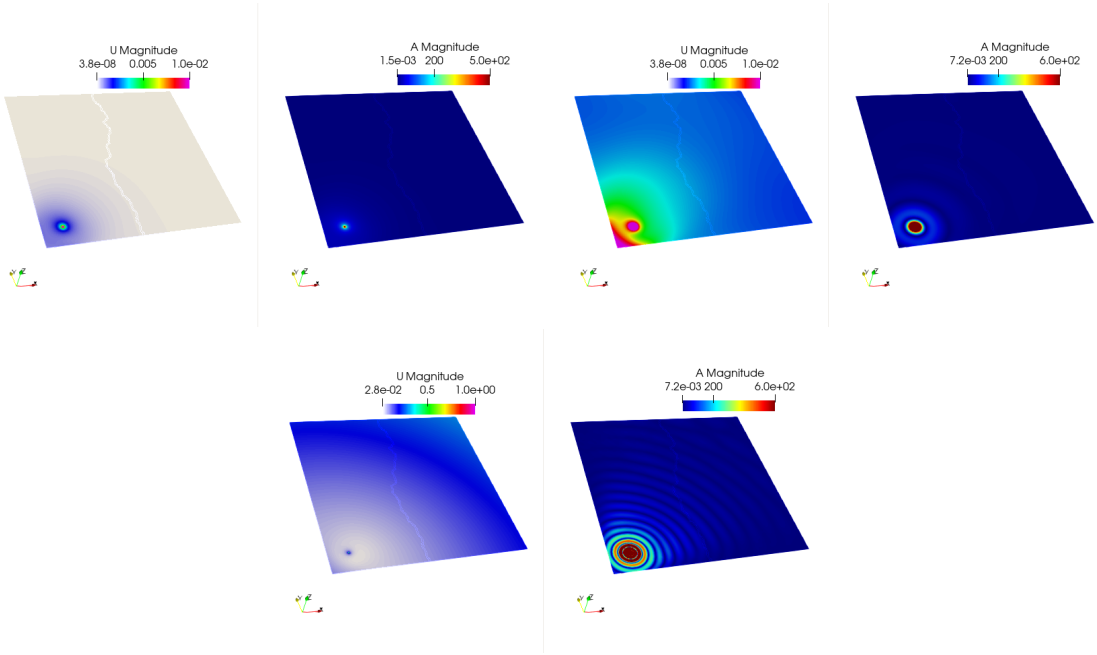


Figure 4.34: Finite element displacement and acceleration fields visualized for the 2D problem with ParaView at different timesteps.

Using ParaView for postprocessing the results that are provided in the VTUs... folder, results such as those shown in figure~4.34 can be extracted.

Similarly try out the 3D problem. However take note that a the mesh ./../Meshes/2D/soil-dc.msh is not provided, so you will have to create your own mesh.

4.4.3 Parallel 3D with top-ii-vol meshing

Single Dirichlet at the bottom and using GFP.

```
PSD_PreProcess -dimension 3 -problem soildynamics -model linear -timediscretization newmark_beta \
-useGFP -top2vol-meshing -timediscretization newmark-beta -postprocess uav
```

```
PSD_Solve -np 4 Main.edp -v 0 -ns -nw
```

4.4.4 Parallel 3D with top-ii-vol meshing and double couple source

Single Dirichlet via double couple and using GFP. Double couple is displacement based.

```
PSD_PreProcess -dimension 3 -problem soildynamics -model linear -timediscretization newmark_beta \
-useGFP -top2vol-meshing -doublecouple displacement-based -postprocess uav
```

```
PSD_Solve -np 3 Main.edp -v 0 -ns -nw
```

4.4.5 Additional exercises on soildynamics

Exercise 1

You are encouraged to try out sequential PSD solver, to do so used add -sequential flag to PSD_PreProcess step and run the solver with PSD_Solve_Seq instead of PSD_Solve. For example, the PSD sequential solver workflow for the first 2D example in this tutorial would be:

```
PSD_PreProcess -dimension 2 -problem soildynamics -dirichletconditions 1 -timediscretization newmark_beta \
-postprocess uav -sequential
```

Once the step above has been performed, we solve the problem using PSD_Solve_Seq, with the given mesh file soil.msh.

```
PSD_Solve_Seq  Main.edp -mesh ./../Meshes/2D/soil.msh -v 0
```

Try it out for other problems of this tutorial.

Exercise 2

For soildynamic problems with double couple source, the double couple source can be introduced into the solver either by displacement-based operator – providing displacements at the double couple points that will be converted to moments – or by force-based operators – providing forces at the double couple points that will be converted to moments. In the tutorials above we already tried displacement-based way of introducing double couple source by using -doublecouple displacement_based. You are encouraged to try out the force-based double couple source by using -doublecouple force_based.

Exercise 3

You are encouraged to try out timelogging and find out if the code (parallel/sequential) is any faster when we use Newmark-β or Generalized-α. Read the documentation for other types of time discretizations that can be performed with PSD, try each one out with -timelog and compare.

Exercise 4

PSD comes with additional set of plugins/functions that are highly optimized for performing certain operations during solving. These operations are handled by GoFast Plugins (GFP) kernel of PSD (optimize C++ classes/templates/structures), by default this functionality is turned off and not used. You are encouraged to try out using GFP functions in a solver by using -useGFP flag flag to PSD_PreProcess For example, the PSD solver workflow for the first 2D example in this tutorial would be:

```
PSD_PreProcess -dimension 2 -problem soildynamics -dirichletconditions 1 -timediscretization newmark\_beta \
-postprocess uav -useGFP
```

Once the step above has been performed, we solve the problem using, with the given mesh file soil.msh.

```
PSD_Solve -np 4 Main.edp -mesh ./../Meshes/2D/soil.msh -v 0
```

Try it out for other problems of this tutorial. -useGFP should lead to a faster solver, it might be a good idea to always use this option. To go one step further, use -timelog flag and determine if you have some speed up.

4.5 General list of examples: Linear Elasticity

==========

Sequential 2D linear-elasticity

==========

```
PSD_PreProcess -dimension 2 -bodyforceconditions 1 conditions 1 -sequential -dirichletconditions 1

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Sequential 3D linear-elasticity

==========

```
PSD_PreProcess -dimension 3 -bodyforceconditions 1 -sequential -dirichletconditions 1

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Sequential 2D linear-elasticity fastmethod

==========

```
PSD_PreProcess -dimension 2 -bodyforceconditions 1 -sequential -dirichletconditions 1 -fastmethod

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Sequential 3D linear-elasticity fastmethod

==========

```
PSD_PreProcess -dimension 3 -bodyforceconditions 1 -sequential -dirichletconditions 1 -fastmethod

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Parallel 2D linear-elasticity

==========

```
PSD_PreProcess -dimension 2 -bodyforceconditions 1  -dirichletconditions 1

ff-mpirun-np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D linear-elasticity

*==========

```
PSD_PreProcess -dimension 3 -bodyforceconditions 1  -dirichletconditions 1

ff-mpirun-np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D linear-elasticity fastmethod

*==========

```
PSD_PreProcess -dimension 2 -bodyforceconditions 1  -dirichletconditions 1 -fastmethod

ff-mpirun-np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D linear-elasticity fastmethod

*==========

```
PSD_PreProcess -dimension 3 -bodyforceconditions 1  -dirichletconditions 1 -fastmethod

ff-mpirun-np 2  Main.edp -v 0 -ns -nw
```

4.6 General list of examples: Fracture mechanics

*==========

Sequential 2D phase-field fracture mechanics

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -sequential -dirichletconditions 2

PSD_Solve Main.edp -v 0 -ns -nw
```

*==========

Sequential 3D phase-field fracture mechanics

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -sequential -dirichletconditions 2

PSD_Solve Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -dirichletconditions 2

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -dirichletconditions 2

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with vectorial FEM

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with vectorial FEM

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Sequential 2D phase-field fracture mechanics energydecomp

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -sequential -dirichletconditions 2 \
-energydecomp

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

*==========

Sequential 3D phase-field fracture mechanics energydecomp

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -sequential -dirichletconditions 2 \
-energydecomp

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics energydecomp

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -dirichletconditions 2 -energydecomp

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics energydecomp

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -dirichletconditions 2 -energydecomp

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics energydecomp & vectorial

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-energydecomp

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics energydecomp

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-energydecomp

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Sequential 2D phase-field fracture mechanics with GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field  -dirichletconditions 2 \
-sequential -useGFP

PSD_Solve Main.edp -v 0 -ns -nw
```

*==========

Sequential 3D phase-field fracture mechanics with GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field  -dirichletconditions 2 \
-sequential -useGFP

PSD_Solve Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -dirichletconditions 2 -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -dirichletconditions 2 -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with GFP & vectorial

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with GFP & vectorial

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Sequential 2D phase-field fracture mechanics with energydecomp & GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -sequential -dirichletconditions 2 \
-energydecomp -useGFP

PSD_Solve Main.edp -v 0 -ns -nw
```

*==========

Sequential 3D phase-field fracture mechanics with energydecomp & GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -sequential -dirichletconditions 2 \
-energydecomp -useGFP

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with energydecomp & GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -dirichletconditions 2 \
-energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with energydecomp & GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -dirichletconditions 2 \
-energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with energydecomp, vectorial & GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with energydecomp, vectorial & GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with reaction-force, energydecomp, vectorial & GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-getreactionforce -energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with reaction-force, energydecomp, vectorial & GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-getreactionforce -energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 2D phase-field fracture mechanics with live reaction-force plotting, energydecomp, vectorial & GFP

*==========

```
PSD_PreProcess -dimension 2 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-getreactionforce -plotreactionforce -energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

*==========

Parallel 3D phase-field fracture mechanics with live reaction-force plotting, energydecomp, vectorial & GFP

*==========

```
PSD_PreProcess -dimension 3 -problem damage -model hybrid-phase-field -vectorial -dirichletconditions 2 \
-getreactionforce -plotreactionforce -energydecomp -useGFP

PSD_Solve -np 2  Main.edp -v 0 -ns -nw
```

4.7 General list of examples: Elastodynamics

==========

Sequential 2D Elastodynamics

==========

```
PSD_PreProcess -dimension 2 -problem elastodynamics -sequential -dirichletconditions 1 -tractionconditions 1

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Sequential 3D Elastodynamics

==========

```
PSD_PreProcess -dimension 3 -problem elastodynamics -sequential -dirichletconditions 1 -tractionconditions 1

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Parallel 2D Elastodynamics

==========

```
PSD_PreProcess -dimension 2 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1

PSD_Solve  -np 2 Main.edp -v 0 -ns -nw
```

==========

Parallel 3D Elastodynamics

==========

```
PSD_PreProcess -dimension 3 -problem elastodynamics -dirichletconditions 1 -tractionconditions 1

PSD_Solve  -np 2 Main.edp -v 0 -ns -nw
```

4.8 General list of examples: Soildynamics

==========

Sequential 2D Soildynamics

==========

```
PSD_PreProcess -dimension 2 -problem soildynamics -sequential -dirichletconditions 1

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Sequential 3D Soildynamics

==========

```
PSD_PreProcess -dimension 3 -problem soildynamics -sequential -dirichletconditions 1

PSD_Solve_Seq Main.edp -v 0 -ns -nw
```

==========

Parallel 2D Soildynamics

==========

```
PSD_PreProcess -dimension 2 -problem soildynamics -dirichletconditions 1

PSD_Solve -np 2 Main.edp -v 0 -ns -nw
```

==========

Parallel 3D Soildynamics

==========

```
PSD_PreProcess -dimension 3 -problem soildynamics -dirichletconditions 1

PSD_Solve -np 2 Main.edp -v 0 -ns -nw
```

Chapter 5

Validation

5.1 Linear elasticity solver validation and verification using method of manufactured solutions

The main aim of this write-up is to develop and implement the method of manufactured solutions (MMS) within the field of linear elasticity. A FEM solver is developed for solving linear elasticity problems. Within this note MMS is applied to verify and validate this solver.

Mathematical typesetting conventions

Unless specified, the mathematical typesetting conventions are as follows.

Typesetting	Example	Description
Bold uppercase	$\mathbf{A}, \mathbf{M}$	Matrices
Bold lowercase	$\mathbf{x}, \mathbf{b}, \mathbf{f}$	Vector
Lowercase Greek	α, β, λ	Scalars
Lowercase Roman	g	Scalars
Bold Greek	$\boldsymbol{\sigma}, \boldsymbol{\varepsilon}$	Tensor
Blackboard bold style	$\mathbb{R}$	Number set

5.1.1 Introduction

In order to use the developed FEM solver to predict outcomes from previously unforeseen situations in linear elasticity, it is important to validate and verify the proposed solver. In other words, it is important to build trust on the solvers reliability and know its limits. This can be done by asserting whether the solver is able to reproduce analytical or experimental observations for certain linear elasticity problems. Another way is to compare against results of certain benchmark problems solved with other numerical tools, hence performing cross-validation. Other interesting option is the use of MMS.

Before progressing further, let us interpret what validation and verification means in the context of numerical modeling. Assuming that the mathematical model for a given physics is accurate, the process of *verification* investigates if an accurate numerical solution to the given mathematical model can be obtained via the numerical method which is being verified. By the process of verification the order of accuracy for the numerical methods can also be calculated. Whereas, the process of *validation* asserts if an appropriate

mathematical model has been chosen to describe the physical phenomenon. More elaborate discussions on the process of validation and verification of numerical tools can be found in [**oberkampf2004verification**].

5.1.2 Verification tests with the method of manufactured solutions

The method of manufactured solutions is used by many numerical communities for solver (code) verification, see for example [**roache1998verification**, **ecca2007verification**, **pautz2001verification**]. Concerning the solid mechanics solvers, studies such as **???** used the method of manufactured solutions for solver verification.

In the method of manufactured solutions, we start with an assumed explicit expression for the solution field (manufactured solution). Then, the solution is substituted in the concerned PDE model. This leads to a consistent set of source terms and/or initial conditions and/or boundary conditions. These terms are then used to solve the equation numerically, with the method (solver) that needs to be verified. Finally, by analyzing the error between the numerical solution and the manufactured solution, one can verify if the numerical method works. In addition, by analyzing how the error decreases when finer numerical discretization is considered, one can obtain the order of convergence for the numerical method.

