pogo.

www.pogo.software

User's Manual

Peter Huthwaite, Stefano Galvan
and
Kostas Zarogoulidis

Non-Destructive Evaluation Laboratory
Department of Mechanical Engineering
Imperial College London
South Kensington
London, SW7 2AZ
UK

Email: support@pogo.software
Web: www.pogo.software

Version
1.0.00
June 2017

Contents

Chapter 1

Introduction

Pogo is a finite element software package designed to very quickly calculate linear elastodynamic wavefields in the time domain. The finite element (FE) method is very general and well suited to a variety of geometrical configurations, but the time needed to run these models can be significant. By exploiting the capabilities of the graphical processing unit (GPU) Pogo can significantly reduce the time needed to run simulations, typically by 1-2 orders of magnitude compared to typical alternative CPU FE packages.

Pogo runs on NVIDIA graphics cards (compute 2.0 and above should work, see the Nvidia website for a list of GPU compute versions), and uses CUDA. There are two important performance criteria when evaluating the cards. Speed is (almost exclusively) dictated by the bandwidth of the card – typically these are of the order of around several hundred GB/sec. It is possible to obtain an approximation of how long a simulation will run through the equation

$$t_{runtime} = F\frac{n_{inc}n_{nodes}}{bw}$$

where bw is the bandwidth (in bits/sec), n_{inc} is the number of time increments in a simulation, n_{nodes} is the number of nodes, and F is a constant. For single precision 2D simulations, $F \approx 300$, while for double precision $F \approx 550$, and in single precision 3D $F \approx 2500$. Secondly, the GPU memory restricts the maximum size of the model which can be run. The NVIDIA Titan GPU has 6GB of memory, which would allow around 36 million nodes in 2D, or half this in double precision. Note that these values are without compression, so when using structured grids it may be possible to do much larger models (see Sect. 7.4 for more details about this). In 3D, around 4 million nodes could be used. The memory requirement can be assumed to scale fairly linearly with the number of nodes, except for rare cases where, for example, many sources are required.

For a theoretical outline of the equations and the technical approach used in Pogo, please see [1], downloadable from http://dx.doi.org/10.1016/j.jcp.2013.10.017. Up-to-date information on Pogo will be posted on http://www.pogo.software/.

There are two main packages for Pogo, the solver (and accompanying programs) and a graphical user interface, Pogo.Pro. In this document, the quickstart guide will extensively rely on Pogo.Pro, while the subsequent section focuses on the theoretical aspects of solving FE problems. Pogo capabilities, as well as how to exploit them in the model files, how to solve problems, then process the results are discussed.

1.1 Requirements

1.1.1 Hardware

The primary requirement for the Pogo solver is a CUDA enabled GPU. Since problem size is dependent on the number and size of GPUs on the system having sufficient large GPUs for the problem to be solved could be considered a secondary requirement. Pogo.Pro should run on most modern machines.

1.1.2 Software

At present, the Pogo solver is primarily supported on Linux, with binaries also available for Windows. In Linux, the most heavily tested distribution is Ubuntu, with the binaries known to run in 14.04, 16.04 and 17.04. The software has also been run on Red Hat 4.8. Provided the kernel is a sufficiently similar version then any distribution should work. Pogo.Pro should run on Windows and major distributions of Linux.

1.2 Installation

There are certain prerequisites for installation: the Nvidia drivers (in the case of the solver) and the HASP driver (for both Pogo.Pro and the solver). The instructions below first deal with the Pogo solver, which is slightly more complex since the GPU drivers must be installed. Then Pogo.Pro is dealt with.

1.2.1 Pogo solver installation

It should be noted that the Pogo solver package actually consists of several executables. Three of these (pogoSolve, pogoSolve64 and pogoSolve3d) are solvers and should all run in the same way, and hence have the same requirements and can all be installed in the same way. The blocker executables (pogoBlock, pogoBlockGreedy and pogoBlockGreedy3d), the Abaqus converter (pogoFromAb), and the mesher (pogoMesh) have fewer requirements, since they do not use GPUs or CUDA. They can be installed in the same way, i.e. placed in the same location in the path, without any difficulties.

1.2.1.1 Nvidia GPU drivers

Suitable modern Nvidia GPU drivers should be installed in order to run the solver. If using Linux, the easiest, most stable approach is to use the drivers directly from the Linux distribution. This will ensure that they are properly compatible with the other packages in your OS and will be updated as the OS is. Alternatively, if using Windows, or if the drivers are unavailable from the Linux OS, then you can download and install directly from Nvidia. In Linux, you can check the GPU status (Nvidia System Management Interface) with the command

```
nvidia-smi
```

and there is a similar command on Windows. This should confirm that the GPUs are properly installed.

1.2.1.2 HASP drivers

Pogo uses Sentinel HASP USB dongles to secure the software. On Windows no driver installation should be necessary with the latest version of the HASP, but Linux will require this to be installed. *Note that the drivers should be installed prior to the HASP being inserted!* On Debian/Ubuntu-based systems the necessary .deb file should be included in the installation package. From the directory in which this resides, you can install the HASP driver with

```
sudo dpkg -i --force-architecture aksusbd_7.55-1_i386.deb
```

with the version updated if necessary from version 7.55 depending on the file provided. The package can be uninstalled with

```
sudo dpkg --remove aksusbd
```

Other drivers for other distributions can be downloaded directly from Sentinel, or you can contact the Pogo team who should be able to help, at support@pogo.software.

1.2.1.3 Paths

Your system must be able to find the executables, so either you need to be in the same directory, or the executables must be on the system path. On Linux this could normally be achieved by copying the binaries to a suitable system location, e.g.

```
sudo mkdir -m /usr/local/pogo
sudo cp pogo* /usr/local/pogo/
```

Then putting symbolic links to a location on the path (in this case, in /usr/local/bin)

```
cd /usr/local/bin
sudo ln -s ../pogo/pogo* .
```

Additionally, if you see library errors on Linux, then the library provided on Linux needs to be somewhere on the library path, e.g. /usr/local/lib

```
sudo cp libcudart* /usr/local/lib/
```

1.2.1.4 Running Pogo

With these two packages installed, it should be possible to test Pogo for your system. For testing purposes, the Pogo solver can be run with the command

```
pogoSolve --test
```

This will check the HASP is working and that the license is valid, then continue to perform a CUDA calculation on all the GPUs on the system. If this does not run, then you are likely to have installation issues.

1.2.2 Pogo.Pro

Pogo.Pro has installers provided for Windows; given that the HASP drivers do not need to be installed for Windows, installation is just a case of running the installer. For Linux the HASP drivers must be installed, then the binaries must be placed on the path; both of these processes match the approach used for the Pogo solver. It should be clear if Pogo.Pro has been successfully installed by running it, and this will also highlight if there are any licensing issues with the HASP.

1.2.2.1 Paraview

Pogo.Pro uses Paraview to visualise and animate the field data. The latest version of Paraview can be downloaded from the web location `http://www.paraview.org/download/`; follow the installation instructions for your specific system.

Chapter 2

Getting started quickly

This chapter demonstrates how to quickly get started with Pogo and Pogo.Pro. In the following sections the SlitBlock example is presented, which includes the creation of a simple, two-dimensional, Pogo input file. This is then solved, and the visualisation of that output are presented. New users are advised to replicate this example to get familiarised with the workflow of Pogo, Pogo.Pro and Paraview. Users who are new should also make sure that they have read and are familiar with all the theoretical content of Chapter 3, because this explains many modelling choices necessary in performing FE simulations of waves.

2.1 Quick start - the SlitBlock example

2.1.1 Problem description

In the SlitBlock example, detailed in this section, a two-dimensional steel block with a 1mm wide notch is designed in Pogo.Pro, afterwards simulated using Pogo and, finally, visualised using Pogo.Pro and Paraview.

2.1.1.1 Geometry definition

The geometry of the block is a rectangle, at 200x100 mm. The notch is 1x6 mm and is located at the bottom and in the middle of the block.

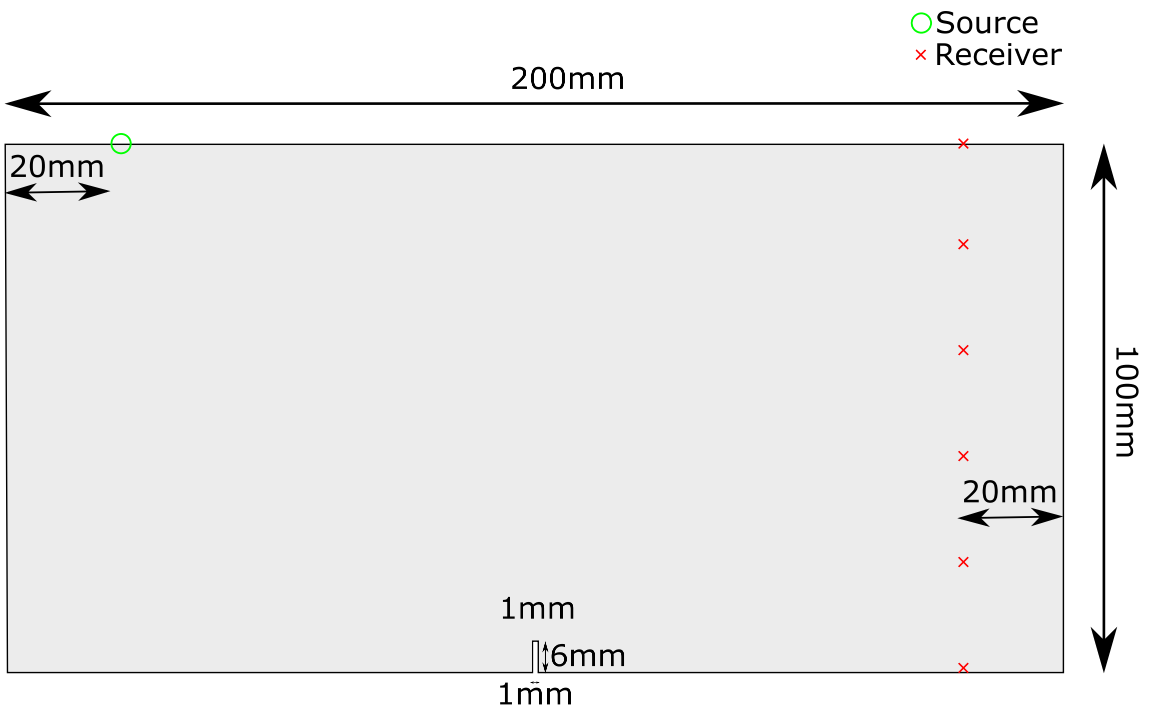


Figure 2.1: The SlitBlock geometry

2.1.1.2 Source and receivers

The setup of the wave source includes a vertical force source on the top surface, 20 mm from the left-hand edge of the block. The source frequency is 2 MHz, with a total duration of 3 cycles and is Hann-window modulated.

A total of six receivers record all the displacements on intervals of 20 mm from the right-hand edge on the top surface.

The geometry and location of the source and receivers are demonstrated in Fig. 2.1.

2.1.1.3 Mesh fineness and simulation time

Twenty elements per wavelength are taken for this problem (see Sect. 3.3.4 for more details on this choice). Also the simulation time should be enough for the wave to bounce off the backwall and on to the receivers.

The wave speed for steel can be evaluated to be 6000 m/s using the formula

$$c_0 = \sqrt{\frac{E\,(1-v)}{\varrho\,(1+v)\,(1-2v)}} \tag{2.1}$$

with the density (ρ), Young's Modulus (E)and Poisson ratio (v) for steel being 7800 kg/m^3, 215.3 GPa and 0.293, respectively. The wavelength is, therefore, 3 mm and the maximum element size should be 0.15 mm.

The distance to travel is 283 mm, so the simulation time should be 0.047 ms plus the signal length (0.00015 ms). A simulation time of 0.06 ms is a good value to include the full expected signal.

2.1.2 Creation of the Pogo input file using Pogo.Pro

To create the SlitBlock input file, please follow these steps:

1. Launch Pogo.Pro:
   - Windows 7/8.x/10: From the Start Menu find and select Pogo.Pro
   - Ubuntu Linux 14.04/16.04: If Pogo.Pro is on the path, just enter 'Pogo.Pro' into the terminal. If not, navigate to the location in the file manager and run from there.

   You will be greeted with the interface shown in Fig. 2.2.
2. In the *Operations Side-Tabs*, select the **[Input File Creation]** side-tab and the **[New 2D Workspace]** from the *Initialisation Buttons*.

   You will be greeted with the main creation workspace, which includes the *ViewPort* and its *Manipulation Toolbar*, the *Workflow Toolbox* and the *Details Panel*, as shown in Fig. 2.3.
3. (optional step) On the **[General Details]** tab of the *Workflow Toolbox* enter a File Name, e.g. **SlitBlock**, and a comment and then press **[Apply]**.
4. Select the **[Geometry Gallery]** tab of the *Workflow Toolbox* and create two rectangles by clicking twice on the **[Rectangle]** option of the available geometries list.