Two-dimensional MMS test case

Let us assume a hypothetical solid material domain ($\Omega \in \mathbb{R}^2$) is acted upon by manufactured forcing vector $\widehat{\mathbf{f}} = [\widehat{f}_1, \widehat{f}_2]^\top$. This causes the body to deform:

$$\begin{aligned} \widehat{u}_1 &= x^3 + x^2 y, \\ \widehat{u}_2 &= xy^2 + x^2 y. \end{aligned} \tag{5.1}$$

The equation (5.1) is our manufactured solution and has been explicitly assumed[1]. The task now is to calculate the manufactured forcing vector $\widehat{\mathbf{f}}$. In order to do so the following steps are applied.

- Using (5.1) for calculating $\nabla \cdot \widehat{\mathbf{u}}$:

$$\begin{aligned} \partial_x \widehat{u}_1 &= 3x^2 + 2xy, & \partial_y \widehat{u}_1 &= x^2, \\ \partial_x \widehat{u}_2 &= y^2 + 2xy, & \partial_y \widehat{u}_2 &= x^2 + 2xy, \end{aligned} \tag{5.2}$$

  then

$$\nabla \cdot \widehat{\mathbf{u}} = \nabla \cdot [\widehat{u}_1 \;\; \widehat{u}_2]^\top = 4(x^2 + xy). \tag{5.3}$$

- Using (5.2) for calculating the manufactured stain tensor $\widehat{\boldsymbol{\varepsilon}}$ components $\widehat{\varepsilon}_{ij}$:

$$\widehat{\varepsilon}_{ij}(\widehat{\mathbf{u}}) = \frac{1}{2}\left(\partial_j \widehat{u}_i + \partial_i \widehat{u}_j\right),$$

  then

$$\begin{aligned} \widehat{\varepsilon}_{11} &= 3x^2 + 2xy, & \widehat{\varepsilon}_{12} &= \frac{1}{2}(x^2 + y^2 + 2xy), \\ \widehat{\varepsilon}_{21} &= \frac{1}{2}(x^2 + y^2 + 2xy), & \widehat{\varepsilon}_{22} &= x^2 + 2xy. \end{aligned} \tag{5.4}$$

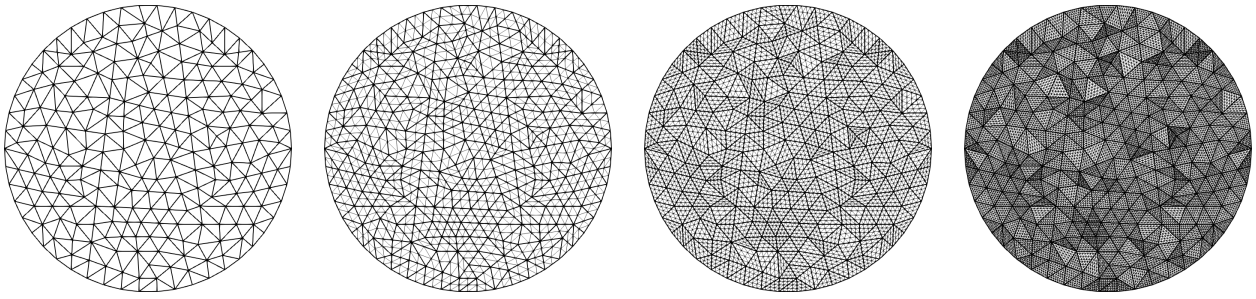
- Using (5.3) and (5.4) for calculating the manufactured stress tensor $\widehat{\boldsymbol{\sigma}}$ components $\widehat{\sigma}_{ij}$:

$$\widehat{\sigma}_{ij} = \lambda \delta_{ij} \nabla \cdot \widehat{\mathbf{u}} + 2\mu \widehat{\varepsilon}_{ij}(\widehat{\mathbf{u}})$$

  then

$$\begin{aligned} \widehat{\sigma}_{11} &= 4\lambda(x^2 + xy) + 2\mu(3x^2 + 2xy), & \widehat{\sigma}_{12} &= \mu(x^2 + y^2 + 2xy), \\ \widehat{\sigma}_{21} &= \mu(x^2 + y^2 + 2xy), & \widehat{\sigma}_{22} &= 4\lambda(x^2 + xy) + 2\mu(3x^2 + 2xy). \end{aligned} \tag{5.5}$$

[1] One could chose other expressions, as long as small strain limiting condition is obeyed: $\|\boldsymbol{\varepsilon}\| = \{0.2 \times 10^{-2}; 0.5 \times 10^{-2}\}$



(a) Level 1 $\Omega^{0.01}$ (b) Level 2 $\Omega^{0.005}$ (c) Level 3 $\Omega^{0.002}$ (d) Level 4 $\Omega^{0.001}$

Figure 5.1: Finite element meshes.

- Finally, using (5.4) for calculating the manufactured force vector $\widehat{\mathbf{f}}$ components $\widehat{f}_i$:

$$\begin{aligned}\widehat{f}_1 &= -\partial_x\widehat{\sigma}_{11} - \partial_x\widehat{\sigma}_{12},\\ \widehat{f}_2 &= -\partial_x\widehat{\sigma}_{21} - \partial_x\widehat{\sigma}_{22},\end{aligned}$$

then

$$\begin{aligned}\widehat{f}_1 &= -x(8\lambda + 14\mu) - y(4\lambda + 6\mu),\\ \widehat{f}_2 &= -x(6\lambda + 4\mu) - y(2\mu).\end{aligned} \tag{5.6}$$

Now all we need is a FEM solver that can solve the elasticity system given the manufactured forcing vector (5.6). Error analysis between the displacement solution vector $\mathbf{u}^h$ from the FEM solver and the manufactured solution $\widehat{\mathbf{u}}$, can then be used to validate and verify the solver. In addition the error analysis will help in assessing the convergence order.

5.1.3 FEM solving model

Assuming Ω^h be the bi-dimensional tessellated reference configuration for the hypothetical solid material, or in other words the finite element mesh defined with size parameter h. Finite element variational formulation in the Lagrangian framework for the unknown displacements vector $\mathbf{u}^h$ then reads,

$$\left\|\begin{aligned}&\text{find } \mathbf{u}^h \in [H_0^1(\Omega^h)]^2 :\\ &\int_{\Omega^h} \lambda\nabla\cdot\mathbf{u}^h\nabla\cdot\mathbf{v}^h + \int_{\Omega^h} 2\mu\boldsymbol{\varepsilon}(\mathbf{u}^h) : \boldsymbol{\varepsilon}(\mathbf{v}^h) + \int_{\Omega^h} \widehat{\mathbf{f}}\cdot\mathbf{v}^h = 0, \quad \forall\mathbf{v}^h \in [H_0^1(\Omega^h)]^2,\\ &\text{given} \quad \mathbf{u}^h = \widehat{\mathbf{u}} \quad \text{on} \quad \partial\Omega_D^h.\end{aligned}\right. \tag{5.7}$$

Notice in this equation the essential Dirichlet boundary conditions[2] are provided by the known manufactured displacement field $\widehat{\mathbf{u}}$ from equation (5.1). Please refer to the other note "The Krylov subspace based CG solver for linear elasticity", in order to know how the finite element linear system $\mathbf{Ax} = \mathbf{b}$ is assembled and then solved iteratively to derive the displacement field $\widehat{\mathbf{u}} : \widehat{\mathbf{u}} = \mathbf{x}^{(m)}$, where m is the converged iteration number for the CG iterative solver.

-	Level 1	Level 2	Level 3	Level 4
N_v	244	923	3,589	14,153
N_e	486	1,844	7,176	28,304
$h_{\min}$	0.0102	0.0051	0.0026	0.00013

Table 5.1: Characteristics of different FEM meshes used for error analysis.

[2] For unique solution of the elasticity problem one needs the Dirichlet boundary conditions.

5.1.4 The FEM solver

To numerically solve equation (5.7), a mixed finite element space based solver is developed using a DSL FreeFem++ [**hecht2012new**]. The space discretization kernel of FreeFem++ uses unstructured (triangular or tetrahedral) mesh inputs. Further, for the linear algebra backend the CG solver provided within FreeFem++. The solver has the capabilities to mix $\mathbb{P}_1$, $\mathbb{P}_2$, and $\mathbb{P}_3$ finite element spaces, for approximating $\mathbf{u}$. For the sake of simplicity we will only use mixed $\mathbb{P}_1$ spaces, i.e. in order to solve (5.7) a mixed finite element space $\boldsymbol{\mathcal{V}}^h := \mathcal{V}^h \times \mathcal{V}^h$ is defined, such that

$$\mathbf{u}^h = [u_1^h, u_2^h]^\top \in \boldsymbol{\mathcal{V}}^h \quad \text{and} \quad \mathbf{v}^h = [v_1^h, v_2^h]^\top \in \boldsymbol{\mathcal{V}}^h.$$

where,

$$\mathcal{V}^h(v^h) = \{v^h \in [H_0^1(\Omega^h)],\ v^h \in \mathbb{P}_1 : v^h = \widehat{u} \text{ on } \partial\Omega_D^h\}.$$

Let us now asses if the developed solver is any good and asses its convergence order. We assume Ω to be a circle[3] of radius 0.1 m and made of a material with modulus of elasticity $E = 100$ GPa and Poissons ratio $\nu = 0.2$. Let us call this test MMS-test 1.

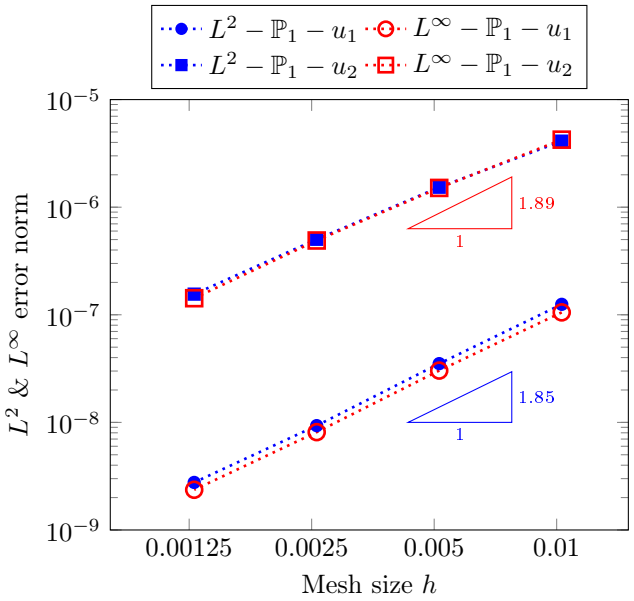


Figure 5.2: Error between the FEM and manufactured solution. Note that the mesh size h is infact the minimum element size $h_{\min}$ that exists within a mesh Ω^h.

To solve this elasticity problem with FEM, four hierarchical meshes are produced using BMAG: Bidimensional Anisotopic Mesh Generator [**hecht1998bamg**]. Hierarchy of refined meshes are obtained by splitting each triangle in the coarse level mesh by four. The split operation is followed to ensure that coarse solutions live in the fine ones. The four hierarchical meshes are presented in figure 5.1. For elaborated characteristics of these meshes refer to table 5.1.

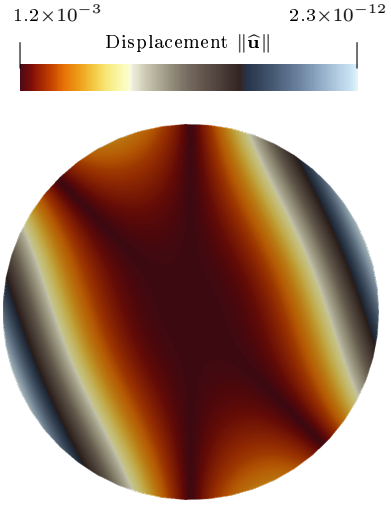
Using the four hierarchical meshes, four simulations of MMS-test 1 were performed. Note, to avoid numerical error discrepancies from the linear solver, the CG iteration was stopped when the relative unpreconditioned residual was lower than 10^{-13}. The mixed FEM solution for the solver was then compared to the manufactured one and L^2 and L^∞ errors were calculated:

$$L^2(\mathbf{u}) = \left(\int_{\Omega^h} (\widehat{\mathbf{u}} - \mathbf{u}^h)^2 \right)^{\frac{1}{2}} \quad \text{and} \tag{5.8}$$

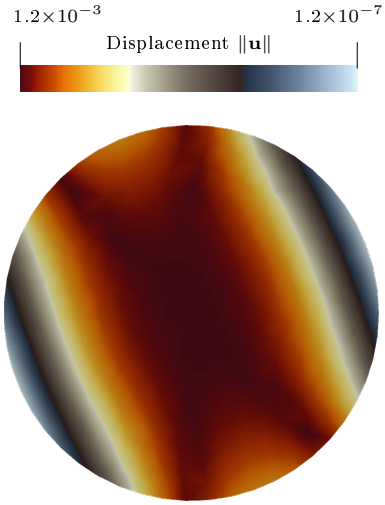
[3]Circular domain is just an assumption, given the mathematics developed in section 5.1.2 one could use any 2D geometry of choice.

$$L^{\infty}(\mathbf{u}) = \max\left(|\widehat{\mathbf{u}} - \mathbf{u}^h|\right), \tag{5.9}$$

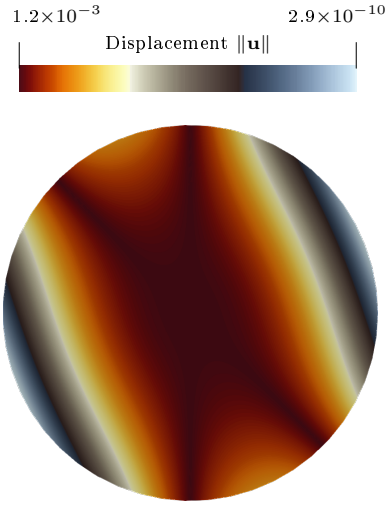
these are plotted in figure 5.2.



(a) MMS solution.



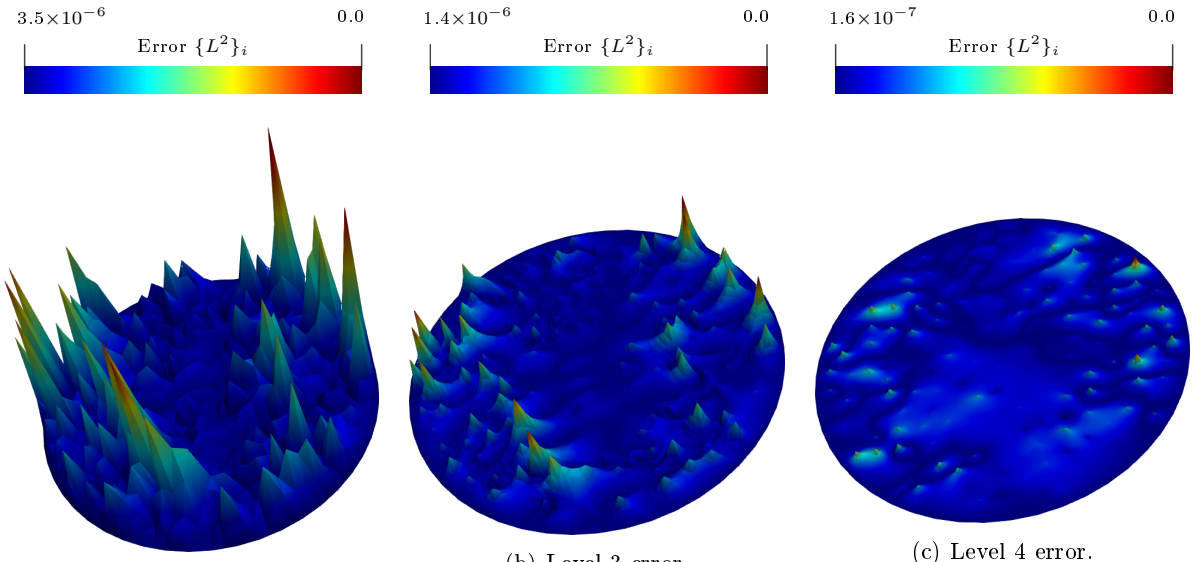
(b) Level 1 FEM solution.



(c) Level 4 FEM solution.

Figure 5.3: Displacement field magnitude visualization.

Error analysis plot provided in figure 5.2 proves that the developed FEM solver has approximately second order convergence rate. More precisely, the L^2 error analysis reveled that the FEM solver has the order of convergences given by 1.85, against theoretical value of 2. Similarly, the L^{∞} error analysis reveled that the FEM solver has the order of convergences given by 1.89, against theoretical value of 2. To investigate further the displacement field magnitudes for the MMS solution and the FEM solutions have been visualized in figure 5.3. Notice how solution improves from the coarsest level mesh (level 1) to the finest one (level 4). It is fair to say that even the coarse level solution is approximating the displacement field well. To investigate further in figure 5.4 the point wise error field is visualized. One can clearly observe how the error reduces when using ore refined meshes. Moreover, notice that error is zero at the borders, this is due to the fact that all border are Dirichlet borders.



(a) Level 2 error.

(b) Level 3 error.

(c) Level 4 error.

Figure 5.4: Warped error field visualization. The warp fields have been magnified a million times for better visualization.

5.2 Damage mechanics solver validation

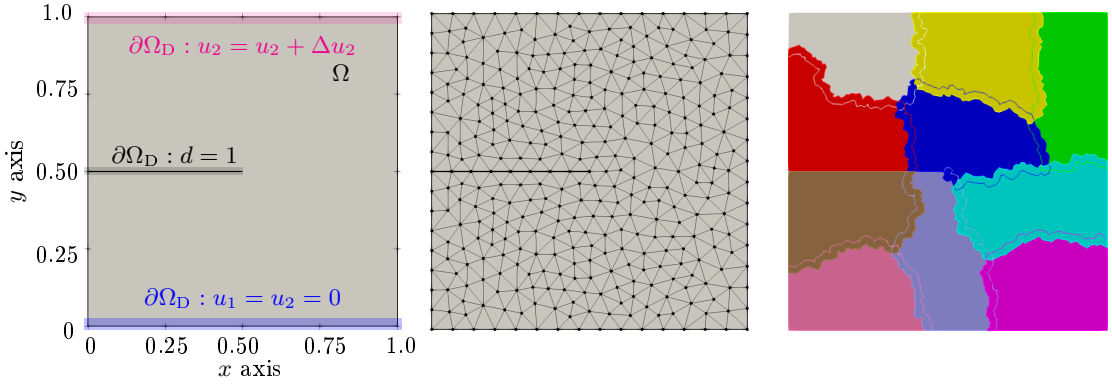
A commonly used numerical test from literature (see e.g., [**Ambati2014**, **Liu2016**, **jeong2018phase**, **Hirshikesh2018**] to cite but a few), the two-dimensional (2D) single-edge notched tensile fracture test, is considered as the benchmark problem in this subsection. From here-forth this test is referenced as test 1 in the text.

Problem setting

The domain of interest is an initially cracked square plate $(x, y) \in \Omega = [0\text{ cm}, 1\text{ cm}]^2$ (fig. 5.5a). With an initial crack and a constrained bottom edge $\partial\Omega_{\text{D}}(x, y : y = 0)$, the plate is subject to increasing displacements on its top edge $\partial\Omega_{\text{D}}(x, y : y = 1)$ until the plate fully cracks open. The initial crack is placed at the center of the plate, i.e., $\partial\Omega_{\text{D}}(x : 0 \leq x \leq 0.5, y : y = 0.5)$. These boundary conditions are also illustrated in fig. 5.5a. The plate material is characterized by $\lambda = 121.15$ kPa, $\mu = 80.77$ kPa, and $G_{\text{c}} = 2.7\ \text{kN}\,\text{mm}^{-1}$.

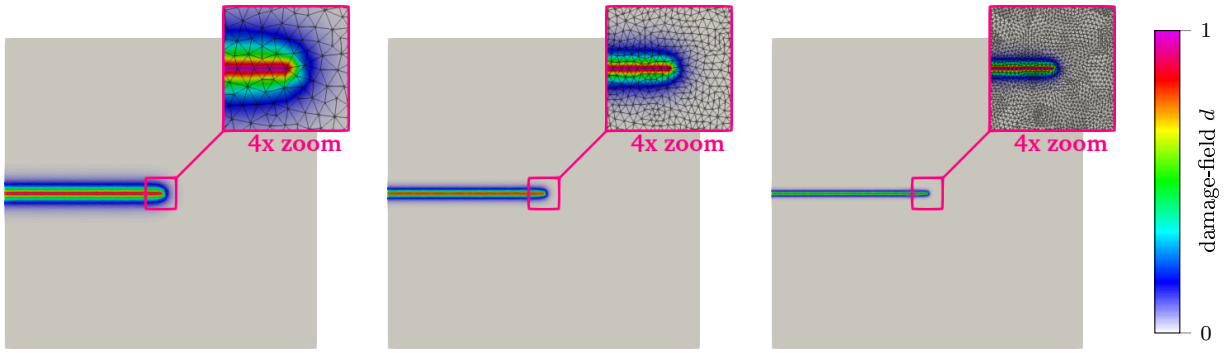
Concerning the computational specifications of test 1, the displacement discontinuity imposed by the initial crack was modeled by nearly overlapping (tolerance $\delta y = 10^{-7}$ m) Dirichlet nodes placed along the cracks edge $\partial\Omega^h_{\text{D}}(x : 0 \leq x \leq 0.5, y : y = 0.5 \pm \delta y)$ within Ω^h. For illustration proposes, a coarse grid featuring Dirichlet nodes for the initial crack of test 1 is presented in fig. 5.5b. The displacement Dirichlet condition on the top edge is applied with an increment of $\Delta\bar{u}_2 = 1 \cdot 10^{-5}$ mm up to $u_2 = 5 \cdot 10^{-3}$ mm and $\Delta\bar{u}_2 = 1 \cdot 10^{-6}$ mm up to failure of the specimen. For the lower edge, the constrained displacement Dirichlet conditions $\bar{u}_1 = \bar{u}_2 = 0$ are applied. Further, for test 1 and for all the simulations that appear in this study, parameter κ is set to $1 \cdot 10^{-6}$ and l_0 is assumed equal to $2h$, where h is the characteristic size of the mesh Ω^h.

The unstructured Delaunay (triangular) meshes generated with Gmsh are used for solving the finite element problem of test 1. To establish mesh convergence, test 1 has been solved multiple times by varying the level of mesh refinements, details of these meshes are provided in table 5.2. The hierarchy of mesh refinements were generated by dividing each triangle in Ω^h into four equal triangles. As such in table 5.2, we observe that with every refinement, the mesh size h halves and the number of triangles quadruple. The initial crack fields for the three mesh refinements (visualized using damage-field d) are presented in fig. 5.6.



(a) domain Ω. (b) a coarse mesh Ω^h. (c) partitioned mesh $\{\Omega_i^h\}_{i=1}^{10}$.

Figure 5.5: domain Ω, mesh Ω^h, and partitioned mesh $\{\Omega_i^h\}_{i=1}^{10}$ for test 1. (a) also illustrates the boundary conditions applied to test 1. (b) represents a coarse unstructured finite element mesh with 'nearly' duplicate Dirichlet nodes for the initial crack.



(a) d at mesh level 1. (b) d at mesh level 2. (c) d at mesh level 3.

Figure 5.6: initial crack visualization of test 1 via damage-field d at different mesh levels.

Solver validation

Using the test 1 we cross-validate and compare our PSD solver (sequential and parallel) against benchmark solutions of this test available in the literature. In fig. 5.7, the top surface reaction force F_y versus applied displacements is plotted for various mesh refinement levels (detailed in table 5.2) and compared to a reference anisotropic phase-field solution from [**amor2009regularized**].

In fig. 5.7, the mesh convergence is evidenced from the improving PSD solutions (solid lines) towards the reference solution, hence validating the PSD solver. Our computations (at level 3) are in good agreement with the results provided in [**amor2009regularized**]. This simulation was executed using 10 processes on the desktop PC. The parMETIS partitioned mesh with 10 subdomains is presented in fig. 5.5c.

To further validate the PSD solver, we compare the errors in computing the maximum reaction-force $\max(F_y)$ obtained from our solver against two different reference solutions provided in [**Ambati2014**] and [**amor2009regularized**]. The last two columns of table 5.1 enumerate these errors. At finest mesh level 3, these errors decrease down to less than 1%. Alongside the plot in fig. 5.7, four instantaneous snapshots of the calculated damage-fields are presented. These damage-fields are obtained from the simulation of test 1 at the finest mesh level 3. Damage-field evolution, crack initiation, and propagation can be observed in these snapshots. As expected, under extreme tensile loading, the crack can be seen to travel along a (almost) straight line dividing the square specimen into two (almost) equal halves. Note that for additional validation, other literature comparative tests (mode I, mode II, and mode III fracture) were also performed but

Table 5.2: characteristics and computational details for the different finite element meshes used for test 1. $E1_{\max(F_y)}$ and $E2_{\max(F_y)}$ are are the the maximum reaction force errors computed against references [**amor2009regularized**] and [**Ambati2014**], respectively.

mesh	nodes	triangles	h	N_{DOF}	N_{nz}	$E1_{\max(\mathbf{l})}$	$E2_{\max(\mathbf{l})}$
level 1	8,353	16,384	0.0156	25,059	520,425	17.82%	19.22%
level 2	33,089	65,536	0.0078	99,267	2,073,033	5.68%	8.12%
level 3	131,713	262,144	0.0039	395,139	8,274,825	0.45%	0.61%

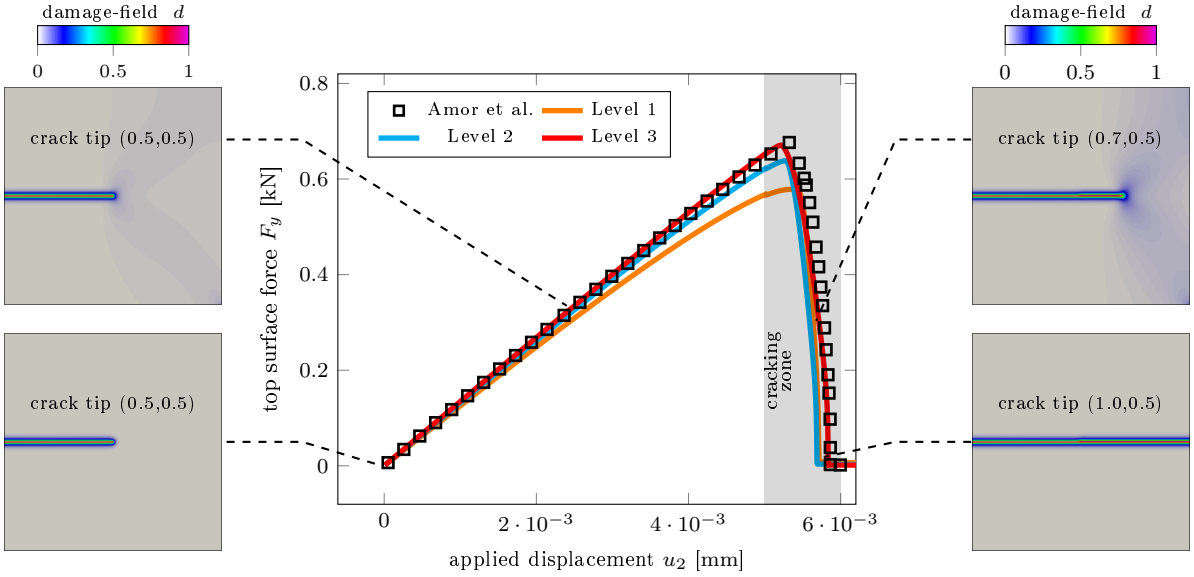


Figure 5.7: mesh convergence demonstrated via force-displacement plot for the two-dimensional single-edge notched tensile fracture simulation, test 1. The solid lines (Level 1 to 3) refers to our PSD solution for different mesh refinements and the square markers denote the reference solution (obtained by the anisotropic phase-field method) presented by [**amor2009regularized**].

these are not shown here for the sake of conciseness.

5.3 Validating the PSD soil-dynamic solver with paraxial boundary conditions

In this subsection we would compare the paraxial absorbing elements implemented in PSD against other absorbing boundary conditions available in CAST3M[4]. This document also serves as a naive cross validation of the PSD solvers parallel/sequential kernel developed for soil-dynamics.

5.3.1 Numerical experiment 1: A two-dimensional square

The geometry is considered to be a square with 50 m side, meshed with 1 m elements. This test is inspired by the validation tests performed in the paper of Bambeger et al. 1988. The paraxial conditions (resp. absorbing boundary conditions) apply to the bottom, left, and right borders for the PSD simulation (resp. for the CAST3M simulation). Figure 5.8 illustrates the geometry, red borders depict the absorbing (paraxial) borders and the green one depicts the free boundary condition.