   The *Viewport* now includes two rectangles that overlap*.
5. Select the topmost rectangle by clicking it in the *Viewport* and, in the *Details Panel*, change its width to 1 mm and its height to 6 mm. Then set its y-position to -47 mm.

   The rectangle is now smaller and moved to the lower part of the other rectangle†.

*Tip: The shape colours are appointed randomly; you can always set a selected shape's colour to your liking in the **[Colour Scrollbox]** of the *Details Panel*.

†Tip: You can use **[Shift + LClick]** to select shapes that are "under" visible shapes in the *Viewport*

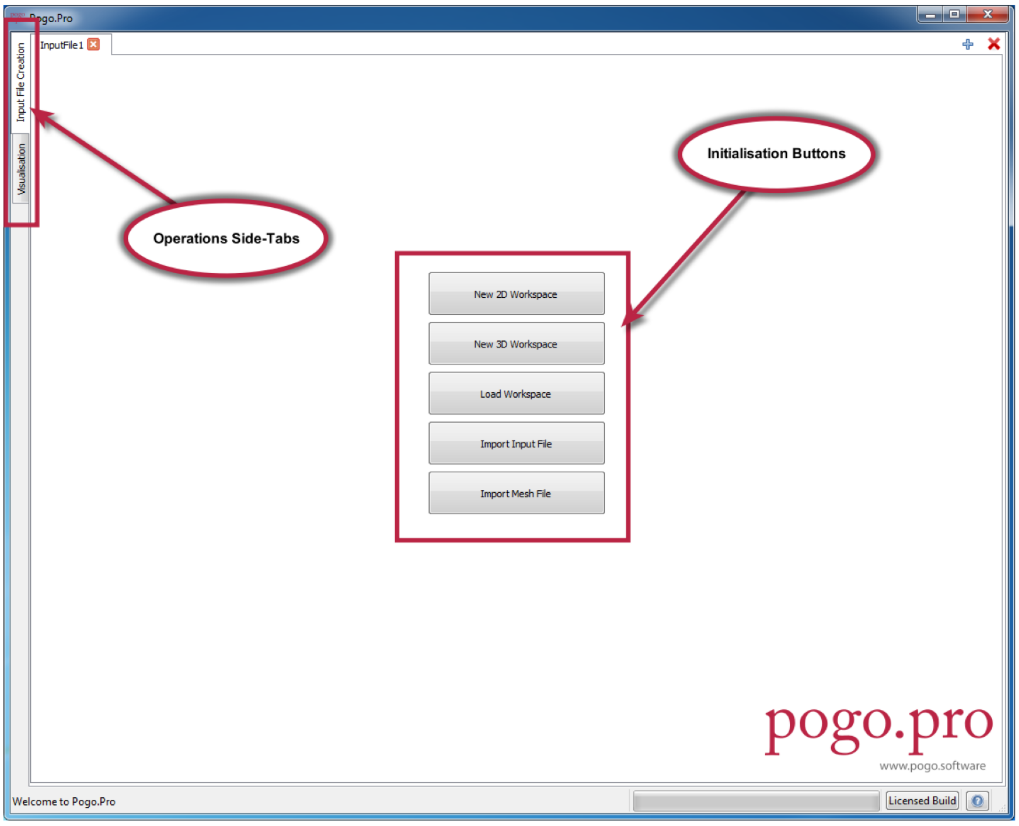


Figure 2.2: The initial Pogo.Pro interface

6. Select the, newly visible, larger rectangle and set its width to 200 mm*.

   The viewport is shown in Fig. 2.4.

7. Select, by clicking, the larger rectangle first and, while holding **[Ctrl]**, the smaller rectangle.

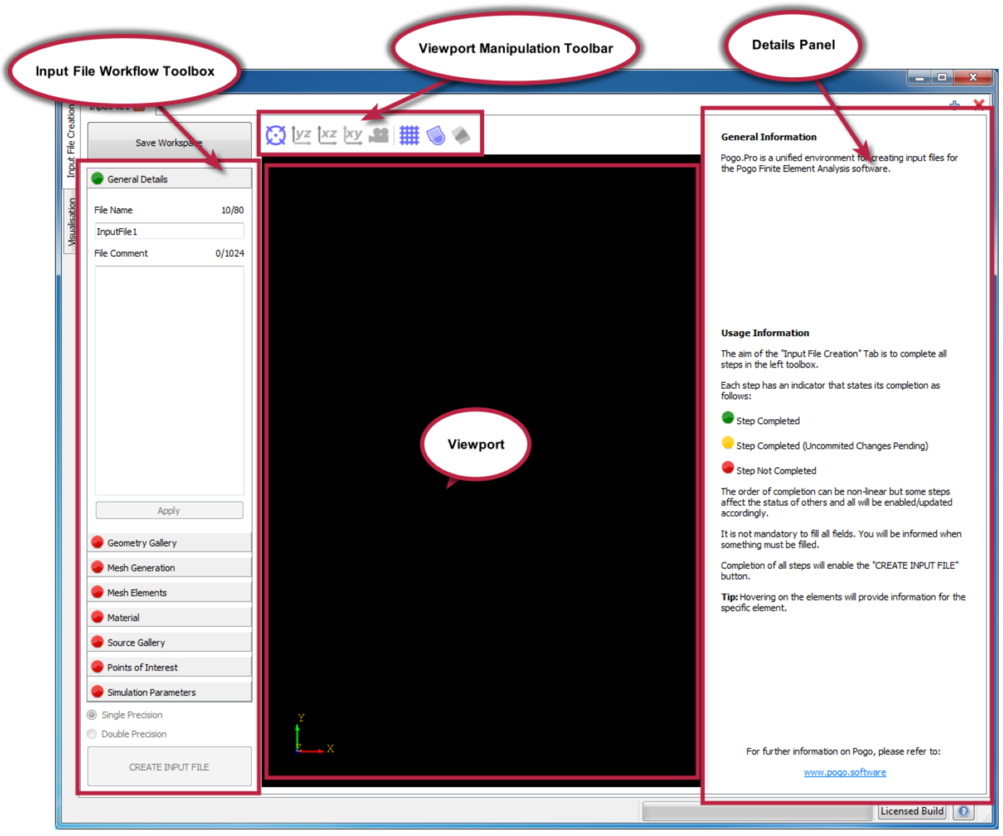
   A new set of options for shape logical operations is now visible in the *Details Panel*, as shown in Fig. 2.5.

8. Press the **[Difference Button]** in the *Details Panel.*

   A new compound geometry is produced; the block with a notch, as shown in Fig. 2.6. Notice how the *Geometry Gallery* indicator has turned to green.

9. Select the **[Mesh Generation]** tab in the *Workflow Toolbox* and enter 0.15 mm in the *Elements->Maximum Size* spinbox.

*Tip:You can press the **[Center Viewport]** button in the *Viewport Manipulation Toolbar* to view the whole geometry



General Information

Pogo.Pro is a unified environment for creating input files for the Pogo Finite Element Analysis software.

Usage Information

The aim of the "Input File Creation" Tab is to complete all steps in the left toolbox.

Each step has an indicator that states its completion as follows:

🟢 Step Completed

🟡 Step Completed (Uncommited Changes Pending)

🔴 Step Not Completed

The order of completion can be non-linear but some steps affect the status of others and all will be enabled/updated accordingly.

It is not mandatory to fill all fields. You will be informed when something must be filled.

Completion of all steps will enable the "CREATE INPUT FILE" button.

Tip: Hovering on the elements will provide information for the specific element.

For further information on Pogo, please refer to:

www.pogo.software

Figure 2.3: The main workspace window

10. Press **[Generate Mesh]** in the *Workflow Toolbox*.

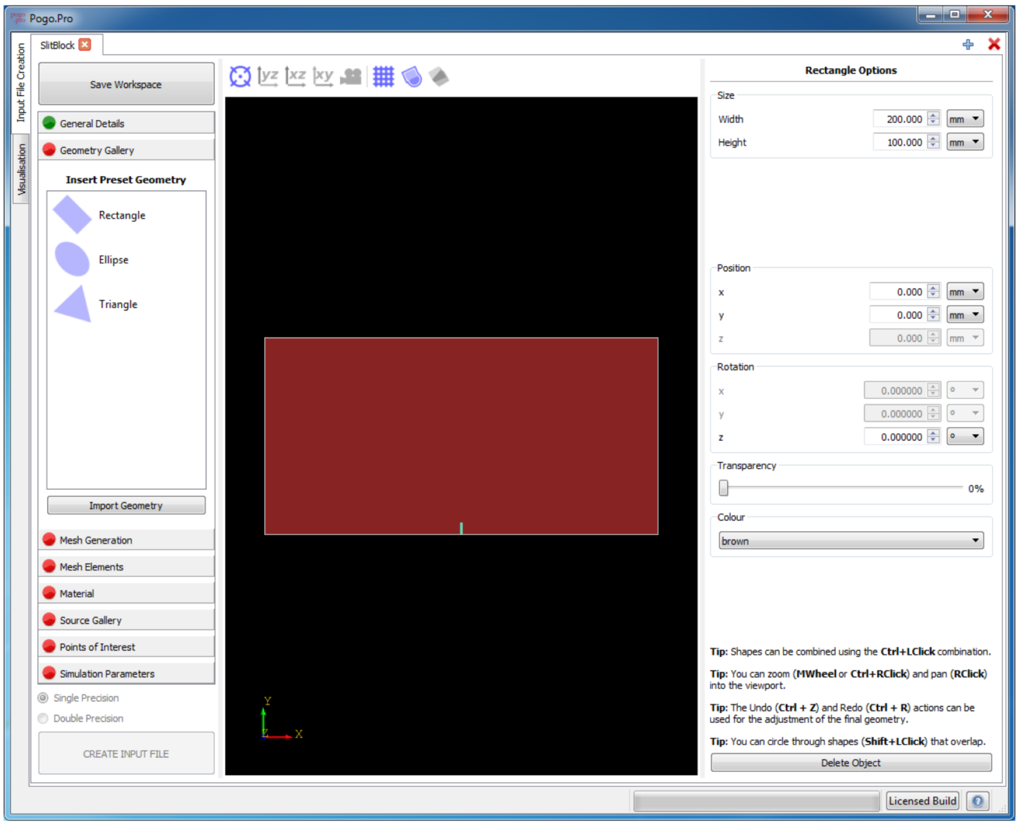
    After some time, in which the mesh is calculated, the geometry appears meshed in the *Viewport*, as shown in Fig. 2.7*.

11. Select **[Mesh Elements]** in the *Workflow Toolbox*. Since the default element type is CPE3, there is no need to change anything.

12. Select **[Material]** in the *Workflow Toolbox*. Since Steel is the default selection, there is no need to change anything; just press **[Apply]**.

13. Select **[Source Gallery]** in the *Workflow Toolbox* and add a new source by pressing the [✚] button.

*Information on the mesh is shown in the *Information* groupbox in the *Details Panel*.



Rectangle Options

Size

Width	200.000	mm
Height	100.000	mm

Position

x	0.000	mm
y	0.000	mm
z	0.000	mm

Rotation

x	0.000000	°
y	0.000000	°
z	0.000000	°

Transparency 0%

Colour brown

Tip: Shapes can be combined using the **Ctrl+LClick** combination.

Tip: You can zoom (**MWheel** or **Ctrl+RClick**) and pan (**RClick**) into the viewport.

Tip: The Undo (**Ctrl + Z**) and Redo (**Ctrl + R**) actions can be used for the adjustment of the final geometry.

Tip: You can circle through shapes (**Shift+LClick**) that overlap.

Figure 2.4: The two rectangle shapes, with the smaller selected, in the *Viewport*.

14. Select **[Source1]** in the *Workflow Toolbox* and set *Time Parameters->Frequency* to 2 MHz and *Time Parameters->Cycles* to 3 in the *Details Panel*.

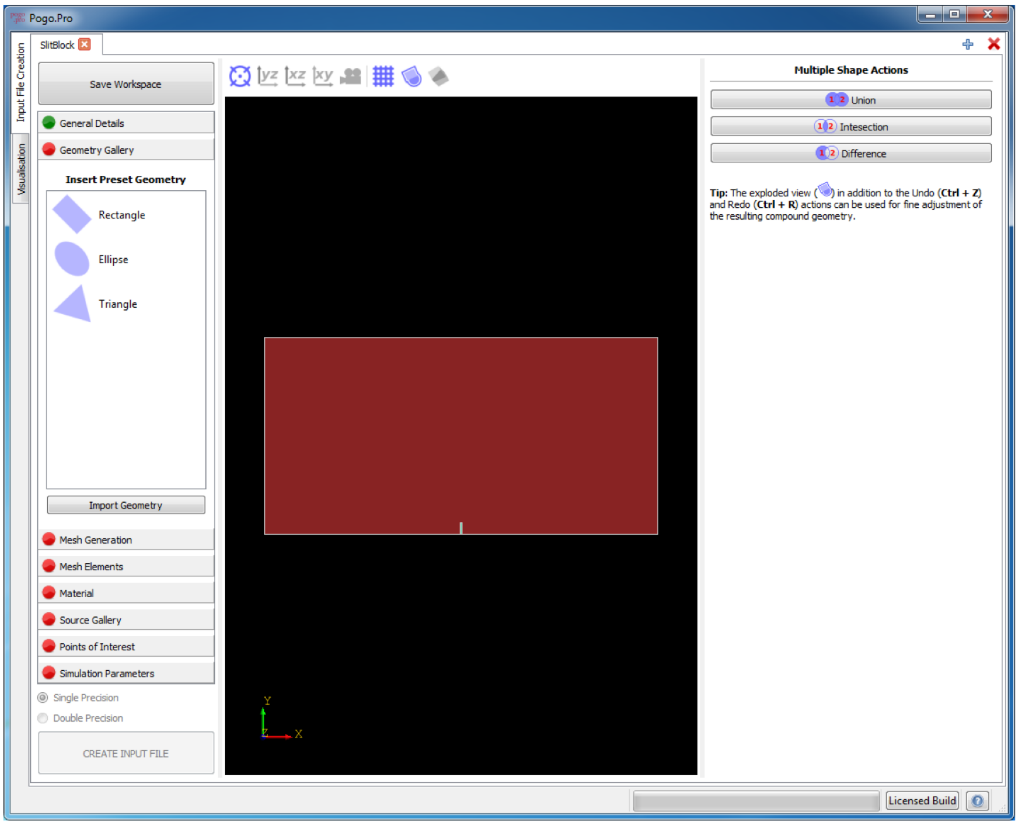
    The Details Panel provides a preview of the new source that was just created; see Fig. 2.8.

15. Select **[Points of Interest]** in the *Workflow Toolbox* and press the [✚] button.

16. Select **[Point1]** and set in the *Details Panel* its *Position->x* to -80 mm and *Position->y* to 50 mm and *Source Degree of Freedom->y* to **[Source1]**.

    Now the vertical degree of freedom is set as a Source1 emission point. The point has moved to its requested position and is indicated by a green circle.

17. Add six new Points of Interest in the *Workflow Toolbox* by setting the counter to 6 next to the [✚] button and pressing it.



Tip: The exploded view ( ) in addition to the Undo (**Ctrl + Z**) and Redo (**Ctrl + R**) actions can be used for fine adjustment of the resulting compound geometry.

Figure 2.5: The shape logical operation options.

18. Select Points 2 to 7 on the Points of Interest list by using **[Shift+LClick]** on each point or **[Shift+Down]** keys.

19. Without deselecting the 6 points set in the *Details Panel* the value *Position->x* to 80 mm and *Properties->Type* to **Receiver**.

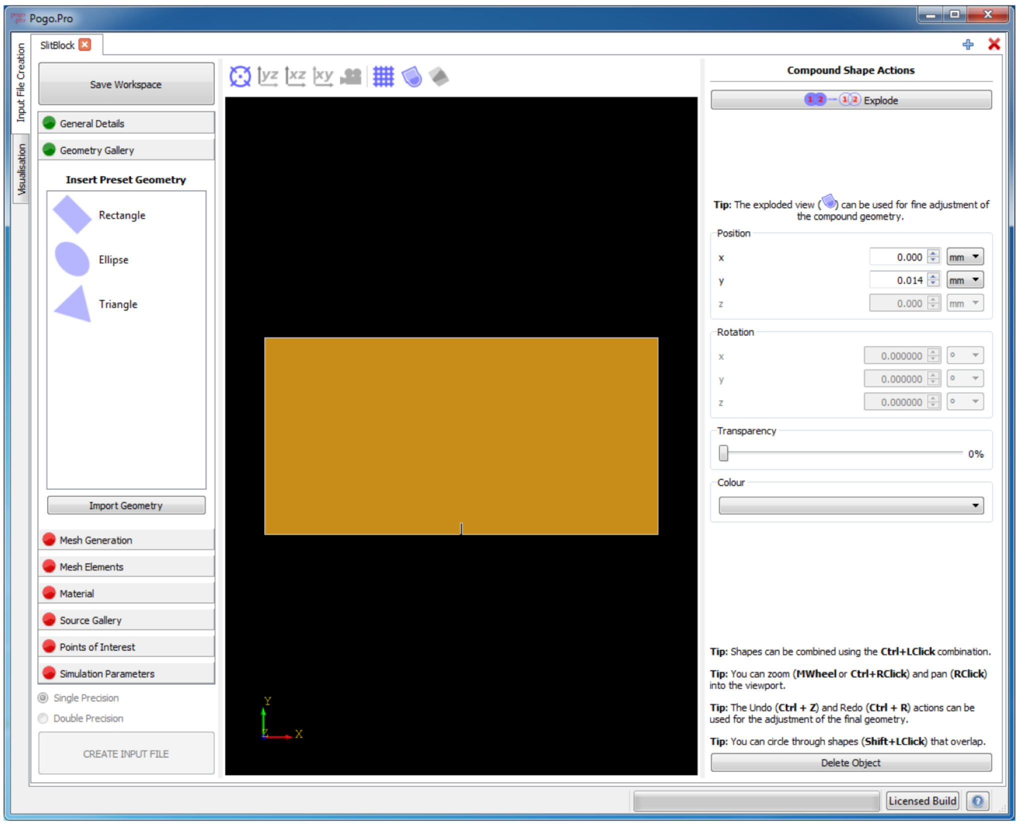
20. Select Points 2 to 7 individually in the *Workflow Toolbox* to change in the *Details Panel* their *Position->y* to 50, 30, 10, -10, -30, -50 mm, respectively.

    The viewport now shows one source and all the receiver points, as shown in Fig. 2.9.

21. Press **[Adapt Points to Mesh]** in the *Workflow Toolbox*.

22. Select **[Simulation Parameters]** in the *Workflow Toolbox* and set *Total Time* to 0.06 ms and press **[Apply]**.

    All the indicators should now be green and the SlitBlock Pogo input file can now be created in either Single or Double precision



Tip: The exploded view ( ) can be used for fine adjustment of the compound geometry.

Position
x 0.000 mm
y 0.014 mm
z 0.000 mm

Rotation
x 0.000000 °
y 0.000000 °
z 0.000000 °

Transparency 0%

Colour

Tip: Shapes can be combined using the **Ctrl+LClick** combination.

Tip: You can zoom (**MWheel** or **Ctrl+RClick**) and pan (**RClick**) into the viewport.

Tip: The Undo (**Ctrl + Z**) and Redo (**Ctrl + R**) actions can be used for the adjustment of the final geometry.

Tip: You can circle through shapes (**Shift+LClick**) that overlap.

Figure 2.6: The compound geometry in the *Viewport*.

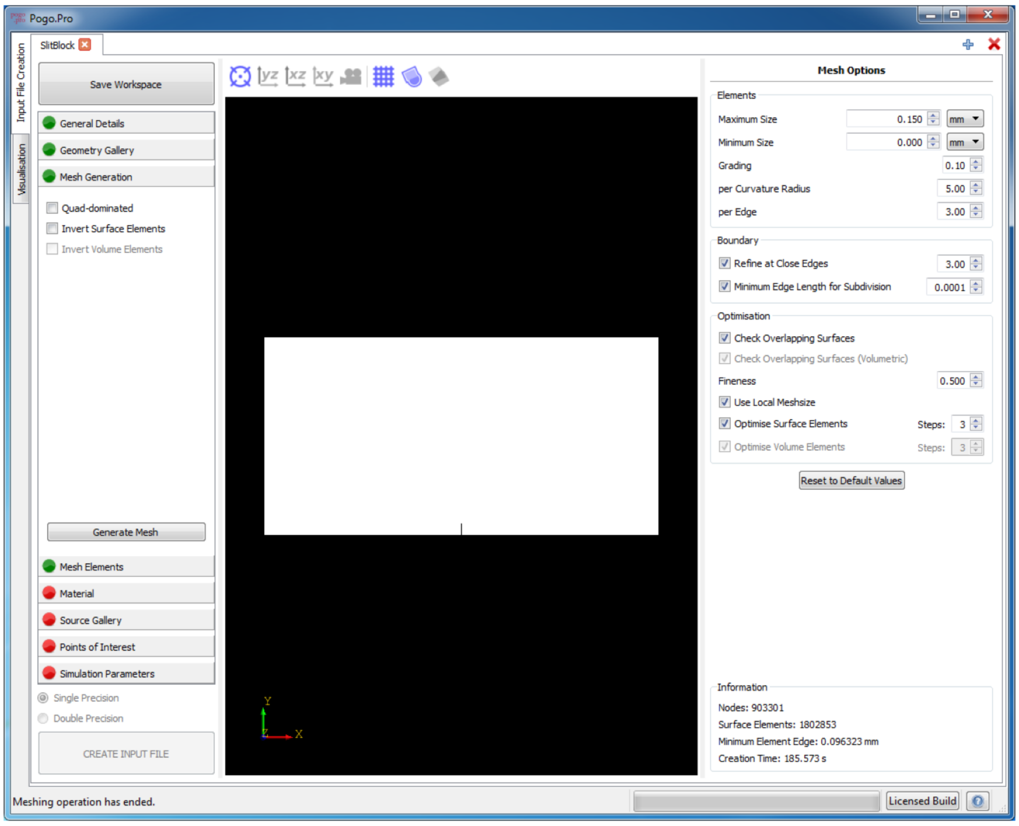
23. Press the **[CREATE INPUT FILE]** button in the *Workflow Toolbox* to create the file in single precision*.

    The file *SlitBlock.pogo-inp* is now created in the location you have chosen.

2.1.3 Simulation using Pogo

To perform the Pogo simulation, you will need a server or workstation with a functional Pogo installation. Pogo v1.3 and upwards will be able to run the simulation successfully. To run the Pogo simulation in this server, named *pogoserver* here for demonstration purposes, please perform the following steps:

*Tip: It would be a good practice to rename the input file to slitBlock-double in *General Details->File Name* for double precision files, in case you export with both precisions.



Quad-dominated ☐
Invert Surface Elements ☐
Invert Volume Elements ☐

Single Precision ●
Double Precision ○

Mesh Options

Elements
Maximum Size 0.150 mm
Minimum Size 0.000 mm
Grading 0.10
per Curvature Radius 5.00
per Edge 3.00

Boundary
☑ Refine at Close Edges 3.00
☑ Minimum Edge Length for Subdivision 0.0001

Optimisation
☑ Check Overlapping Surfaces
☑ Check Overlapping Surfaces (Volumetric)
Fineness 0.500
☑ Use Local Meshsize
☑ Optimise Surface Elements Steps: 3
☑ Optimise Volume Elements Steps: 3

Information
Nodes: 903301
Surface Elements: 1802853
Minimum Element Edge: 0.096323 mm
Creation Time: 185.573 s

Meshing operation has ended.

Figure 2.7: The meshed geometry in the *Viewport*.

1. Copy *SlitBlock.pogo-inp* to a folder of your choosing, e.g. *SlitBlockExample*, in *pogoserver* using a terminal or SFTP software such as WinSCP or FileZilla.