[4]The CAST3M numerical experiments are performed by Reine Fares - Research Engineer - CEA/SEMT.

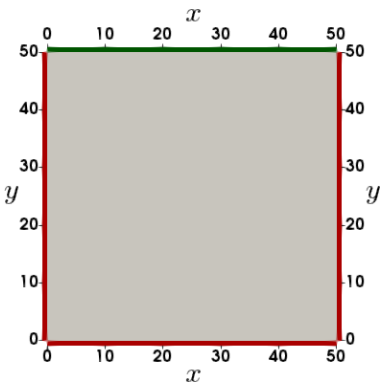


Figure 5.8: The two-dimensional geometry with paraxial borders in red and the free field border in green.

The essential algorithmic parameters concerning time discretization are enlisted below, while the material properties are tabulated in table 5.3.

- Time step used for the generalized-α scheme is $\mathrm{d}t = 0.01$ sec
- The generalized-α parameters read $\alpha_m = 0.0$ and $\alpha_f = 0.0$. Note that these null values for α_m and α_f transforms the generalized-α time discretization scheme into a Newmark-β time discretization scheme. Hence, a Newmark-β is used both in PSD and CAST3M.
- The simulation is run for $t = 4.0$ seconds, hence requiring 400 iterations of the solver.

Case	ρ [kg m^{-3}]	E [Pa]	ν
Test 1	2500	6.62E6	0.45

Table 5.3: Parameters for the seismic test.

Before proceeding to the simulation comparison it is notified that meshes used by PSD and CASTEM, are triangular and quard type, respective. Figure 5.9 illustrates the difference between the PSD and the CASTEM meshes.

5.3.2 Test 1: top loading of the square

We test how the solvers behave when the free boundary is loaded.

- Concerning the loading, on the center region of the top border we apply a small sinusoidal excitation spread over 1 seconds. The applied force reads:

$$\int_{\partial\Omega} (\sigma \cdot \mathbf{n}) \cdot \mathbf{v} = \int_{\partial\Omega} \left(\rho c_{\mathrm{p}} \left(\sin(2\pi t/1.0)\right) \mathbb{1}_{[x>20 \;\&\; x<30 \;\&\; y=50]} \times \mathbb{1}_{[t\leq 1]}\right) v_1$$

5.3.3 Test 2: bottom loading of the square

We test how the solvers behave when one of the paraxial (absorbing) boundary is loaded.

- Concerning the loading, on the center region of the bottom border we apply a small sinusoidal excitation spread over 1 seconds. The applied force reads:

$$\int_{\partial\Omega} (\sigma \cdot \mathbf{n}) \cdot \mathbf{v} = \int_{\partial\Omega} \left(\rho c_{\mathrm{p}} \left(\sin(2\pi t/1.0)\right) \mathbb{1}_{[x>20 \;\&\; x<30 \;\&\; y=0]} \times \mathbb{1}_{[t\leq 1]}\right) v_1$$

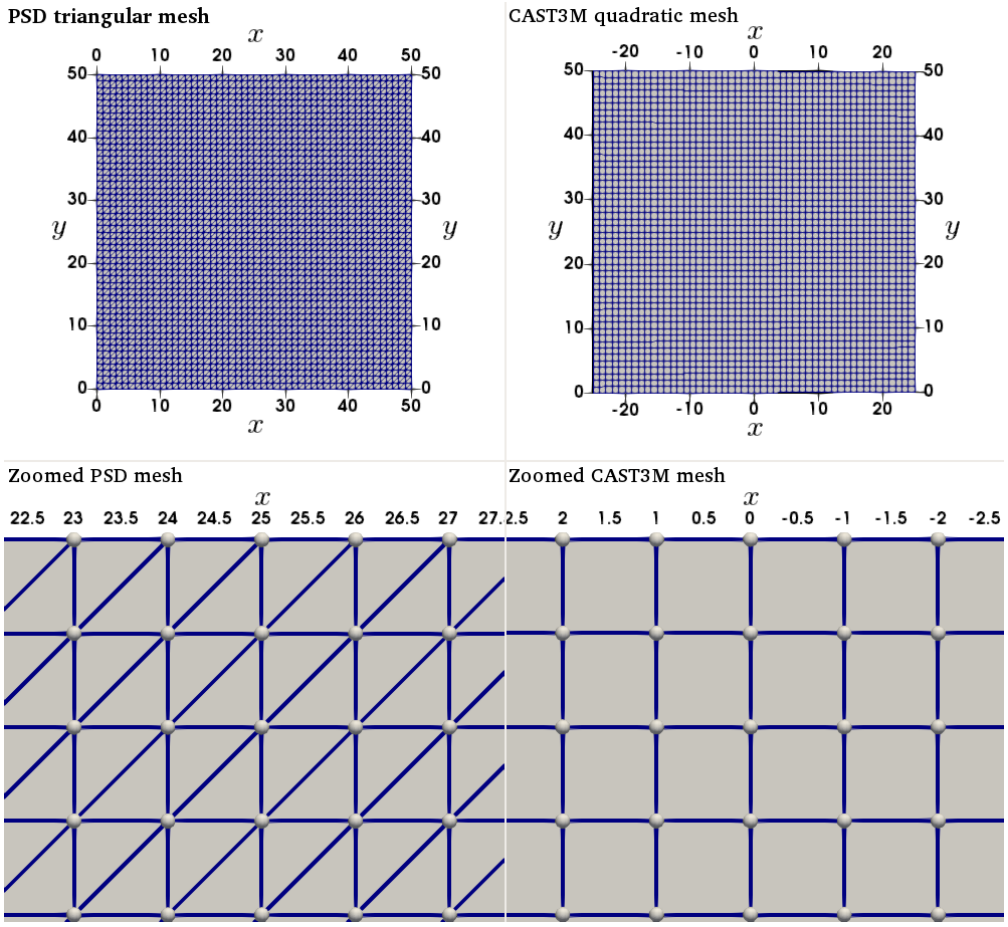


Figure 5.9: Meshes used by CAST3M (right) and PSD (left).

5.3.4 Numerical experiment 2: 3D case with complex geometry

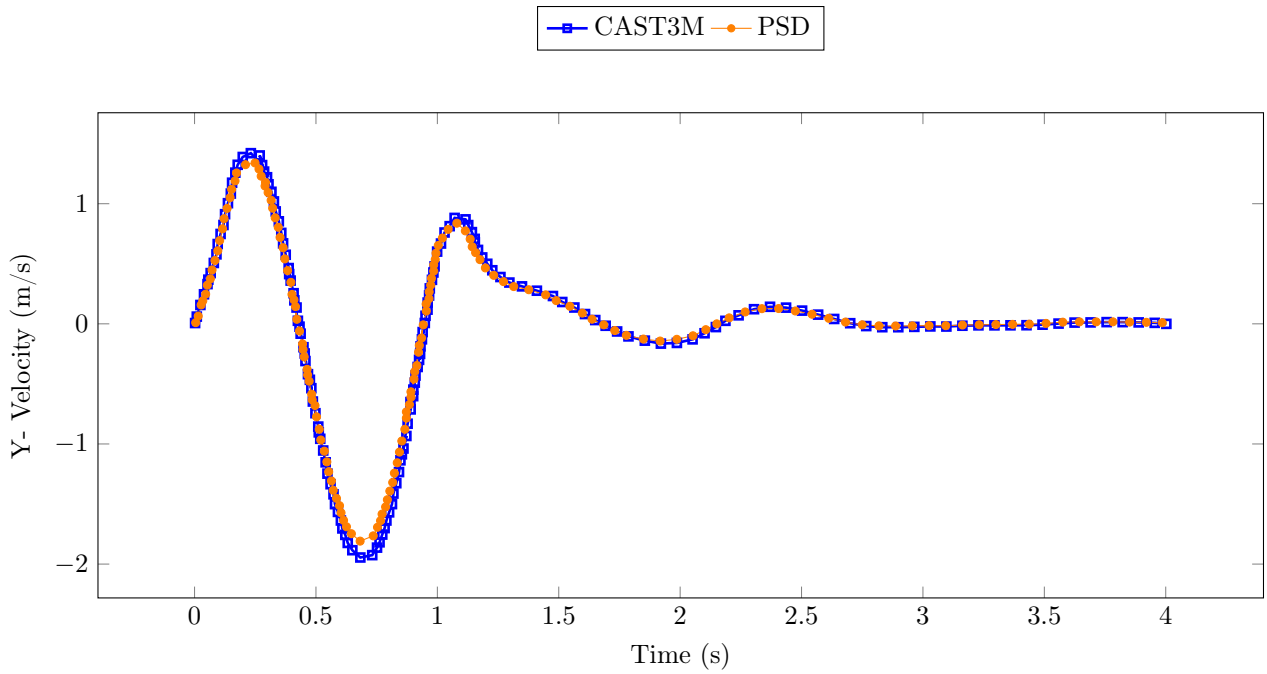


Figure 5.10: Test 1 results. Comparison of Y-velocities of a point $\mathbf{x} = (25, 50)$ obtained by CAST3M and PSD for a 4 second simulation with 1 second sinusoidal wave excitation.

Case	package	version	time	boundary
Test 1	PSD	sequential	96	paraxial
Test 1	PSD	sequential-opt.	45	paraxial
Test 1	PSD	parallel	9	paraxial
Test 1	CAST3M	sequential	180	Lysmer-type

Table 5.4: CPU time comparison of CAST3M and PSD for different test/versions. CPU time is given in seconds, PSD version sequential-opt. means PSD with the GFP library (GoFastPlugins), by default the GFP library is used for PSD parallel version. As the mesh is tiny, only 4 MPI processes are used in the PSD parallel version.

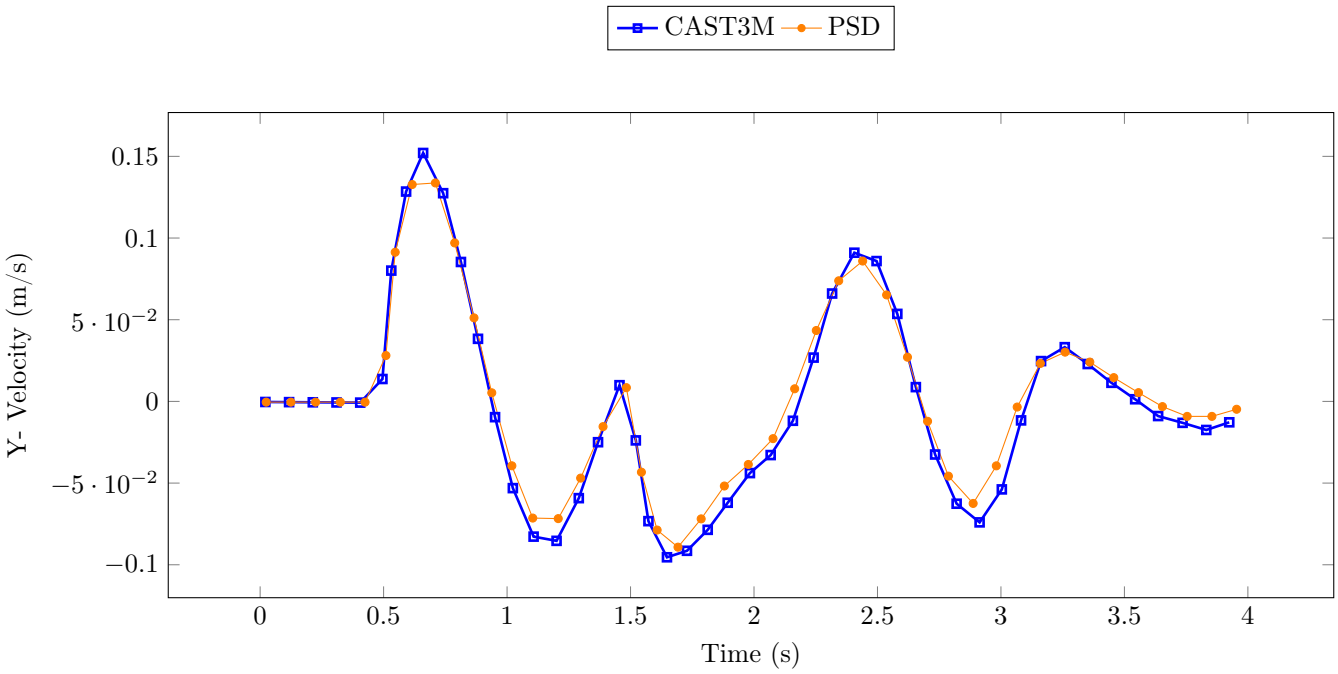


Figure 5.11: Test 1 results. Comparison of Y-velocities of a point $\mathbf{x} = (25, 0)$ obtained by CAST3M and PSD for a 4 second simulation with 1 second sinusoidal wave excitation.