2. Using a terminal window (or Putty), navigate using SSH to the server folder you have put the input file in:

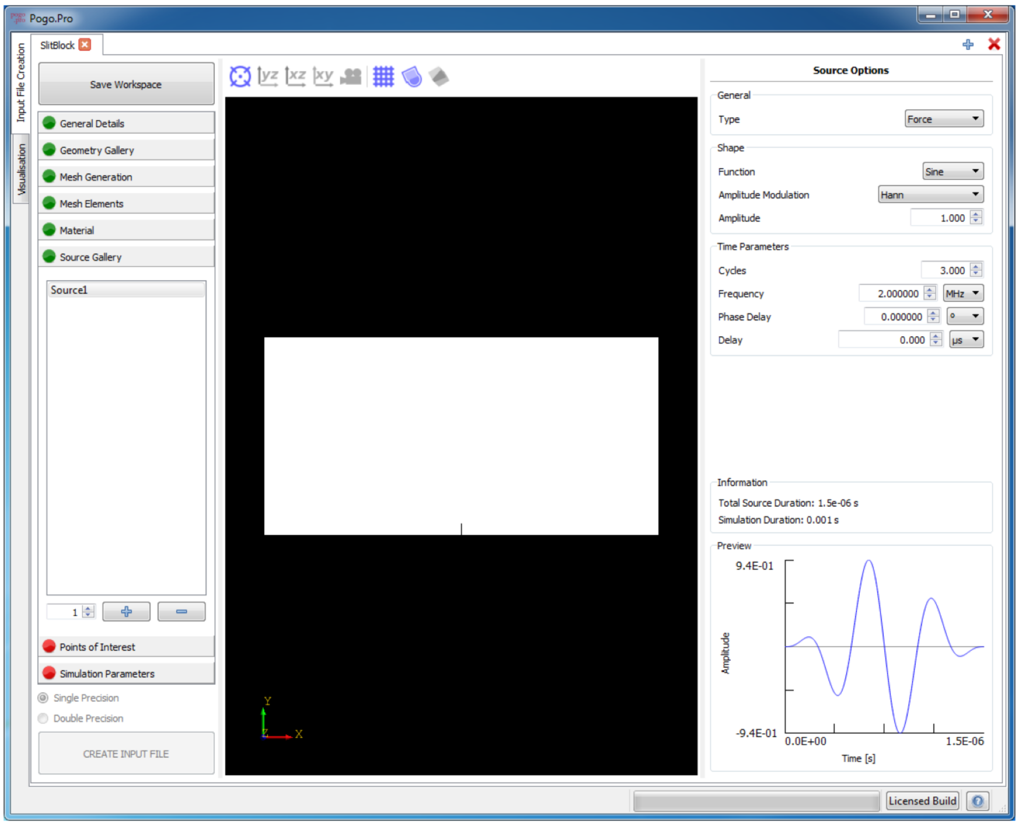
```
ssh username@pogoserver
cd ~/SlitBlockExample
```

3. Perform the blocking of the mesh contained in the input file:

```
pogoBlock SlitBlock
```

4. Perform the simulation:

```
pogoSolve SlitBlock
```



Source Options

General
Type: Force

Shape
Function: Sine
Amplitude Modulation: Hann
Amplitude: 1.000

Time Parameters
Cycles: 3.000
Frequency: 2.000000 MHz
Phase Delay: 0.000000 °
Delay: 0.000 µs

Information
Total Source Duration: 1.5e-06 s
Simulation Duration: 0.001 s

Figure 2.8: The Source preview.

5. Move the input (*.pogo-inp) and resulting output files (*.pogo-field, *.pogo-hist) to the workstation running Pogo.Pro using the SSH software of your choice (Terminal/WinSCP/-FileZilla).

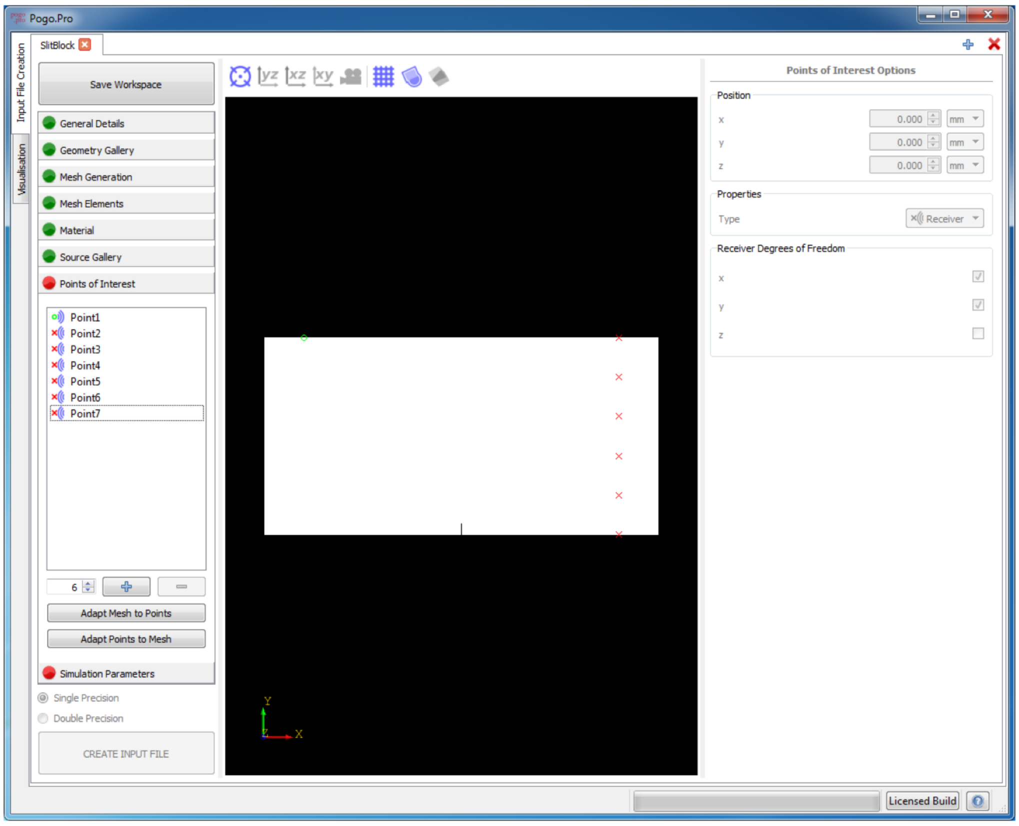
For a better understanding of the aforementioned commands, please refer to Appendix A.

2.1.4 Exporting the field and time traces using Pogo.Pro

1. In Pogo.Pro, select the **[Visualisation]** in the *Operations Side Tabs*.

   You will be greeted with the interface shown in Fig. 2.10.

2. Select **[Load Input File]** and select *slitBlock.pogo-inp* in the location in your workstation where you have moved the files after the Pogo simulation, described in the previous section.



Position
x 0.000 mm
y 0.000 mm
z 0.000 mm
Properties
Type Receiver
Receiver Degrees of Freedom
x ☑
y ☑
z ☐

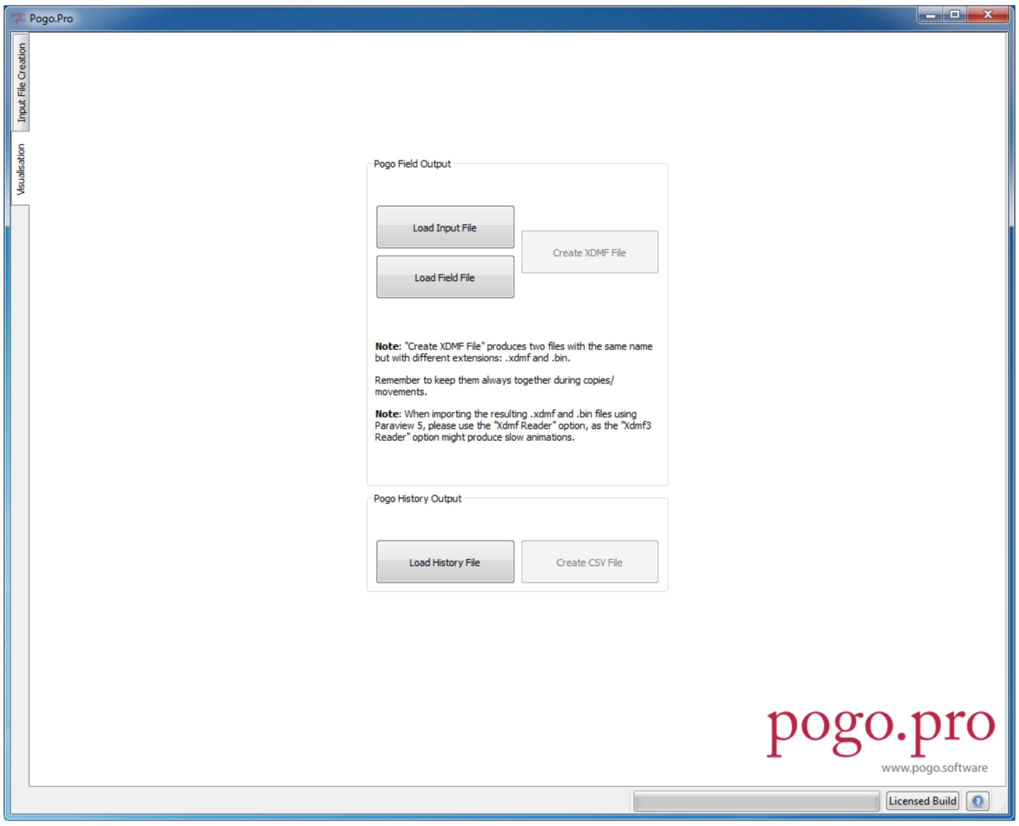
Figure 2.9: The points of interest in the meshed geometry.

3. Select **[Load Field File]** and select *slitBlock.pogo-field* in the location in your workstation where you have moved the files after the Pogo simulation, described in the previous section.

4. If the block and field files match, press the the newly-activated **[Create XDMF File]** and select the location where you want the XDMF file to be saved.

5. Select **[Load History File]** and select *slitBlock.pogo-hist* in the location in your workstation where you have moved the files after the Pogo simulation, described in the previous section.

6. Select the newly-activated button **[Create CSV File]** and select the location where you want the CSV file to be saved.



Note: "Create XDMF File" produces two files with the same name but with different extensions: .xdmf and .bin.

Remember to keep them always together during copies/ movements.

Note: When importing the resulting .xdmf and .bin files using Paraview 5, please use the "Xdmf Reader" option, as the "Xdmf3 Reader" option might produce slow animations.

Figure 2.10: The visualisation interface of Pogo.Pro.

2.1.5 Field and time trace visualisation using Paraview

For the visualisation of the field and time traces, either MATLAB or Paraview can be currently used. You can download the newest Paraview version from the web location http://www.paraview.org/download/.

2.1.5.1 Field visualisation

The easiest way to view the field is to carry on with the following steps:

1. Launch Paraview:
   - Windows 7/8.x/10: From the Start Menu find and select Paraview

- Ubuntu Linux 14.04/16.04: If Paraview is on the path, simply open a terminal and enter ‘paraview’. If not, navigate to the installation directory in the file manager and double click.

2. Select **[Open File]** in the menu and select *SlitBlock.xdmf* that was exported from the previous section.
3. On the popup dialog select *Xdmf Reader* and select *OK*.
4. In the *Pipeline Browser* select the newly-loaded file and press **[Apply]** on *Properties*.
5. On the main toolbar, change **[Solid Color]** to **[Node Centered Values]**.
6. Next to the previous combobox, select either **[Amplitude]** or **[X]**/**[Y]**/**[Z]**, depending on what you need to visualise.
7. Finally, on the top toolbar, use the playback buttons to view the field visualisation in frames or as a movie.

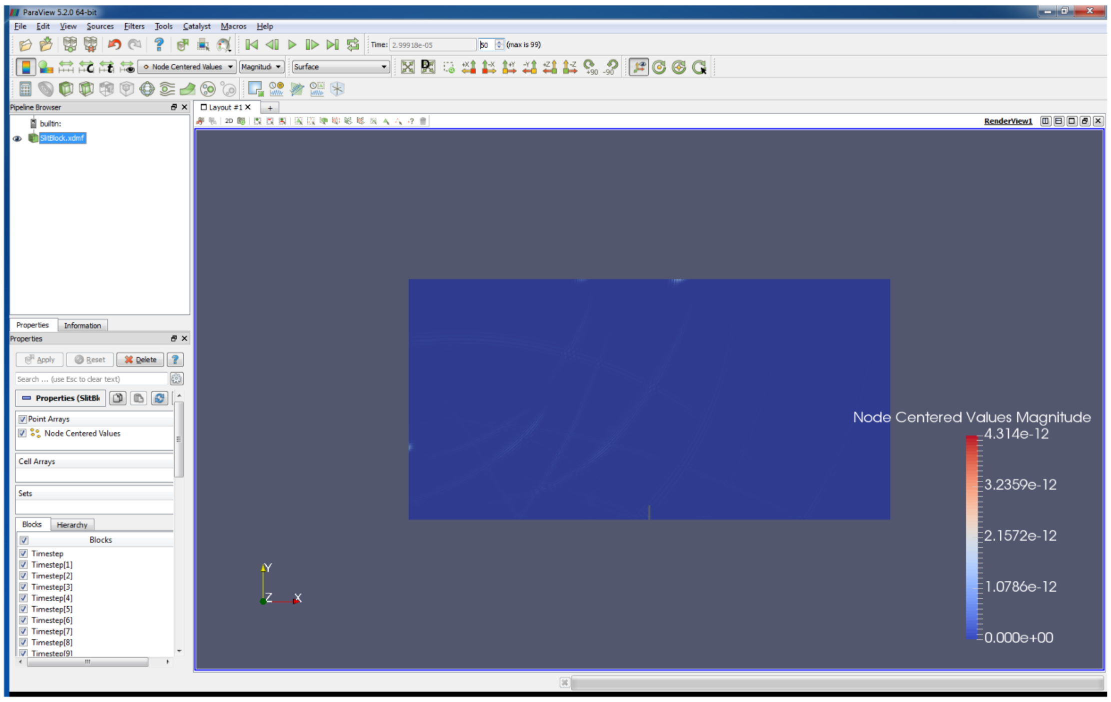
   Fig. 2.11 shows the visualisation at half the simulation time.