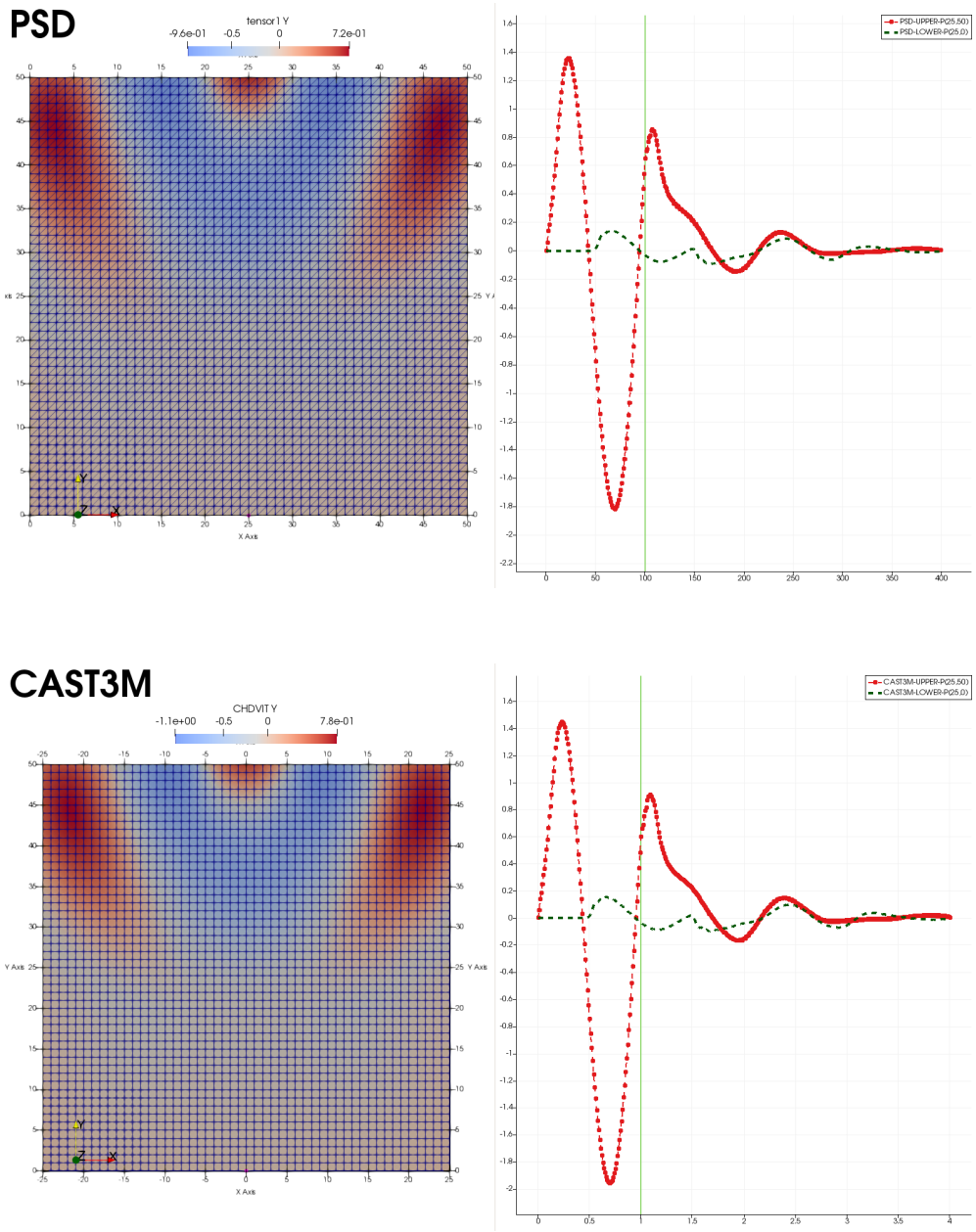


Figure 5.12: Comparison of simulations performed in CAST3M and PSD. Left: Y-velocity field snapshot at $t = 1.0$ second. Right: Top and bottom border point ($x = 25$) time history for Y-velocities.

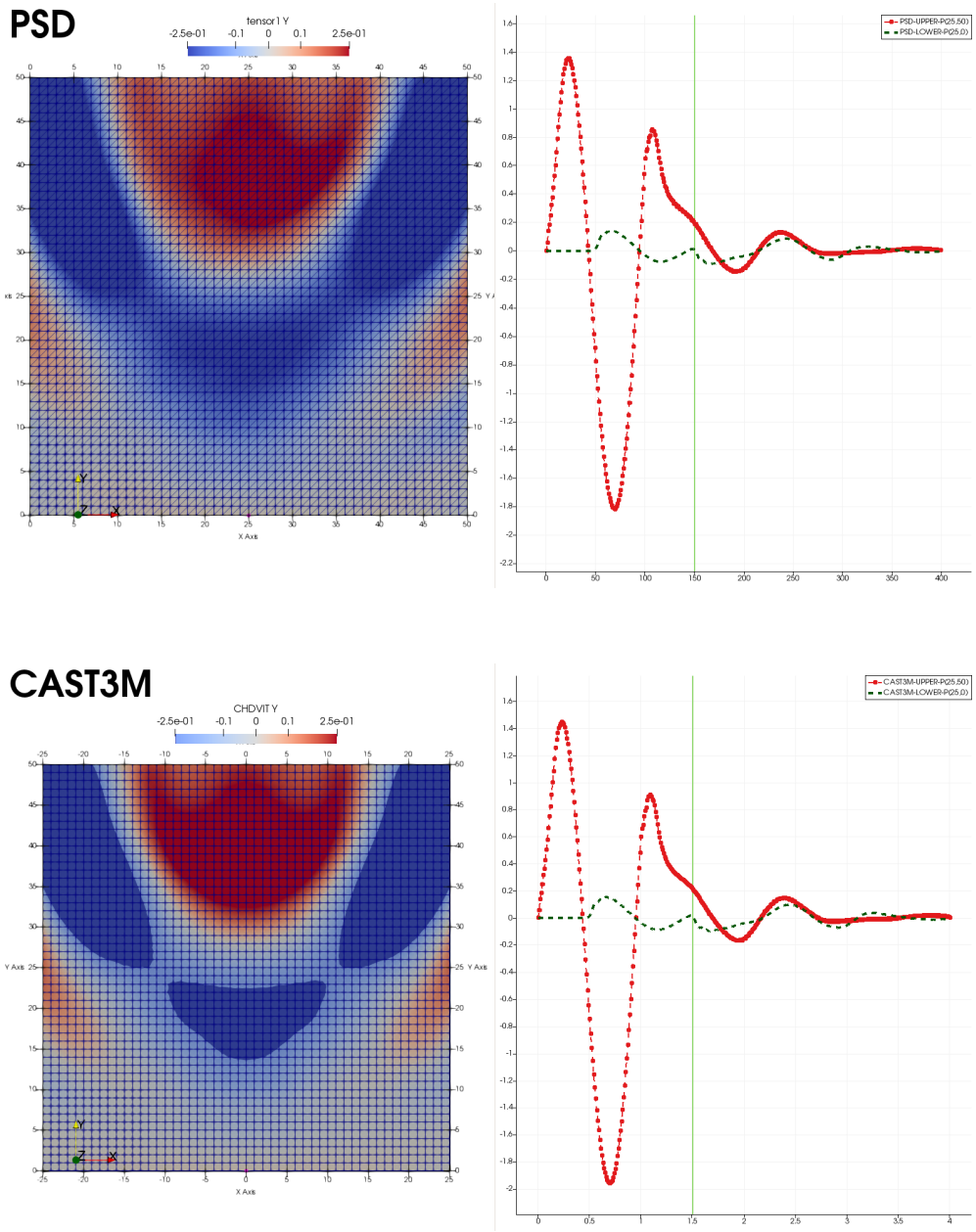


Figure 5.13: Comparison of simulations performed in CAST3M and PSD. Left: Y-velocity field snapshot at $t = 1.5$ second. Right: Top and bottom border point ($x = 25$) time history for Y-velocities.

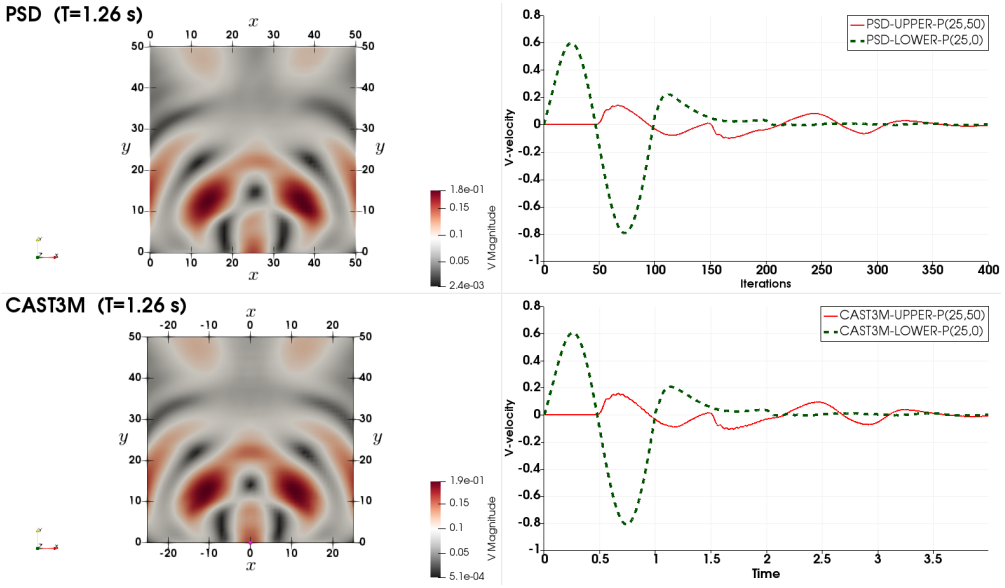


Figure 5.14: Comparison of simulations performed in CAST3M and PSD. Left: Y-velocity field snapshot at $t = 1.26$ second. Right: Top and bottom border point ($x = 25$) time history for Y-velocities.

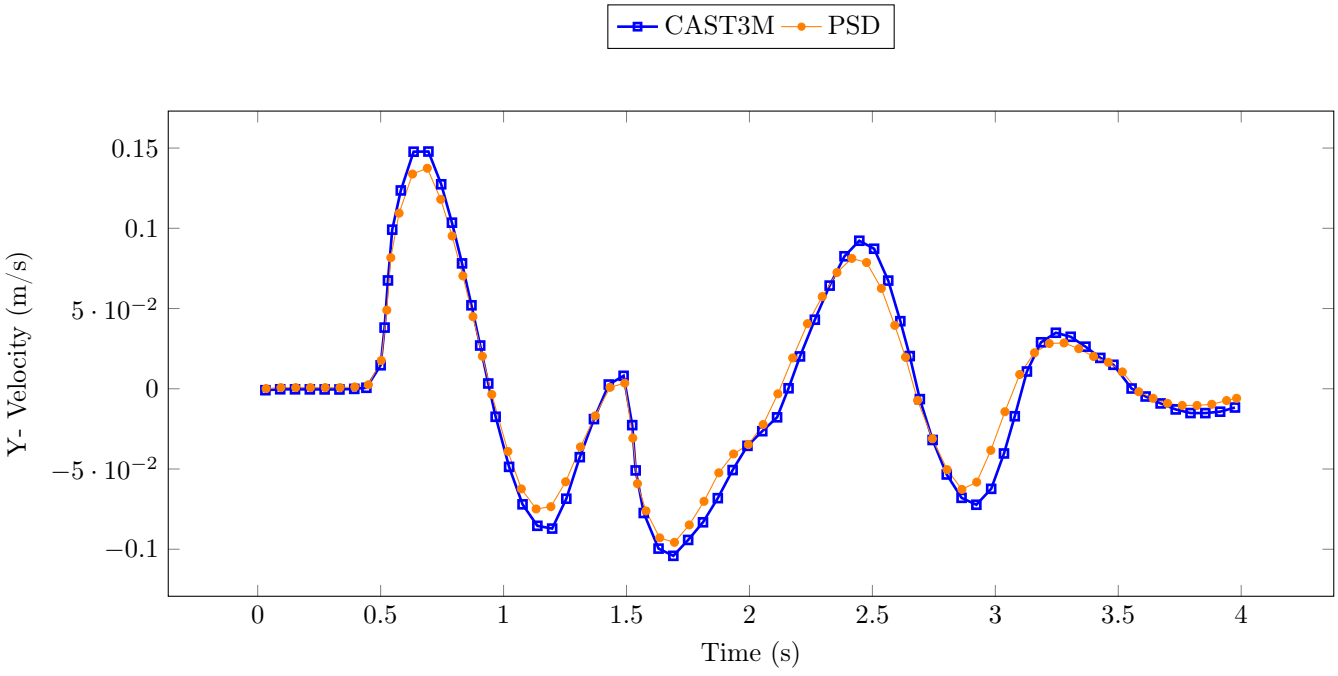


Figure 5.15: Test 2 results. Comparison of Y-velocities of a point $\mathbf{x} = (25, 50)$ obtained by CAST3M and PSD for a 4 second simulation with 1 second sinusoidal wave excitation.

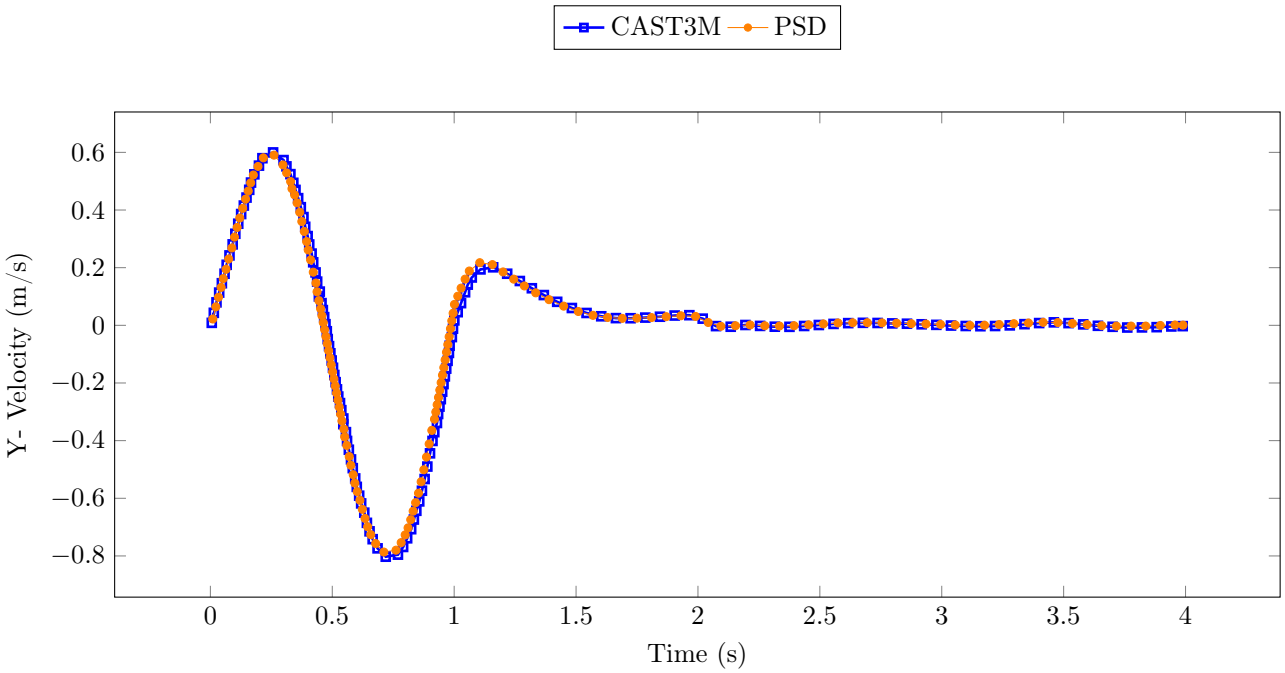


Figure 5.16: Test 2 results. Comparison of Y-velocities of a point $\mathbf{x} = (25, 0)$ obtained by CAST3M and PSD for a 4 second simulation with 1 second sinusoidal wave excitation.