2.1.5.2 Time trace visualisation

The easiest way to view the time traces is to carry on with the following steps:

1. Launch Paraview:
   - Windows 7/8.x/10: From the Start Menu find and select Pogo.Pro
   - Ubuntu Linux 14.04/16.04:
2. Select **[Open File]** in the menu and select *SlitBlock.csv* that was exported from the previous section.
3. In the *Pipeline Browser* select the newly-loaded file and press **[Apply]** on *Properties*.
4. Select in the Paraview menu *Filters->Alphabetical->Plot Data*. A new plot entry will appear in the *Pipeline Browser*.
5. In the *Pipeline Browser* select the newly-loaded plot and press **[Apply]** on *Properties*.

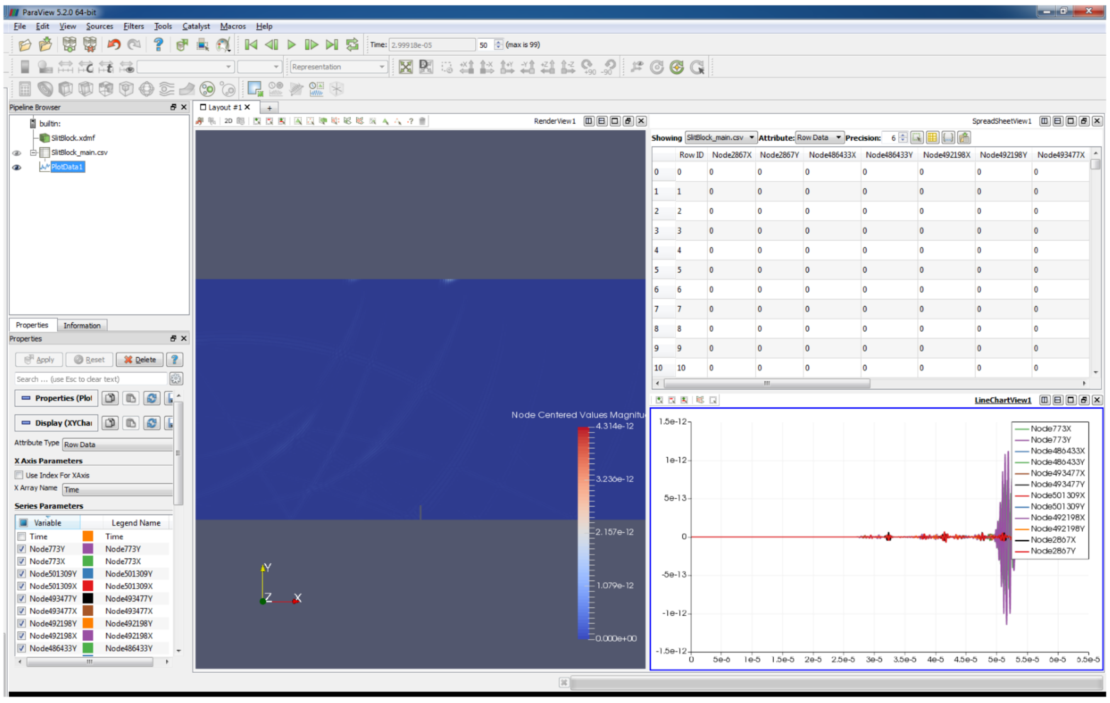
   Fig. 2.12 shows the visualisation at half the simulation time.



Time: 2.99918e-05 50 (max is 99)

Node Centered Values Magnitude Surface

Figure 2.11: The visualisation of the field in Paraview.



Showing SlitBlock_main.csv Attribute: Row Data Precision: 6

	Row ID	Node2867X	Node2867Y	Node486433X	Node486433Y	Node492198X	Node492198Y	Node493477X
0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0
3	3	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0	0
5	5	0	0	0	0	0	0	0
6	6	0	0	0	0	0	0	0
7	7	0	0	0	0	0	0	0
8	8	0	0	0	0	0	0	0
9	9	0	0	0	0	0	0	0
10	10	0	0	0	0	0	0	0

Attribute Type: Row Data

X Axis Parameters

☐ Use Index For XAxis

X Array Name: Time

Series Parameters

Variable		Legend Name
☐ Time	🟧	Time
✓ Node773Y	🟪	Node773Y
✓ Node773X	🟩	Node773X
✓ Node501309Y	🟦	Node501309Y
✓ Node501309X	🟥	Node501309X
✓ Node493477Y	⬛	Node493477Y
✓ Node493477X	🟫	Node493477X
✓ Node492198Y	🟧	Node492198Y
✓ Node492198X	🟪	Node492198X
✓ Node486433Y	🟩	Node486433Y

Figure 2.12: The visualisation of the time traces in Paraview.

Chapter 3

Acoustics and finite element essentials

3.1 Introduction

The intention of this section is to provide a quick reference for the essential fundamentals needed for people to make use of Pogo. It aims to provide a reasonably concise source of information which can be easily referred to. There are two aspects which will be covered in this section, acoustics and the finite element method. While it aims to be as comprehensive as possible, it would take up far too much space to fully cover all the knowledge needed for everything which can be achieved in Pogo here; this section is not intended to replace a good textbook or a course relevant to the area, and it is expected that Pogo users will independently learn sufficient background knowledge relevant to their applications.

3.2 Acoustics

3.2.1 Wave physics fundamentals

In order for a wave to be able to exist and propagate there needs to be an interplay between some form of 'righting' elasticity term and an inertial term. This is true across both physical and EM waves, although the latter will not be considered here. Energy stored as elastic potential energy will be converted into kinetic energy, which will then be converted back to elastic potential energy again; this behaviour will be described by $F = ma$ (Newton's second law) combined with some form of material behaviour equation (a constitutive equation). The traditional wave equation is described in 1D as

$$\frac{\partial_2 p}{\partial x^2} - \frac{1}{c^2}\frac{\partial_2 p}{\partial t^2} = 0 \tag{3.1}$$

where p is pressure, c represents the homogeneous wave speed and t and x are time and spatial dimensions respectively. It should be noted that the same equations are valid for vibrations problems too, except that the boundary conditions are different such that the wave energy is trapped in a small region. For elastodynamic problems, the wave equation becomes more complicated. Most notably, shear components become significant and rather than having a single degree of freedom (pressure), there will be a displacement value in each dimension. The full vector equation can be expressed in terms of displacement $\boldsymbol{u}$ as

$$\rho\, \ddot{\boldsymbol{u}} = \boldsymbol{\nabla\lambda}\,(\boldsymbol{\nabla.u}) + \boldsymbol{\nabla\mu}.\left[\boldsymbol{\nabla u} + (\boldsymbol{\nabla u})^{\boldsymbol{T}}\right] + (\boldsymbol{\lambda}+\boldsymbol{2\mu})\,\boldsymbol{\nabla}\,(\boldsymbol{\nabla.u}) - \boldsymbol{\mu\nabla}\times\boldsymbol{\nabla}\times\boldsymbol{u}$$

where the elastic constants are expressed in terms of the Lamé parameters λ and μ. By making some simplifications, within a homogeneous region, it is possible to substitute a purely divergent (i.e. curl free, $\nabla\times\boldsymbol{u} = \boldsymbol{0}$) field into this equation, which will result in the standard wave equation (3.1) with the wave speed set to that of the longitudinal wave. It is also possible to do the same for shear waves, by substituting a divergence-free ($\nabla.\boldsymbol{u} = 0$) field into the domain. So there are significant similarities between the scalar wave equation and the elastodynamic wave equation, so much of the theory overlaps. One significant difference is that the shear wave typically has a much lower wave speed than that of the longitudinal wave, and hence they have very different wavelengths which require different levels of discretisation.

3.2.1.1 Solving the wave equation

The wave equation can be solved in general for a homogeneous medium, by defining pressure as

$$p(x,t) = A\cos(2\pi ft + \phi - kx) \tag{3.2}$$

where A is an amplitude value, f is the frequency, ϕ is a phase shift of the wave and $k = 2\pi f/c$ is the wavenumber in the medium (it is possible to see that this is a solution by substitution into eq. (3.1)). This sinusoidal behaviour is important as will be seen later.

3.2.1.2 Complex numbers

There is an alternative way of representing the solution, using complex numbers. In complex numbers, $i = \sqrt{-1}$, which therefore enables a solution

$$p = Re\left[Ae^{ikx - i2\pi ft}\right] \tag{3.3}$$

where A is now a complex amplitude value, incorporating any phase shift ϕ too, and $Re()$ represents the real part of the argument. This exploits the rule that

$$e^{i\phi} = \cos\phi + i\sin\phi. \tag{3.4}$$

Complex numbers have a number of advantages over eq. (3.2) - in part it is clearly more concise to include the wave phase and amplitude in a single term. It is also very easy to account for phase shifts by multiplication; whereas previously, it was necessary to adjust a term within the cosine function, now it is possible to simply multiple the complex amplitude A by another complex term.

3.2.1.3 Wave speed, wavelength

The longitudinal wave speed for a solid medium can be calculated as

$$c_0 = \sqrt{\frac{E\left(1-v\right)}{\varrho\left(1+v\right)\left(1-2v\right)}} \tag{3.5}$$

while the shear wave speed is

$$c_{sh} = \sqrt{\frac{E}{2\varrho\left(1+v\right)}} \tag{3.6}$$

where E is the Young's modulus, ν is the Poisson's ratio and ρ is the density of the medium. For a particular frequency, the wavelength can therefore be calculated as

$$\lambda = \frac{c}{f} \tag{3.7}$$

where c can be replaced with the longitudinal or shear value as necessary.

3.2.1.4 The superposition principle

The wave equation above is linear. This means that if two valid wave fields exist, they can be summed and the resulting wave field will also satisfy the wave equation[*]. Given the solution above, this means that many cosine terms with different phases, frequencies and amplitudes can be summed together, i.e.

$$p(x,t) = \sum A\cos\left(2\pi ft + \phi - kx\right) \tag{3.8}$$

or, using complex numbers

$$p(x,t) = Re\left(\sum Ae^{ikx-i2\pi ft}\right) \tag{3.9}$$

to generate a full solution.

[*]Note that non-linear behaviour is not modelled in Pogo, so is not considered in this document.

3.2.1.5 Fourier transforms

A common approach is to consider harmonic solutions to the problem, i.e. look at a single frequency at a time. In this case the harmonic field can be written as

$$\phi\left(x,f\right) = A(f)e^{ikx}$$

then

$$p\left(x,t\right) = Re\left[\sum \phi\left(x,f\right)e^{-i2\pi ft}\right]$$

which means that p is just a Fourier transform of ϕ. Alternatively it is possible to directly Fourier transform the wave equation (3.1), giving

$$\frac{\partial_2 \phi}{\partial x^2} + k^2\phi = 0 \tag{3.10}$$

which is a 1D version of the Helmholtz equation. The use of Fourier transforms is very important across wave propagation in a number of areas, including signal processing and filtering. Much of the behaviour and errors which arise from discretisation of the wave equation are related to the wavelength of the wave; since the majority of pulses are broadband, Fourier theory is critical to understanding each separate frequency component.

3.2.1.6 Reflection, refraction and mode conversion

When a wave interacts with a boundary, it must satisfy the boundary conditions at that location. For a 'sound hard' boundary, the displacement (and hence velocity and acceleration) will be fixed to zero. For a 'sound soft' boundary, the pressure (or stress) will be zero. For displacement-based FE models, the default boundary would be a sound soft stress-free boundary. Alternatively, there can be a penetrable boundary. In this case the wave will be transmitted into the adjacent medium, which will typically have different material properties than the first medium. At the boundary the stresses and displacements must match. As a result of solving the wave equation for such a boundary, there will be reflections coming from the surface and refractions of the wave travelling into the other medium. In general for an elastic wave there will be mode conversion, so that, for example, a longitudinal wave reflecting will produce reflections and refractions of both longitudinal and shear waves. The angles of these can be calculated via Snell's law.

3.2.1.7 Diffraction

The previous section discussed interactions with an infinite planar interface. Many interactions occur at smaller scales, closer to the wavelength of the object. In this case, the waves will undergo diffraction. Diffraction can be seen physically by the fact that we can hear sound from, say the other side of a car - we do not require 'line of sight', because the wavelength of the sound wave is comparable to the size of the scatterer.

3.2.2 Transducer behaviour

The superposition principle has important consequences for wave physics, which are quite relevant for modelling real world behaviour. A standard piezo-electric transducer can typically be considered to behave as a piston source, i.e. a line (or area in 3D) along which a pressure with uniform amplitude and phase acts. This can be modelled by adding multiple sources in a line with the same input signal applied to all of them. Provided that the sources are sufficiently densely spaced (in FE, one source on every node along the line is generally necessary) then this will behave like the piston source. Other transducers (e.g. shear transducers, EMATs) can also be modelled via a distribution of point sources.