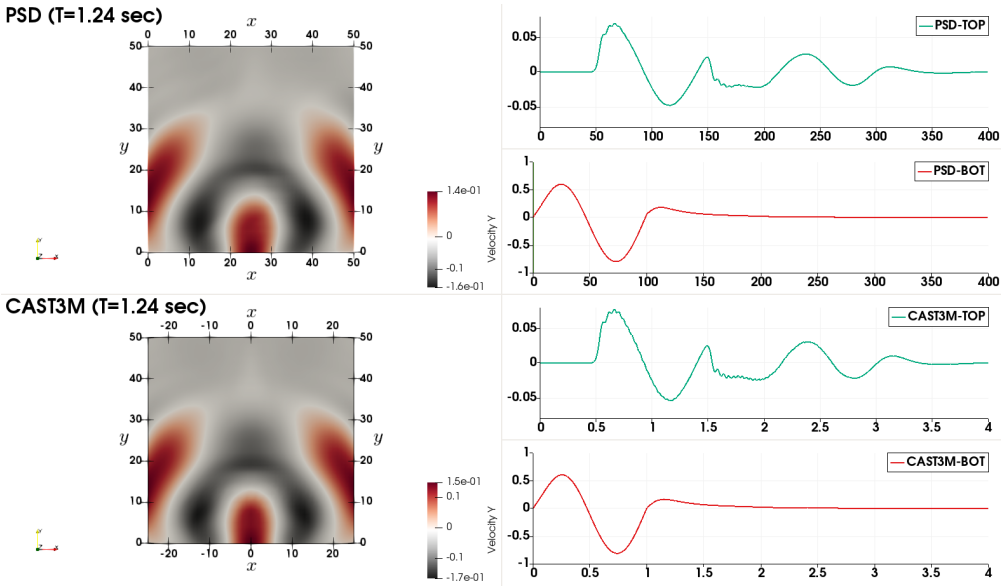


Figure 5.17: Comparison of simulations performed in CAST3M and PSD. Left: Y-velocity field snapshot at $t = 1.26$ second. Right: Top and bottom border point ($x = 25$) time history for Y-velocities.

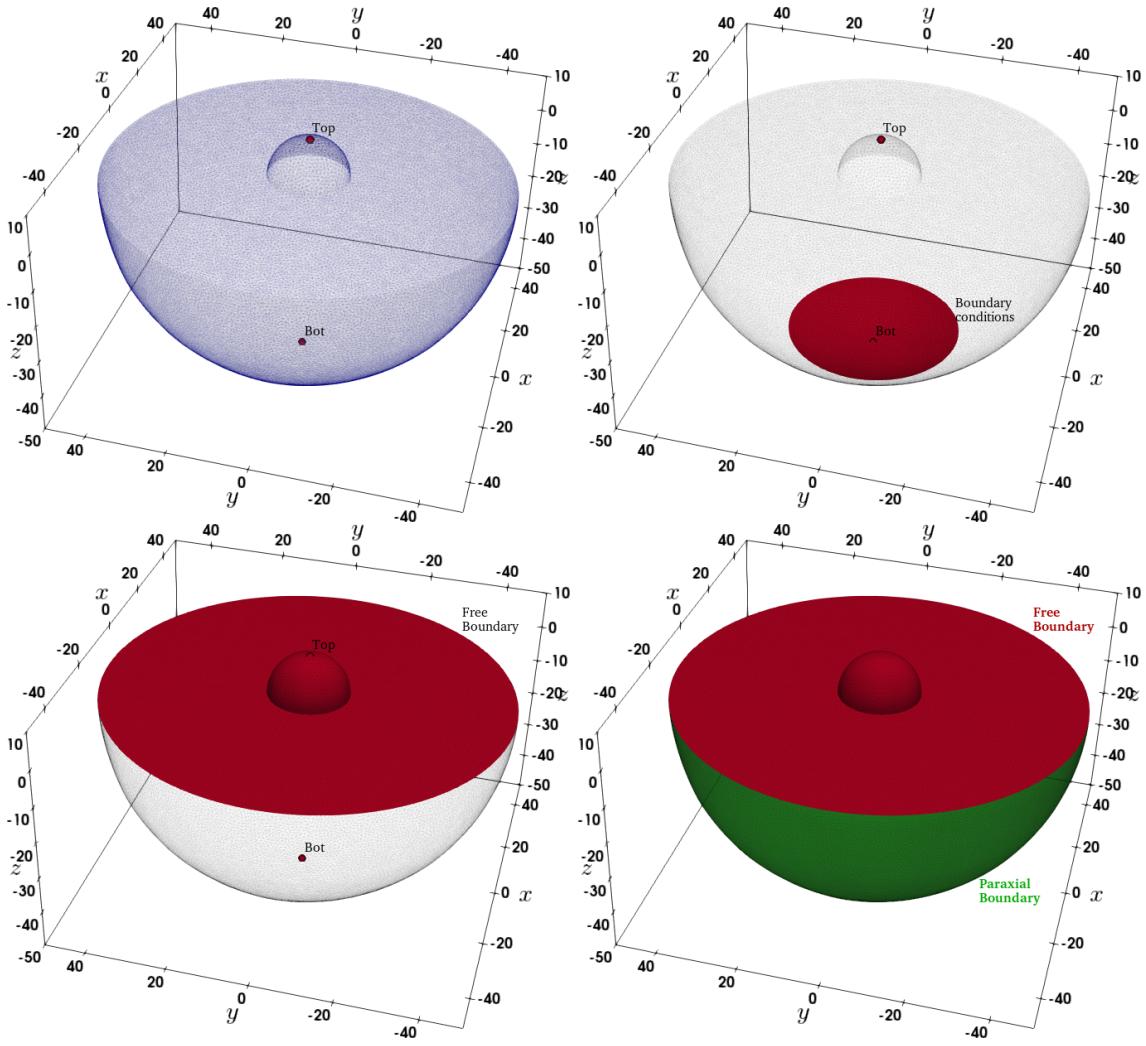


Figure 5.18: Seismic 3D cased with complex geometry and boundary conditions.

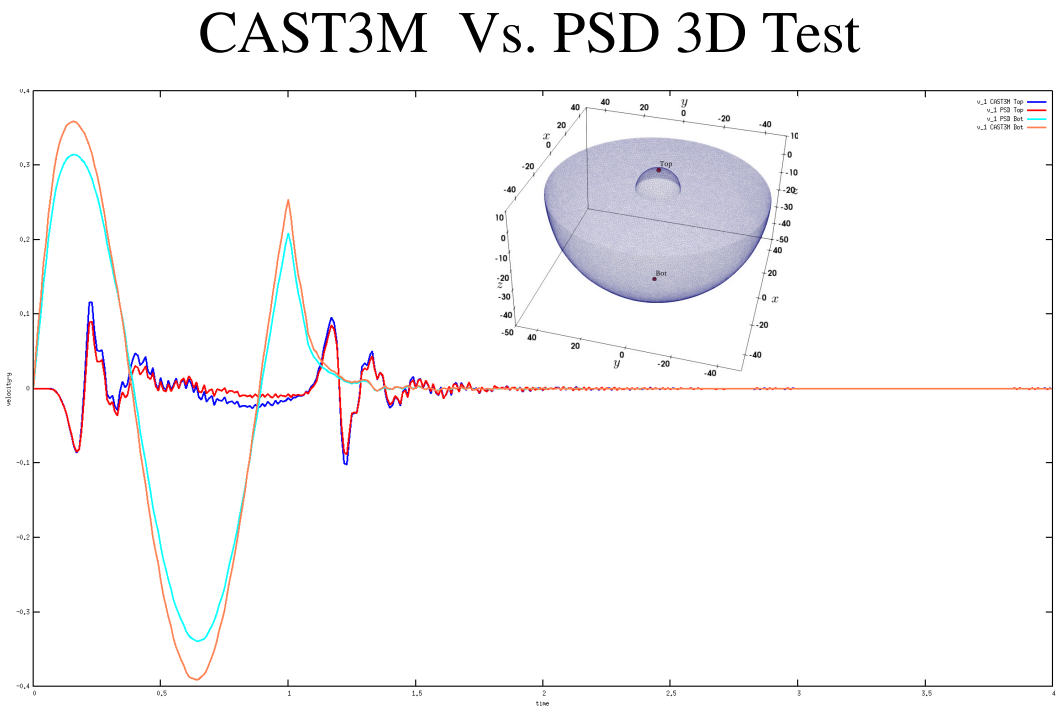


Figure 5.19: Comparison of 3D simulations performed in CAST3M and PSD.

Chapter 6

Functions and descriptions

6.1 Flags for PSD_PreProcess

The `PSD_PreProcess` relies heavily on command line flags for user interaction. These flags become a medium of communication between the user and the PSD solver. Three types of flags can be used i) `int` type : these are integer type flags which expect an integer argument, ii) `string` type : these flags expect a string argument, and iii) `bool` type : these are boolean type flags with no argument.

6.1.1 Integer type flags used for PSD_PreProcess

`-dirichletpointconditions`	Number of Dirichlet points. Default `0`.
`-dirichletconditions`	Number of Dirichlet boundaries. Default `0`.
`-bodyforceconditions`	Number of regions to which body force is applied. Default `0`.
`-tractionconditions`	Number of Neumann/traction boundaries. Default `0`.
`-parmetis_worker`	Active when mesh partitioner is parmetis.
`-dimension`	Dimension of problem. 2 for 2D 3 for 3D. Default 2.
`-lagrange`	Lagrange order used for building FE space. Default `1 for P1`.

6.1.2 String type flags used for PSD_PreProcess

`-timediscretization`	Time discretization type. Use `generalized-alpha | newmark-beta | hht-alpha | central-difference`.
`-nonlinearmethod`	Nonlinear method type. Use `Picard | Newton-Raphsons`.
`-partitioner`	Mesh partitioner. Use `metis | scotch | parmetis`.
`-postprocess`	Indicate postprocessing quantity. Use `u | v | a | phi | uphi | uva`.
`-doublecouple`	Soil dynamics double couple boundary condition. Use `force-based | displacement-based`.
`-reactionforce`	Reaction force calculation method. Use `stress-based | variational-based`.
`-problem`	Interested problem. Use `linear-elasticity | damage | elastodynamics | soildynamics`.
`-model`	Interested model. Use `hybrid-phase-field | Mazar | pseudo-nonlinear`.

6.1.3 Bool type flags used for PSD_PreProcess

`-help`	Helping message on the terminal.
`-debug`	OpenGL plotting routine for displaying solution.
`-useGFP`	Activate use of GoFastPlugins. A suite of C++ plugins.
`-timelog`	To setup time logging for various phases of the solver.
`-useRCM`	Mesh level renumbering via Reverse Cuthill Mckee.
`-vectorial`	Generate vectorial space solver for non-linear.
`-sequential`	To generate a sequential PSD solver.
`-pointprobe`	Setup a point probe to record variables.
`-energydecomp`	Hybrid phase-field energy decomposition.
`-top2vol-meshing`	top-ii-vol point source meshing for soil-dynamics.
`-getreactionforce`	Extraction reactions at surface.
`-plotreactionforce`	Live pipe plotting using GnuPlot.
`-withmaterialtensor`	Use material tensor to build stiffness matrix.
`-dirichletpointconditions`	To activate pre-cracked surface Dirichlet.

6.2 Flags for PSD_Solve

Just like `PSD_PreProcess`, the solver in PSD `PSD_Solve` also relies heavily on command line flags for user interaction. These flags become a medium of communication between the user and the PSD solver. Three types of flags can be used i) `int` type : these are integer type flags which expect an integer argument, ii) `string` type : these flags expect a string argument, and iii) `bool` type : these are boolean type flags with no argument.

6.2.1 Integer type flags used for PSD_Solve

`-np`	Number of processes. Default `2. Values accepted` 1 `to` ∞
`-v`	FreeFEM flag verbosity of the background FEM kernel. Default `1. Set 0 for no output. Values accepted 0 to 1000`
`-split`	FreeFEM flag to split uniformly the mesh. Default `1 = no splitting. Use 2 to split once. Values accepted 2 to 12.`
`-ksp_max_it`	PETSc flag to choose Krylov solver max iterations. Default 1000.

6.2.2 Real type flags used for PSD_Solve

`-ksp_atol`	PETSc flag to set absolute tolerance for the Krylov solver convergence. Default 1e-5.
`-ksp_rtol`	PETSc flag to set relative tolerance for the Krylov solver. Default 1e-5.

6.2.3 String type flags used for PSD_Solve

`-mesh`	Mesh file to be used. Either use .mesh or .msh. e.g., `-mesh nameofmesh.mesh` .
`-ksp_type`	PETSc flag to choose the Krylov solver. Default `cg` . Use `-ksp_type cg | gmres | bicg | bicgstab` .
`-pc_type`	PETSc PETSc flag to choose preconditioner for the solver. Default `bjacobi` . Use `-pc_type jacobi | bjacobi | lu | gamg` .
`-sub_pc_type`	PETSc flag to choose sub preconditioner for solver. Default `ilu` . Use `-sub_pc_type ilu | icc | lu | gamg` .

6.2.4 Bool type flags used for PSD_Solve

`-wg`	FreeFEM flag to activate with graphics option for OpenGL graphics. Default false.
`-nw`	FreeFEM flag to activate no windows output on screen. Default false.
`-ns`	FreeFEM flag to activate no script output on screen. Default false.