3.2.2.1 Reciprocity

Linear ultrasonic models should obey the principle of reciprocity. This means that if exciting in direction A at position B, then measuring in direction C at position D, the signal will be the same as if exciting in direction C at position D and measuring in direction A at position B.

This can be extended beyond individual point sources. A source can be made from superposing multiple sources at different locations in different directions together. An equivalent receiver would be to make measurements at the same locations in the same directions and sum; this is how the majority of transducers work in transmission and reception. This source could produce a wavefield measured in direction C at position D, then by reciprocity, if a wavefield was excited at D in direction C, the measured signal would be the same. It terms of modelling transducers, this means that receiver transducers can be modelled in the same way as sources; dofGroups, described in Sect. 4.3.1, can help with this.

3.2.3 Input signals

One aspect of Fourier theory is that a signal which is finite in time cannot be finite in the frequency domain, and vice versa. For the mathematically inclined, this phenomenon is known as the Paley–Wiener theorem. This is inconvenient; typically we desire a signal which is of limited bandwidth (i.e. finite in the frequency domain) whilst also being finite in time (so our simulations are not infinite). We can approximate this behaviour, however, with windowing functions. A windowing function is multiplied by an underlying sinusoid to give a time-limited signal. This window will define the frequency spectrum, and by carefully selecting such a window it is possible to minimise the amount of energy appearing in the frequencies away from the main bandwidth.

A very commonly used window is the Hann window, effectively a raised cosine function, given by

$$H(t) = \begin{cases} \frac{1-\cos(2\pi ft/nCyc)}{2} & t < nCyc/f \\ 0 & otherwise \end{cases}$$

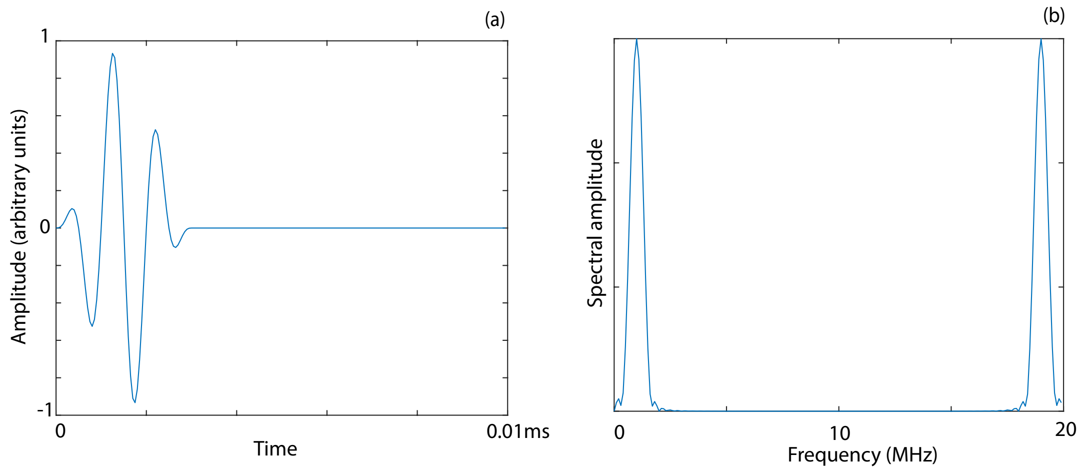


Figure 3.1: The frequency spectrum of a 1 MHz signal, windowed for 3 cycles with a Hann window. (a) shows the signal itself, and (b) shows the spectrum.

where $nCyc$ is the length of the window, in number of cycles of the signal, t represents time and f is the frequency of the underlying signal. This window is named after Julius von Hann, and is sometimes known as the 'Hanning' window, although this name is likely to have arisen through its similarity with the similar 'Hamming' window, and this continues to cause confusion. An example signal windowed by the Hann function is shown in Fig. 3.1(a), along with the spectrum in Fig. 3.1(b). It can be seen that the majority of the energy is concentrated in the central lobe around 1MHz, with small side lobes spreading either side. These never drop completely to zero, but can be neglected far enough from the central lobe. We also see a mirror of the main lobe around 19MHz. The cause of this is the Nyquist sampling limit; users are advised to find a good reference on signal processing to read more about this. There are many different types of window, some with finer control over the size and shape of the side lobes, but essentially the features are similar to those seen with the Hann window.

3.3 Finite element method

The wave equation represents a continuum and in order to solve, it is necessary to discretise the equations. There are many different ways of achieving this, including finite difference (FD) and finite element (FE). While the focus of Pogo is on finite elements, the similarities in behaviour between FE and FD make it worthwhile to include a brief discussion of FD, given it is conceptually easier to understand.

3.3.1 Finite difference

The wave equation will be considered here. Like many continuum equations it contains derivatives of properties with respect to both time and space. A first order derivative can be approximated between two points x_n and x_{n+1} by

$$\frac{\partial p}{\partial x} \approx \frac{p\left(x_{n+1}\right) - p\left(x_n\right)}{x_{n+1} - x_n}. \tag{3.11}$$

Effectively this assumes a constant gradient between the two points and calculates its slope.

The finite difference approach is to define variables at nodal locations (at x_n), often on a uniform grid, and replace the continuous derivatives in the equation with their finite difference approximations. For the wave equation (3.1), this becomes

$$\frac{p_{n+1}^t - 2p_n^t + p_{n-1}^t}{\delta x^2} - \frac{1}{c_n^2}\frac{p_n^{t+1} - 2p_n^t + p_n^{t-1}}{\delta t^2} = 0 \tag{3.12}$$

where superscript is used to indicate the values at particular time step and subscript for values at a particular node. Rearranging gives

$$p_n^{t+1} = c_n \delta t^2 \frac{p_{n+1}^t - 2p_n^t + p_{n-1}^t}{\delta x^2} + 2p_n^t - p_n^{t-1}. \tag{3.13}$$

The equation written in this way has a key advantage for numerical solution; it is an explicit equation. There is a single unknown on the left hand side, p_n^{t+1}, the nodal pressure value at the next time step. All the values on the right hand side are from the current or previous time step, and are therefore known. Therefore no matrix inversions need to be performed; the calculation can be done with simple arithmetic operators. Additionally, the equation is very parallelisable. The values for each node at the $t + 1$ timestep can be calculated independently of each other because there are no interdependencies. This calculation approach is very lightweight and well suited to GPUs.

3.3.2 Finite element

The finite element method is conceptually very similar, certainly from a user's perspective. In the FE method, nodes are defined and elements join these up. Within each element, shape functions describe how the field behaves between the nodes - in the case of Pogo this is the displacement field. These shape functions are typically polynomial, and first order for all the elements within Pogo. Discretised equations can be derived in terms of nodal values by finding a solution which is a best fit for the underlying equations across each element. Readers are referred to [2] for a full description of how this is achieved.

While FE is more complex to implement, it has a number of advantages over FD. Since the formulation is derived for an arbitrary element size and shape, it is straightforward for the approach to be applied to non uniform, free meshes, whereas FD is typically restricted to uniform grids. However, there are many similarities and much of FD behaviour is also seen in FE, and therefore FD can help to understand the issues.

3.3.2.1 Explicit time domain FE

The elastodynamic equation, having been discretised in the spatial domain, can be written as

$$Ku + C\dot{u} + M\ddot{u} = F \tag{3.14}$$

where u is the vector of nodal displacement values, K, C and M are matrices containing stiffness, damping and mass coefficients respectively, and F is the source vector. Dots are used to indicate time derivatives. These time derivatives must also be discretised; one option is a finite element formulation, but instead Pogo uses FD in time. By taking lumped diagonal C and M matrices, an explicit formulation can be obtained along the lines of eq. (3.13). These full equations are outlined in [1].

3.3.2.2 Solution

As with the finite difference approach, this can be solved by stepping through in time, at each increment calculating the nodal values for the next increment. For linear problems K, C and M remain constant throughout, while F is generally zero except at particular defined source nodes, at which the source is applied.

3.3.2.3 Degrees of freedom

The phrase 'degree of freedom' (often shortened to 'DOF') is common in numerical methods, including FE. For a 3D displacement model, there will be 3 displacements at each node, while for acoustics (for example) there will just be pressure, which is a scalar, giving 1 degree of freedom per node. The model is solved for all the degrees of freedom at all nodes; essentially this corresponds to the number of variables in the solution. Sources will be applied to, and measurements taken from, particular degrees of freedom rather than nodes, so it is important to understand the meaning of this phrase.

3.3.3 Boundary conditions

The default FE boundary condition is a free boundary, i.e. sound soft. This corresponds to being connected to a medium with very low impedance, such as a vibrating object in a vacuum. Sound hard, i.e. rigid boundaries, can also be implemented by fixing the displacement to zero.

In many cases, a model may have an axis of symmetry. If this is the case, at the axis, then the displacement perpendicular to it must be zero. This can be understood physically since if there was any displacement in this direction at the boundary then the solution clearly cannot be symmetric. Therefore a model can be built where all the perpendicular components are zero.

If the parallel components are set to zero on the boundary, this produces an antisymmetric boundary condition. This corresponds to the case where the loadings on two sides of the plane are flipped around the perpendicular axis (vs the parallel axis in the case described above). So if the boundary is vertical, for antisymmetry, any loading on the right would have to be flipped around the horizontal axis before being placed on the left.

3.3.3.1 Absorbing boundaries

In many cases, we will be interested in modelling a small section of a large, effectively infinite, domain. One solution is simply to make the domain sufficiently large that no boundary reflections interfere with the area of interest. This means that the domain will be much larger than it actually needs to be. Instead we would prefer to place some form of boundary which absorbs all outgoing waves completely, around the region of interest, and truncate the domain at this location. While this sounds conceptually simple, in practice it is highly challenging, and is still the subject of active research across a number of wave modalities.

In Pogo, the preferred approach is an absorbing region, where the damping gradually increases as the wave propagates into the medium. This gradual change minimises the acoustic impedance mismatch, so helps to minimise any reflections from the boundary [3]. Improvements in performance can be achieved by simultaneously reducing the stiffness as the damping increases in order to reduce the impedance mismatch and hence reduce any reflections; this is known as SRM, the stiffness reduction method [4].

3.3.4 Accuracy

The discretisation approach, whether with FE, FD or any other scheme, needs to capture the behaviour of the physics. This means it is important to use elements that are small enough that their shape functions give a good representation of the underlying function they are trying to approximate. In, for example, static analysis, the elements must be small enough to capture the geometry of any structures, as well as the variations in displacement fields around these structures. This means that more elements are typically used around stress concentrations, where the fields are likely to vary quickly.

In wave problems, the main variation is given by the wavelength, rather than geometrical variations. Therefore the aspect limiting discretisation is the wavelength, and commonly in wave problems, discretisation is expressed as ‘elements per wavelength’. One of the key errors seen in propagation in free space with FE is that the wave velocity is incorrect; the wave speed is often slightly slower than it should be, but this will reduce with improved mesh refinement. For a full analysis of this, readers are referred to the thesis of Drozdz [5], from the Imperial NDE group. There is a trade-off between the level of accuracy in the models and the run time of the simulation; inevitably with the improvement in computational speed over time, the compromise changes. Generally, 20 elements per wavelength is a minimum. It should be noted that elastic problems have two wavelengths because of the shear wave, and depending on the problem, it may be necessary to ensure that both of these are discretised to a particular refinement, which will be more restrictive than just focusing on one wave type.

It is also important what direction the wave is travelling relative to the element. For a square element, the length along one edge is $1/\sqrt{2}$ the length of the diagonal, so different discretisation levels will occur in different directions, and hence the wave speed will be subtly different in each direction. This is effectively an elemental anisotropy, which can have different consequences if a free mesh or a uniform mesh is used. For the uniform mesh, the anisotropy will be the same for every element, so the wave speed will vary with direction. For a free mesh, without any preferential direction, the bulk anisotropy will be averaged out. However, from one element to the next, there will be a small impedance mismatch. This will cause 'mesh scattering' - a small amount of noise which will scatter around behind the initial wave packet. In both cases, these effects will reduce with mesh refinement, but eliminating them is impossible since they are inherently a result of the discretisation approach. Again, readers are referred to the Drozdz thesis [5] which analyses the wave propagation in different directions.

Another critical aspect is that the wave pulse, for reasons described in Sect. 3.2.3, will contain a broad range of different frequencies. These frequencies will have different wavelengths, and therefore experience different numbers of elements per wavelength. This will cause 'numerical dispersion', where different frequencies have different speeds, in extreme cases, causing the frequencies to separate. Again, mesh refinement will help to minimise this. Since the highest frequencies have the shortest wavelengths, they are worst discretised, so in poorly refined models with broadband inputs (hence containing a lot of high frequency components) there is often a noticeable high frequency component after the wave has passed. Therefore it is also important to ensure that the pulse minimises the amount of high frequency energy input into the model.

3.3.4.1 Element shapes

Generally, it is known that distorted elements produce the worst results. Ideally all elements should have the same lengths between the nodes. In some circumstances, if it is known that there will be significantly more detail in one direction than the other, it may be possible to adjust the element shape, but in general this should be avoided. A measure of mesh quality is the minimum angle within an element. If this is small then it can indicate elements exist in the mesh which are quite distorted.

3.3.4.2 Time steps

The time step for explicit methods is dictated by the Courant–Friedrichs–Lewy (CFL) condition. Effectively this means that a wavefront must not travel more than one element in one time step. Generally the time step is therefore calculated as a fraction of the minimum element traversal time; this fraction is known as the CFL number. It is often necessary to be conservative with this; since the waves do not travel at exactly the true wave velocity, then it could be that the time step needs to be reduced. Instability of this form is generally straightforward to identify, with displacement values rapidly becoming very large and heading towards infinity.

3.3.4.3 Rounding and precision

In general, Pogo uses single precision floating point numbers. These store 6-9 decimal significant figures of precision*, along with an exponent from -127 to 128. Rounding errors can result in the result drifting from the true result with each increment. Double precision may help; there is a double precision 2D version of the Pogo solver available.

Note that the GPU rounding behaviour can be slightly different from on a CPU, although these differences are likely to cause negligible difference to the results. One cause of these differences is explained by the CUDA FAQ:

> Why are the results of my GPU computation slightly different from the CPU results?
>
> There are many possible reasons. Floating point computations are not guaranteed to give identical results across any set of processor architectures. The order of operations will often be different when implementing algorithms in a data parallel way on the GPU.
>
> From https://developer.nvidia.com/cuda-faq#Programming

More information can be found on the Nvidia website.

3.3.4.4 Units

It is advised that SI units are used throughout. Note that there is no need to scale the numbers in order to improve precision; this will have no effect because of the exponential terms in the floating point numbers. The use of SI units avoids having to scale the various different parameters and will help to minimise any human errors in the model.

*The variation is caused because the number is stored in binary with 24 bits, which does not map to a constant number of decimal significant figures.

3.3.5 Meshing theory

As outlined above, one of the most critical parameters for achieving the best performance from an FE model is the use of a good mesh. Ideally, this should have elements which are well formed (i.e. sides of similar lengths) and also elements which are reasonably uniform. If meshing a rectangular region, it is straightforward to utilise a uniform grid of square elements in order to meet these criteria. This uniform grid is a structured grid, i.e. the links between the nodes are defined uniformly such that memory access is reasonably straightforward, which is important for some codes. An extension on this is when the uniform grid is distorted, allowing other shapes to be produced, but keeping the relationships between the adjacent nodes the same.

One of the benefits of FE is that it does not need to have a structured element arrangement. Free meshes provide much more flexibility with modelling arbitrary regions. But these need to be meshed suitably. This document will not go into the generation of free meshes in any detail, but do note that there are a huge variety of different meshers out there, which use a number of different principles; any user is advised to research and find the most suitable one for their uses. One of the most common approaches is based on Delaunay triangulation. In this case the domain is triangulated in such a way that no triangle contains the circumcentre of any other triangle. This typically results in high quality meshes, and is the approach that the Pogo mesher (pogoMesh) uses.

Chapter 4

Pogo capabilities

The file format as outlined in the appendix (see Sect. C) contains a comprehensive list of all the features permitted within a Pogo input file. This chapter provides full details about each of the different features which can be exploited within the Pogo solver.

4.1 Geometry, materials and boundary conditions

4.1.1 Mesh

The FE method is built around the meshing of physical domains; in the case of Pogo, this will be a solid medium in which the waves will be simulated. The mesh itself will consist of nodal locations combined with elements to join these together. The nodal locations are defined as x and y values (and z if the mesh is 3D). The elements which join these are defined as a list of nodes for each element, in a specific order to match up with the geometry of the mesh - details of element ordering is included in Sect. B for each element type. For 2D linear elements, nodes are numbered in a clockwise direction. For 3D elements, the base is done first, clockwise from above, then the upper level is done clockwise too (if more than one node).

4.1.2 Materials

A Pogo model will define a set of materials to be used throughout the model. These materials can be isotropic, in which case the parameters defined are E - Young's modulus, ν - Poisson's ratio, ρ - density and, optionally, α - mass proportional damping. Anisotropy can be modelled with other material types, both with orthotropic materials and fully anisotropic; users are referred to Sect. B.2 for details of which parameters should be defined. Orienting anisotropic materials is also possible as discussed in Sect. B.3.

4.1.3 Rigid boundary conditions

It is possible to force the displacement on particular degrees of freedom to zero in Pogo. This can be used to make displacement (i.e. sound hard) boundaries. This also has potential for making symmetric and anti-symmetric boundary conditions by only fixing certain degrees of freedom. Full details on these approaches are in Sect. 3.3. The fixed degrees of freedom are simply listed in the Pogo model. The Pogo solver then loops through and forces these to be zero throughout the time steps.

4.2 Time stepping

The time increment (dt) and number of time steps (nt) are stored within the Pogo input file. Section 3.3.4.2 provides guidelines to what to set these to. The Pogo solver will flag if it finds significant instability occurring in the model, which is commonly an indication that the time increment is too large - see Sect. 3.3.4.2 for details.

4.3 Sources and receivers

4.3.1 dofGroups

Sources and receivers can be modelled in a number of ways in Pogo. If desired, these can be applied to a single degree of freedom. However, typically, as discussed in Sect. 3.2.2, transducers will typically behave as a weighted combination of many degrees of freedom. The latest version of Pogo includes the ability to define a weighted group of degrees of freedom which can be used both as a transmitter and a receiver, in order to provide this capability - essentially these are just a list of degrees of freedom in the model and corresponding weights, and referred to as a ‘dofGroup’.

4.3.2 Sources

A source will consist of a function defined at every time increment. This function is typically a windowed sinusoidal function. When a source is applied, this function is either used to excite individual degrees of freedom, or it can point to a group of degrees of freedom previously defined. Sources in Pogo can either be forces or displacements; it is generally strongly advisable to use a force input unless there is a strong case otherwise. This is because forces are 'transparent' to waves. Consider eq. (3.14)

$$Ku + C\dot{u} + M\ddot{u} = F \tag{4.1}$$

for time steps where F is zero, the wave just propagates as normal. By contrast, setting displacement to a particular value will directly adjust the u values, and hence a reflection will occur for any waves reflected back to the source location. It may be possible to include a force excitation which gives the same effect as a particular displacement input, and hence behave in a transparent way, however, discussions of this approach are considered beyond the scope of this document.

4.3.3 Frames

In many situations it will be necessary to run multiple simulations of the same model, just with different sources. A typical example is a phased array. For an N element array, it will typically be necessary to run the model N times in order to simulate the full set of measured data. Pogo includes a 'frames' feature which streamlines this process, and minimises the overhead of repeatedly preprocessing such a model (such as recalculating the element stiffnesses). Within each frame, a different set of sources can be defined, exactly as for a single-frame model. When run, a separate history and field file will be output for each frame, with '-1', '-2' etc. added to each file name to indicate which frame it corresponds to. Note that frames only vary the sources - all other aspects of the model, such as history locations, material properties etc. will remain the same. The solver can be set to just run a specific subset of frames via the –startframe and –stopframe flags - see Sect. A.4 for details.

4.3.4 History outputs

As discussed in Sect. 7.2, one of the outputs from Pogo is history data. The model must specify where the history is recorded from, as well as how frequently. Typically the model time increment will be a small fraction of the wave period, sampling at every time increment is often unnecessary, and hence the sampling frequency can often be set higher values like 2, 3 or 4 time increments. Location requests can either be directly from specific degrees of freedom in the model, or for a dofGroup.

It is also possible to group history outputs together. For example, if physically recording from two arrays, it would be possible to label one 'array1' and the other 'array2', and these labels will be accessible in the output files.

4.3.5 Application of multiple sources or history requests to a single degree of freedom

Previous versions of Pogo (pre v1.11 for sources, pre v1.3 for history) had a restriction that only a single source term could be applied to a single degree of freedom. This restriction has been removed in the recent versions. Similarly, history requests could only be one per degree of freedom, but this has now been updated. This means, for example, that the dofGroups can overlap each other.

The only circumstance where problems will occur is if a model exists where both displacement and force sources are used, and there is an overlap. Pogo will flag that there is an issue in this case. Adding both sources to a single degree of freedom will result in undefined behaviour. It is considered very rare that any user will wish to include both types of sources in a single model, and rarer still that there will be overlaps in this.

4.3.6 Field outputs

Section 7.2 also mentions that field outputs can be obtained from Pogo, which can be used to check large regions of the model at particular points in time. The time increments are stored in the Pogo model and used to indicate when the output should be generated. It is also possible to specify a subset of nodes at which to store the outputs, if necessary, so that the entire field need not be stored.

Chapter 5

Model generation

5.1 Model description

The Pogo input file format (extension .pogo-inp) is given in Appendix C. The fields in the file largely correspond to the Pogo features discussed in Sect. 4. This is a binary format, which has certain advantages (improved precision, loading speed, smaller file sizes) over text formats, although this makes is difficult to edit without a dedicated tool. For the size of typical models run these days, manually editing a text file is considered impractical anyway.

There are several tools available for generating Pogo input files. The first is the GUI, Pogo.Pro; this is under active development to add more features and is currently best suited for simpler models. The second option is to utilise the set of Matlab functions available to generate models, possibly in conjunction with the Pogo mesher. The last option is to convert an existing Abaqus input file (extension .inp) into a Pogo input file with a converter. These options are explained below.

5.2 Pogo.Pro

The GUI, Pogo.Pro is designed to provide a user-friendly method to generate Pogo input files. For instructions on how to use this, users are advised to read Chapter 2, and follow the example through.

5.3 pogoMesh

The Pogo suite includes a 2D triangular meshing tool, which can take a set of points and adjoining lines and generate a suitable mesh. The mesher is based on Delaunay triangulations, and is based around the papers by Jonathan Shewchuk [6]. While his version of the mesher ('Triangle') uses C code, pogoMesh is written from the ground up in C++ and typically shows a performance advantage over Triangle. PogoMesh also has a number of additional features including the use of uniform meshes and smoothing (using a similar approach to DistMesh [7], more information at http://persson.berkeley.edu/distmesh/).

As an input, the mesher takes a .poly file. This contains three types of data: point coordinates, line segments, and holes. Any points defined in the .poly file are guaranteed to appear in the final mesh (provided they are enclosed in segments), and any mesh will conform to the line segments included. The hole definitions are used to indicate areas which should not be meshed; a set of coordinates for each hole is given, then any triangles in this region, up to a boundary marked by line segments, is removed. An example of this is presented in Fig. 5.1. Figure 5.1(a) shows the mesher input, with a number of points labelled, along with segments joining the points and a hole marked. Figure 5.1(b) shows the resulting mesh with no refinement. This clearly shows how points 11 and 12 are maintained in the final triangulation. Figure 5.1(c) presents the mesh with 2mm mesh refinement - it is now clear that the mesh has been removed from around the hole marked, and Fig. 5.1(d) gives the mesh with a 1mm refinement.