6.3 Flags as per physics

Linear Elasticity

INT TYPE

-dirichletpointconditions
-dirichletconditions
-bodyforceconditions
-tractionconditions
-parmetis_worker
-ksp_max_it
-dimension
-lagrange
-split
-np
-v

REAL TYPE

-ksp_rtol
-ksp_atol

STRING TYPE

-partitioner
-postprocess
-sub_pc_type
-ksp_type
-pc_type
-problem
-mesh

BOOL TYPE

-withmaterialtensor
-sequential
-timelog
-useGFP
-useRCM
-debug
-help
-wg
-nw
-ns

Damage Mechanics

INT TYPE

-dirichletpointconditions
-dirichletconditions
-bodyforceconditions
-tractionconditions
-parmetis_worker
-ksp_max_it
-dimension
-lagrange
-split
-np
-v

REAL TYPE

-ksp_rtol
-ksp_atol

STRING TYPE

-nonlinearmethod
-reactionforce
-partitioner
-postprocess
-sub_pc_type
-ksp_type
-pc_type
-problem
-mesh
-model

BOOL TYPE

-crackdirichletcondition
-plotreactionforce
-getreactionforce
-energydecomp
-sequential
-vectorial
-timelog
-useGFP
-useRCM
-debug
-help
-wg
-nw
-ns

Elastodynamics

INT TYPE

-dirichletpointconditions
-dirichletconditions
-bodyforceconditions
-tractionconditions
-parmetis_worker
-ksp_max_it
-dimension
-lagrange
-split
-np
-v

REAL TYPE

-ksp_rtol
-ksp_atol

STRING TYPE

-timediscretization
-partitioner
-postprocess
-sub_pc_type
-ksp_type
-pc_type
-problem
-mesh
-model

BOOL TYPE

-sequential
-timelog
-useGFP
-useRCM
-debug
-help
-wg
-nw
-ns

Soildynamics

INT TYPE

-dirichletpointconditions
-dirichletconditions
-bodyforceconditions
-tractionconditions
-parmetis_worker
-ksp_max_it
-dimension
-lagrange
-split
-np
-v

REAL TYPE

-ksp_rtol
-ksp_atol

STRING TYPE

-doublecouple
-postprocess
-partitioner
-postprocess
-sub_pc_type
-ksp_type
-pc_type
-problem
-mesh
-model

BOOL TYPE

-top2vol-meshing
-sequential
-timelog
-useGFP
-useRCM
-debug
-help
-wg
-nw
-ns

6.4 Functions in gofastplugins.cpp

6.4.1 GFPeigen

`GFPeigen(A,Eval,Evec);`	`A` is a matrix, `Eval` is vector returning eigenvalues, `Evec` is the matrix returning eigenvectors.

This is a call by reference pointer-based function of GFP library. It is used for computation of the eigenvalues and eigenvectors of a real symmetric matrix (upper triangular). This function inturn uses LAPACK libraries `dsyev_` for calculation of eigenvalues and eigenvectors.

The function `GFPeigen` which can be called from PSD is coded as `lapack_dsyevIn` function within the gofastplugins.cpp. this function argument 1: `A` is the supplied symmetric matrix, argument 2: `vp` are the output eigenvalues and argument 3: `vectp` are the output eigenvectors.

```
long lapack_dsyev (KNM<double> *const &A, KN<double> *const &vp, KNM<double> *
    const &vectp)
{
  intblas n = A->N();
  KNM<double> mat(*A);

    .
    .
    dsyev_(&JOBZ, &UPLO, &n, mat, &n, *vp, w, &lw, &info);
    .
    .
    *vectp = mat;
}
```

6.4.2 GFPeigenAlone

`GFPeigen(A,Eval,Evec);`	`A` is a matrix, `Eval` is vector returning eigenvalues.

This is a call by reference pointer based function of GFP library. It is used for computation of the eigenvalues of a real symmetric matrix (upper triangular). This function inturn uses LAPACK library for calculation of eigenvalues. The function `GFPeigenAlone` which can be called from PSD is coded as `lapack_dsyevAlone` function within the gofastplugins.cpp. In this function argument 1: `A` is the supplied symmetric matrix and argument 2: `vp` is the output eigenvalues of matrix `A` .

```
long lapack_dsyevAlone (KNM<double> *const &A, KN<double> *const &vp)
```

6.4.3 GFPmaxintwoFEfields

This is a call by reference pointer based function of GFP library. It is used to find out max between two real numbers `f` and `f1` (two 1D arrays). The max is stored in array `f`.

```
double GFPmaxintwoP1(KN<double> *const & f, KN<double> *const & f1)
```

Chapter 7

Gallery

This chapter showcases some test cases that have been performed with PSD.

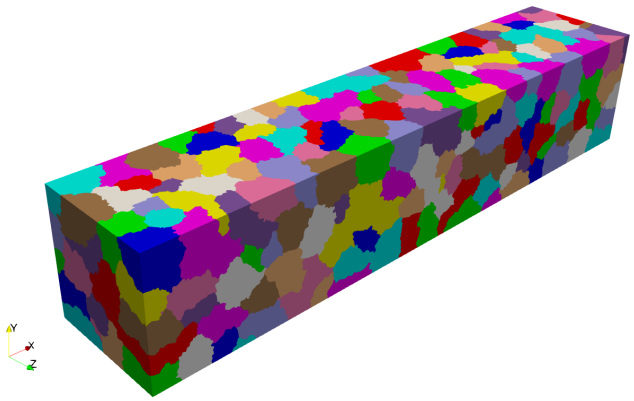


Figure 7.1: 90 M dof with 400 partitions.

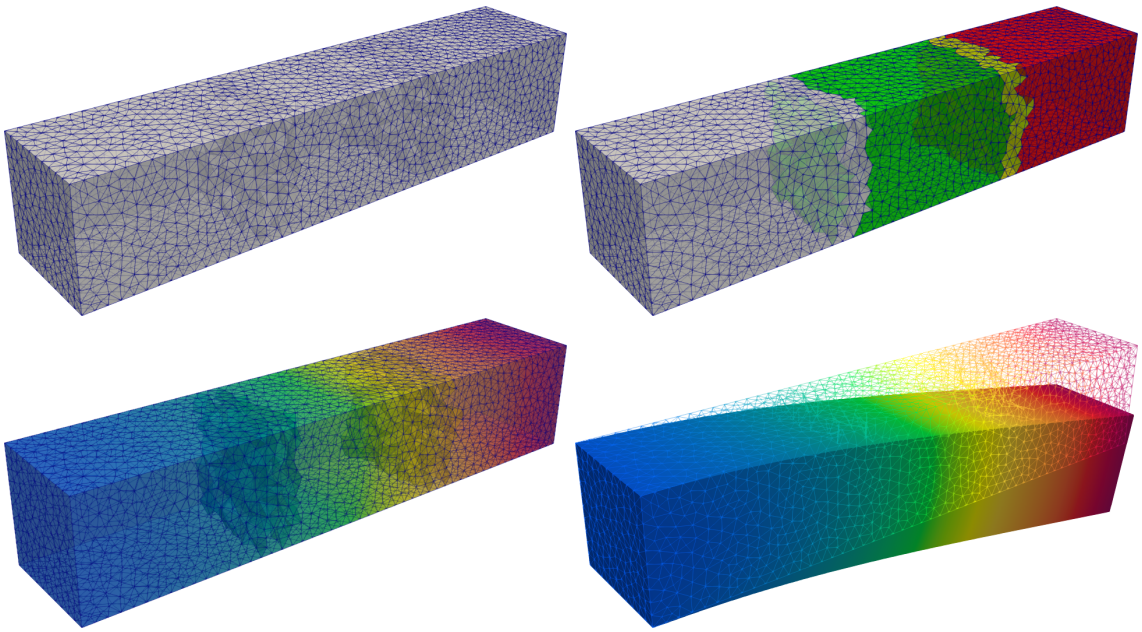


Figure 7.2: Bending of clamped bar under loading.

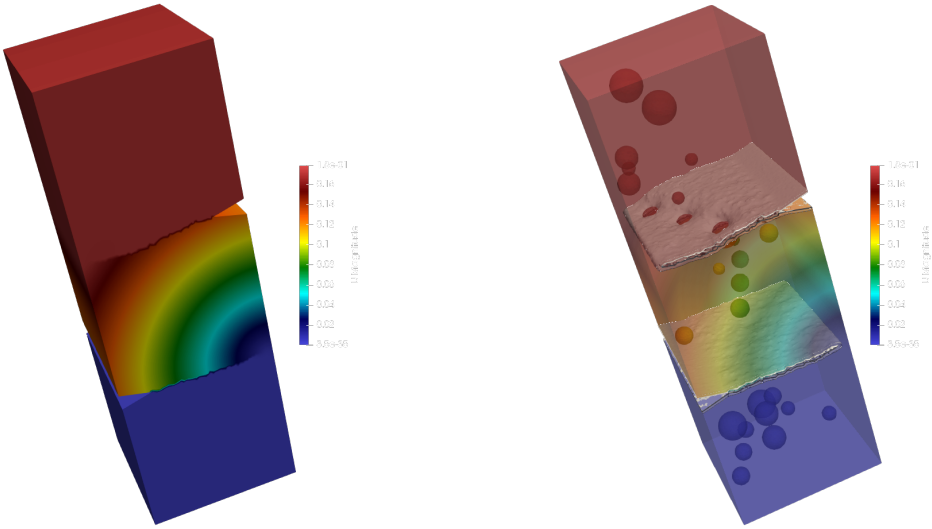


Figure 7.3: Perforated concrete bar cracking.

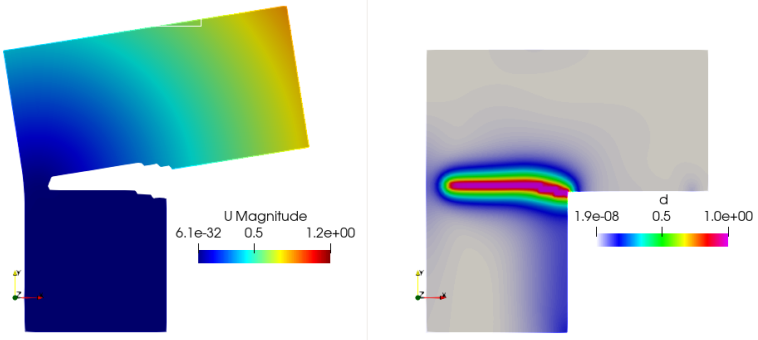


Figure 7.4: Point loading causing fracture in L-shaped material.

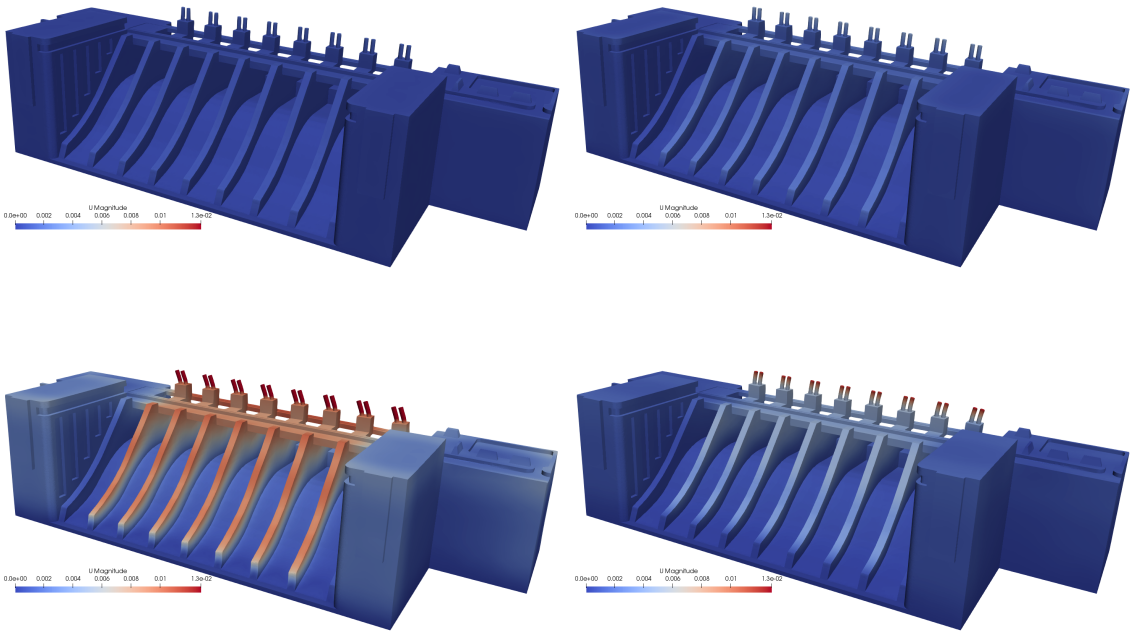


Figure 7.5: Full scale dam under seismic load.

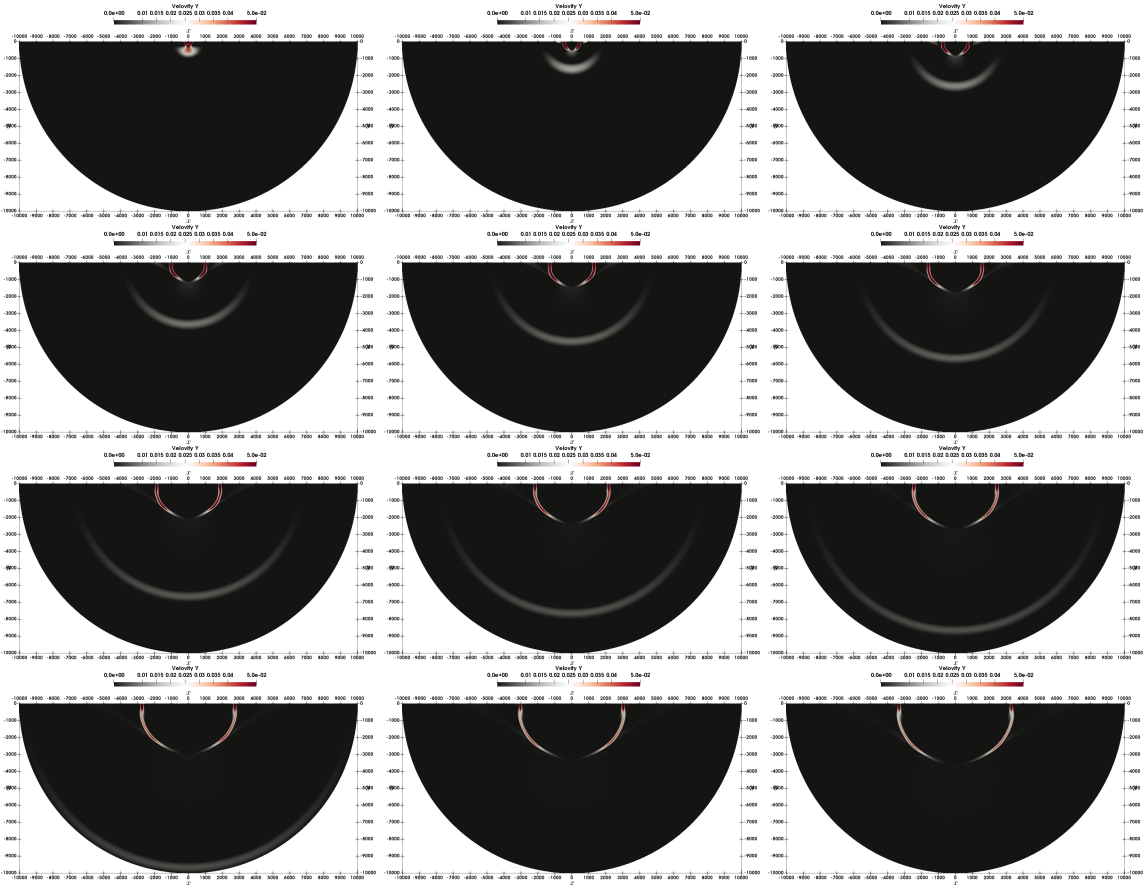


Figure 7.6: Seismic signal dispersion in soil.

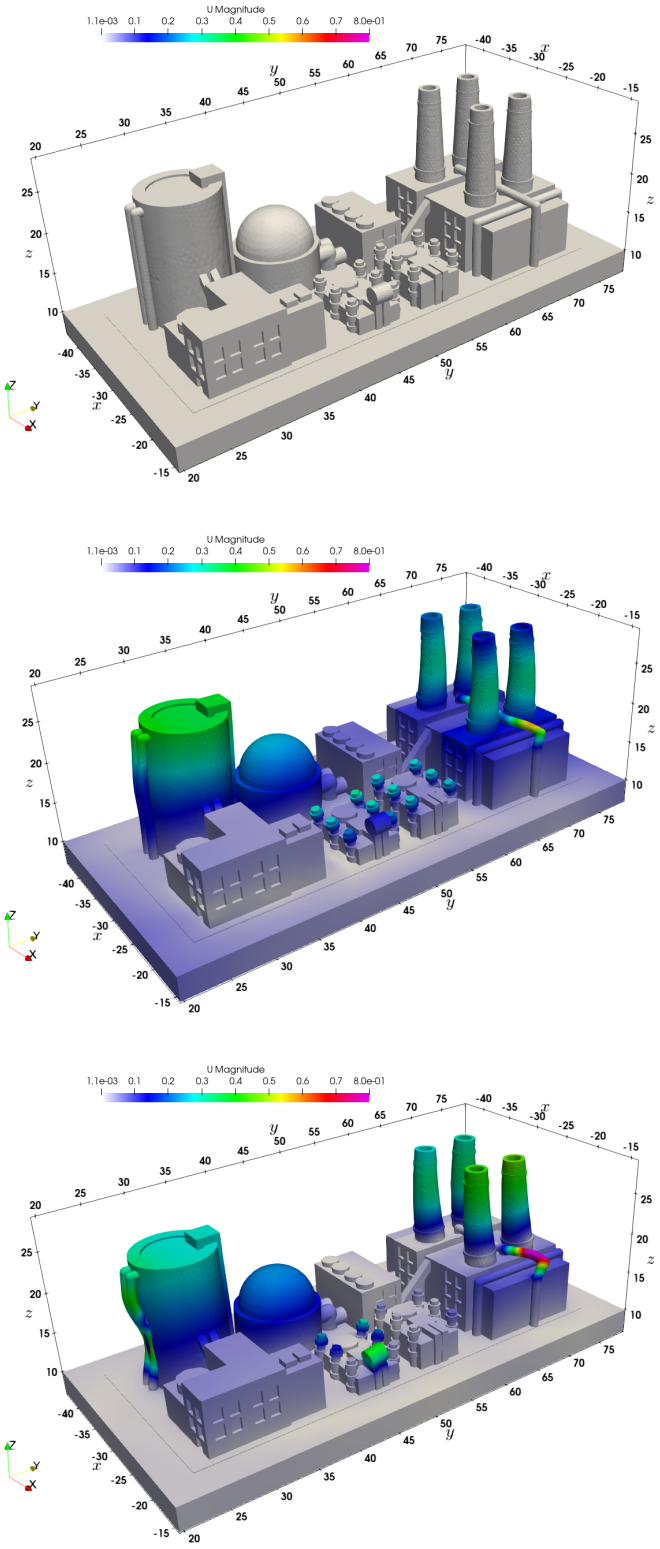


Figure 7.7: Seismic signal on nuclear site.

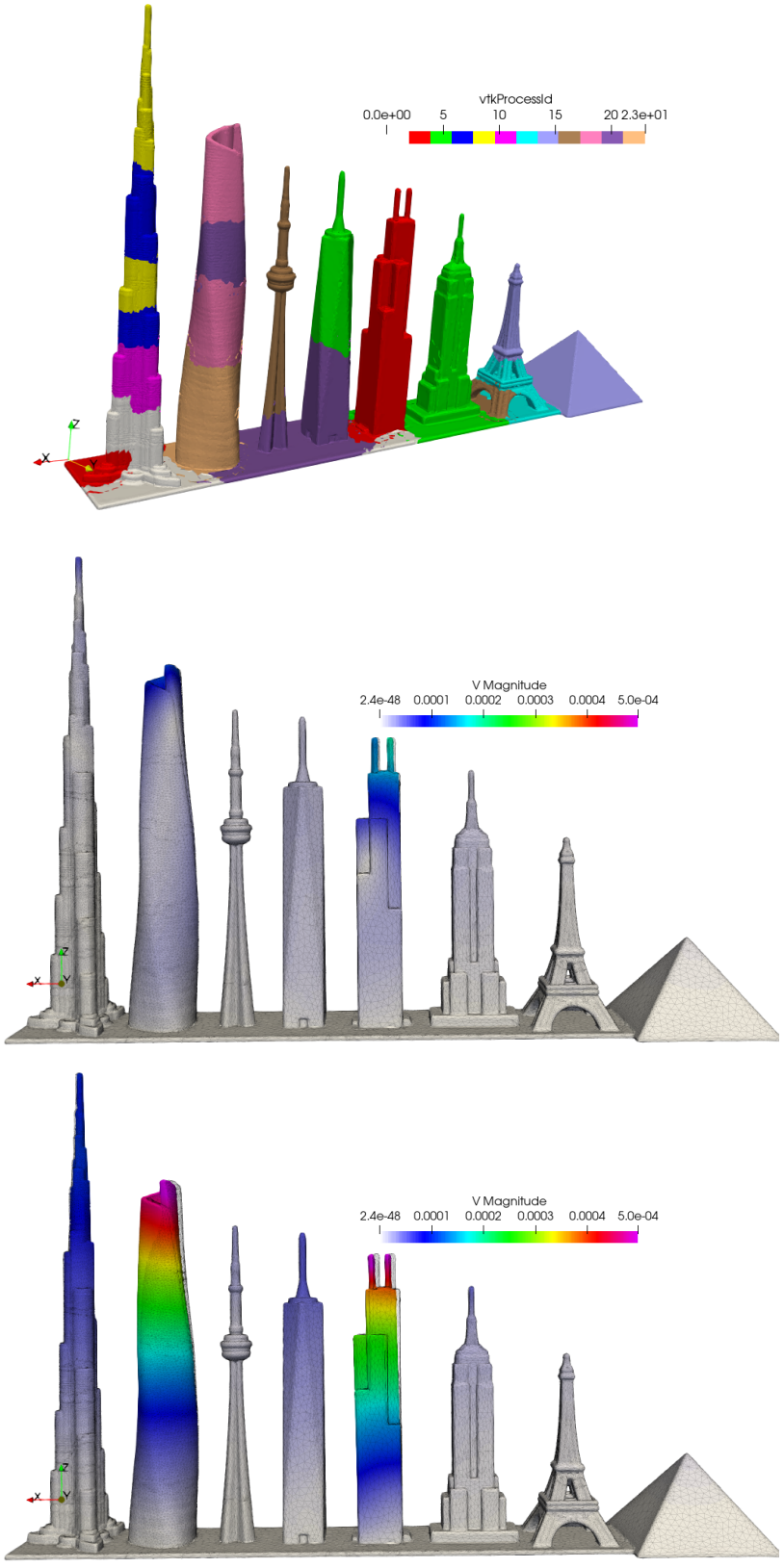


Figure 7.8: Seismic signal on famous world buildings.

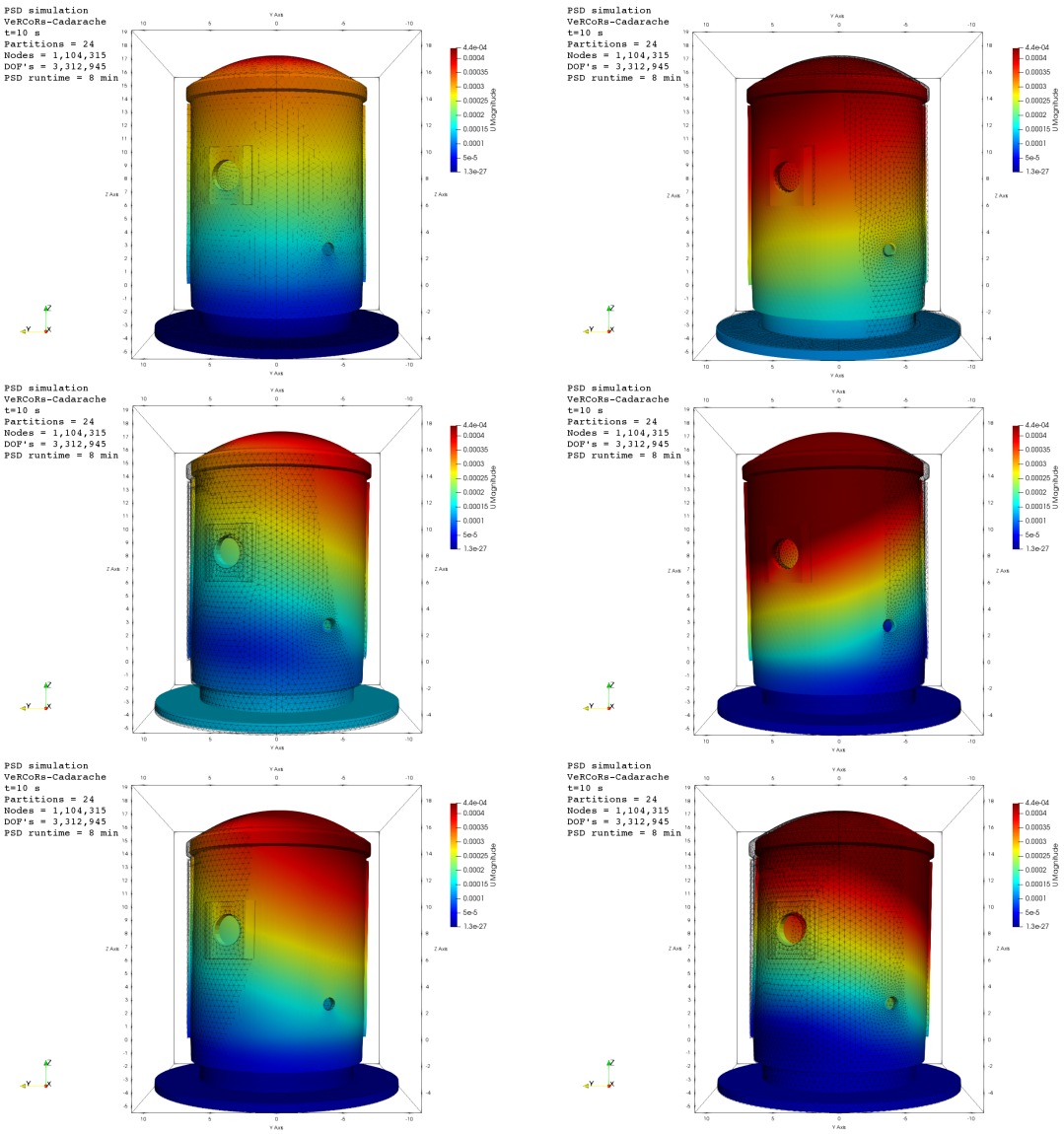


Figure 7.9: Seismic signal on vercor nuclear building.

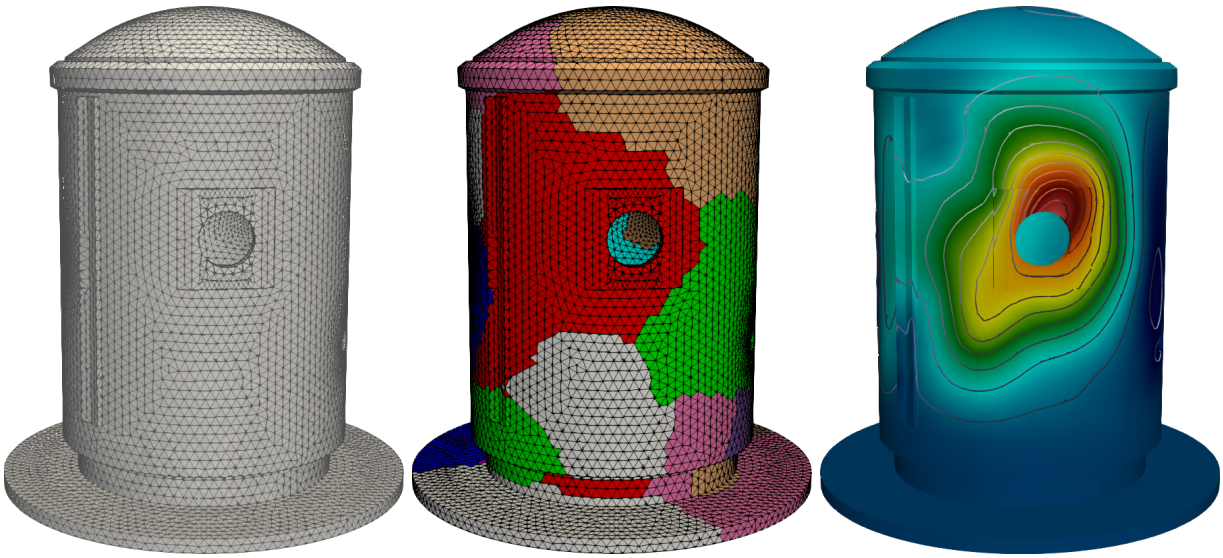


Figure 7.10: Damage mechanics of vercor nuclear building.