The .poly file format is outlined in Sect. C.5. It is a simple text based format. For the example in Fig. 5.1, the .poly file is:

```
12 2 0 0
1 0.06 0.038
2 0.102 0.05
3 0.102 0.06
4 0.091 0.063
5 0.097 0.087
6 0.077 0.11
7 0.044 0.071
8 0.056 0.063
9 0.066 0.062
10 0.068 0.082
11 0.072 0.052
12 0.087 0.083
10 0
1 1 2
2 2 3
3 3 4
4 4 5
5 5 6
6 6 7
7 7 1
8 8 9
9 9 10
10 10 8
1
```

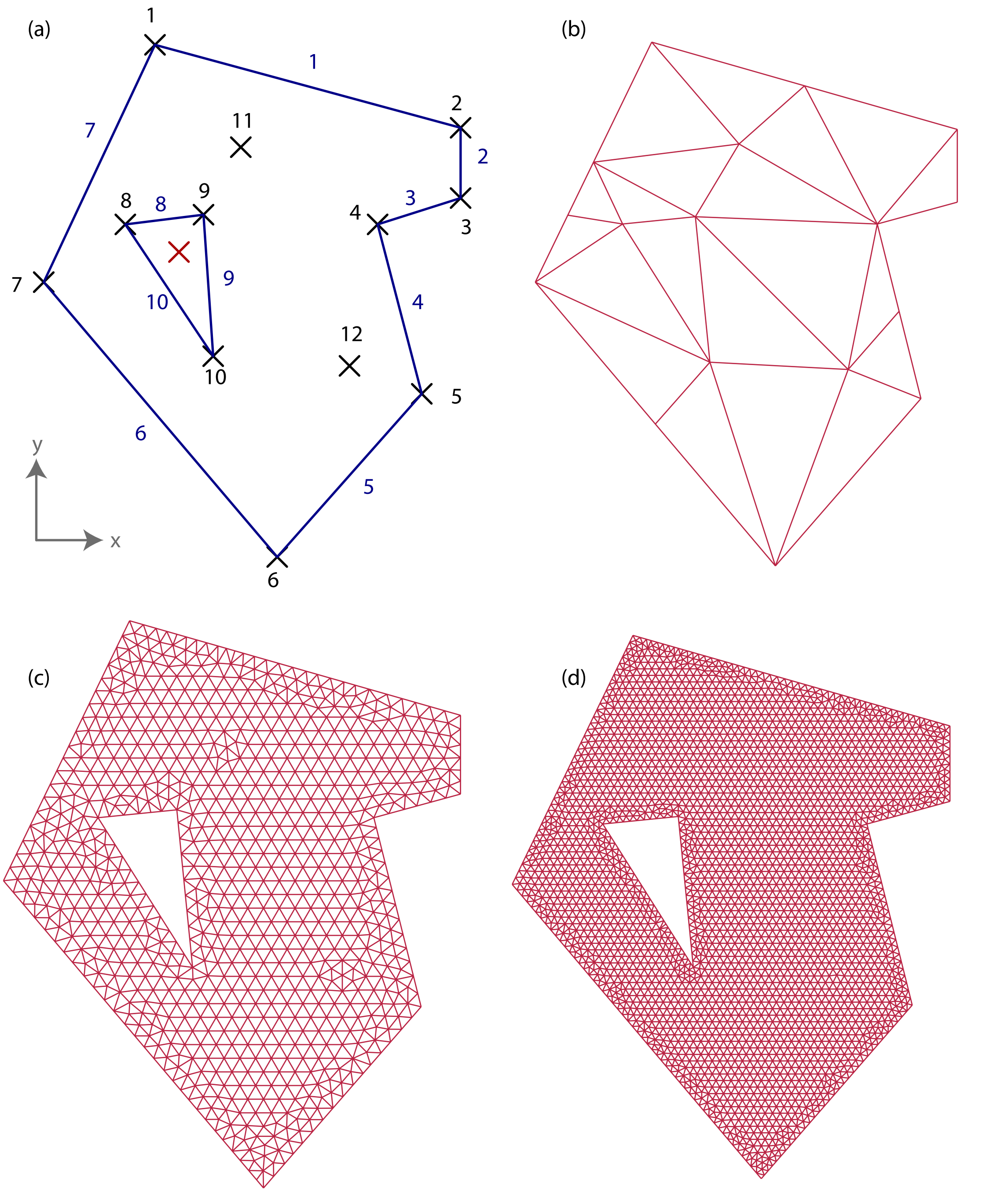


Figure 5.1: (a) Input for pogoMesh. Points are defined with black crosses, and segments with blue lines. The red cross indicates a hole; any triangles in this region will be removed. (b), (c) and (d) show the produced mesh with different levels of refinement. In (b) no refinement is performed. In (c) the desired edge length is 2mm and in (d) it is 1mm.

```
1 0.063 0.067
```

The first line defines the number of points, number of dimensions (always 2) and two unused values (0), and subsequent lines define the x and y coordinates of each point. The following line defines the number of segments and one unused value (0), then the lines after that indicate the start and end points of each segment. Note that it is not necessary for every point to have a segment attached to it, as shown in Fig. 5.1. The last section begins with a line defining the number of holes, then subsequent lines giving the central coordinates of each hole, with the behaviour as described above. The .poly format can be saved via a Matlab function as described in Sect. 5.4.2.2, which takes the points, segments and hole values in Matlab matrices.

pogoMesh is run from the command line. Full details can be found in Sect. A.2. The main flag to use is '-s X' - this sets the desired size of the element to X.

The output from pogoMesh is in the VTU format (VTK unstructured grid). This is a widely used open mesh format, which is compatible with a range of software packages. It can be loaded in to Matlab (see Sect. 5.4.2.2) and hence used as part of the model generation process, as well as imported directly into Pogo.Pro. The VTU format is outlined at https://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf.

5.4 Matlab model generation functions

Matlab functions are provided for loading and saving all the Pogo formats, and similar functions can be made in languages like Python, so one approach is to make a suitable model (i.e. generate nodal locations and elements to join these) in one of these languages, then save directly to the Pogo format. For the use of the Matlab functions, it is assumed that these scripts are on the local machine and accessible on the Matlab path.

5.4.1 Direct model generation

Within Matlab, a struct construct is used to contain all aspects of the model. This can be passed to the saving routine to generate a .pogo-inp file, or alternatively it can be loaded into matlab via the loading routine. Details of this are below.

5.4.1.1 Loading and saving model

The struct can be loaded and saved with the following commands:

```
m = loadPogoInp(fileName);
```

```
savePogoInp(fileName , m, ver);
```

In these functions, *m* defines the model struct. *fileName* defines the file name to be loaded or saved to, and *ver* defines the file version to save to (default 1.09, the latest, with 1.07 and 1.08 also available).

The full details of the fields which need to be placed into the struct *m* are below, taken from the output of 'help savePogoInp'.

- m.nDims - 2 or 3 for 2D or 3D
- m.nDofPerNode - as above for elastodynamic problems
- m.runName - unused at present
- m.dt - time step
- m.nt - number of time increments
- m.nodePos - nodal locations; size nDims x nNodes
- m.elNodes - nodes for each element; size nNodesPerElMax x nEls
- m.elTypeRefs - which of the element types each element refers to, length nEls
- m.matTypeRefs - which of the material types each element refes to, length nEls
- m.orientRefs - which of the orientations each element refes to, length nEls
- m.elTypes{n} - structure for each of the element types, referred to by model.elTypeRefs
- m.elTypes{n}.name - element name
- m.elTypes{n}.paramsType - parameters associated with the element type - usually just 0
- m.matTypes{n} - structure for each of the material types, referred to by model.matTypeRefs
- m.matTypes{n}.parent - what is the parent material (0 if no parent)
  - used in absorbing boundaries; any 'derived' material whose damping or stiffness is varied from the original material should have their parent values set to that material number
- m.matTypes{n}.paramsType - parameters associated with the element type
  - 0 means paramValues are E, nu, rho, alpha (optional), 1 is anisotropic
- m.matTypes{n}.paramValues - the parameters mentioned above
- m.or{n} can specify orientations

- m.or{n}.paramsType - parameter type (0 normally)
- m.or{n}.paramValues - values for the parameters (see Sect. B.2)
- m.fixNodes - nodes with DOFs to be fixed
- m.fixDof - DOF corresponding to the nodes above
  - Nodes can appear in fixNodes multiple times to specify different DOF.
- m.dofGroups{n}.dofSpec - which DOFs
- m.dofGroups{n}.nodeSpec - which nodes
- m.dofGroups{n}.dofWeight - what is the weighting for each
- m.frames{n} - structure for each frame
  - several frames can be used in each model to specify different source setups; can be useful for array imaging
- m.frames{n}.ntSig - number of time points for all the signals (commonly just set to m.nt)
- m.frames{n}.dtSig - time step for the signals (commonly just set to m.dt)
- m.frames{n}.sigs{m} - structure for each of the signals used
- m.frames{n}.sigs{m}.isDofGroup - 0 if not, 1 if dofGroups are referred to in dofSpec
  - if 0:
  - m.frames{n}.sigs{m}.nodeSpec - nodes the signal is applied to.
    * NB if saving to v1.08 or earlier: while nodes can appear multiple times in nodeSpec, they must be applied to different DOF. Only a single signal can be applied to each DOF in the model.
    * From v1.09, these restrictions are lifted.
  - m.frames{n}.sigs{m}.dofSpec
    * DOF to apply the signal to (matches nodeSpec) in (range: 1 to model.nDofPerNode)
  - if 1:
    * m.frames{n}.sigs{m}.dofGroup - vector of numbers corresponding to the dofGroups defined earlier to apply the signal to
- m.frames{n}.sigs{m}.sigType - 0 force, 1 displacement
- m.frames{n}.sigs{m}.sigAmps - amplitudes the signals are multiplied by for each DOF/-groups specified
- m.frames{n}.sigs{m}.sig - signal (length model.ntSig)
- m.measFreq - number of time increments between when history measurements are taken

- m.measStart - starting increment (1 indexed)
- m.measSets{n} - structure for each of the measured node sets
- m.measSets{n}.name - string name for the set
- m.measSets{n}.isDofGroup - do we refer to DOF groups or do we use nodal values (0 or 1)
  - if 0:
  - m.measSets{n}.measNodes - nodes to take history measurements from
  - m.measSets{n}.measDof - degrees of freedom to take history measurements from
  - if 1:
  - m.measSets{n}.dofGroup - vector of numbers corresponding to the dofGroups defined earlier to take the signal from
- m.fieldStoreIncs - which increments to output the field at
- m.fieldStoreNodes - nodes at which to output the field. Omit if all. Empty array if none.

It is wise to follow one of the Matlab examples and view the struct fields this way in order to learn more about what must be included.

5.4.2 Mesh generation

5.4.2.1 Structured grid

Several automatic routines are included to make a structured grid:

```
m = genGrid2D(nx, ny, dx, dy, cx, cy)
m = genTriGrid2D(nx, ny, dx, dy, cx, cy)
m = genGrid3D(nx, ny, nz, dx, dy, dz, cx, cy, cz)
m = genPrismGrid3D(nx, ny, nz, dx, dy, dz, cx, cy, cz)
m = genTetGrid3D(nx, ny, nz, dx, dy, dz, cx, cy, cz)
m = genGridPipe(ri, nr, nc, nz, dr, dz)
```

In each case, a grid of nodes is generated, with x the fastest dimension, followed by y, then z for 3D models. For the pipe, the radial direction is fastest, followed by circumferential, then axial. The variables are given below.

- nx, ny, nz
  - Number of nodes in x, y, z respectively (note that number of elements will be one less than this in each direction)

- dx, dy, dz
  - Spacing between nodes in x, y and z respectively
- cx, cy, cz (optional)
  - Position of the centre of the grid in x, y and z respectively. Defaults to zero in each dimension.
- ri
  - Internal radius of pipe
- nr, nc, nz
  - Number of nodes radially, circumferentially and axially, respectively. Note that the number of elements is 1 less in the radial and axial directions, but the same in the circumferential direction.
- dr, dz
  - Node spacing radially and axially respectively

The default cases use quadrilateral and hexahedral reduced integration elements in 2D and 3D respectively. *genTriGrid2D* generates triangular elements be subdividing the 2D quads, *genPrismGrid3D* subdivides the 3D hexahedral elements into a pair of prisms, and *genTetGrid3D* subdivides the 3D elements into 5 tetrahedral elements. *genTetGrid3D* should only be used for demonstration/testing purposes, since the meshes it generates will be nonconforming because the subdivision will be on a different diagonal on the top and bottom surfaces of the cube.

5.4.2.2 Mesher interface

As discussed in Sect. 5.3, it is possible to generate a 2D mesh using pogoMesh. To generate a suitable .poly file to pass to the mesher, the function

```
savePoly(fileName, points, segments, holes)
```

can be used. It is also possible to write the file directly; the format is not complex. The parameters are:

- points
  - a 2D matrix of the coordinates, with the first dimension indicating x or y, and the second looping through all the points.
- segments

  - indicates the segments linking the points. The first dimension indicates the start or end points for each segment (note points are 1-indexed). The second dimension loops through the segments.

- holes (optional)
  - coordinates indicating where holes should be made in the mesh (same format as points).

For details of how the mesher uses this, see Sect. 5.3. Loading in the output can be done via the loadVtuMesh function (note that this requires access to the xmlreadstring.m function on the path).

```
m = loadVtuMesh(fileName);
```

This can only load triangular meshes, as generated by pogoMesh, and outputs a suitable model struct which can be added to to make a full model.

5.4.2.3 Adjusting the mesh

Some tools are available for doing simple mesh adjustments. To delete elements, there is the function

```
mDel = deleteEls(m, elsDelete);
```

where *m* is the input model, and *mDel* is the output model, where *elsDelete* have been deleted. This will adjust the node numbers, accounting for all the sources and receivers.

It is also possible to disconnect two adjacent elements, for example to make a small crack, with the function

```
mDec = decoupleEls(m, dupeNodes, dir);
```

Here, *m* and *mDec* are the input and output models respectively. *dupeNodes* is a list of nodes which will be duplicated in the model, and *dir* is a direction vector. Elements in this direction relative to the decoupled node will be allocated to the new node.

In some cases, it may be desirable to combine two meshes. This can be done via

```
m3 = combinePogoMesh( m1, m2, n1, n2 );
```

where *m1*, *m2* and *m3*, correspond respectively to the two meshes to be combined and the one resulting output mesh. Note that *m1* is the ‘master’ mesh, so any sources etc. from this mesh are maintained. *n1* and *n2* correspond to the set of nodes in both models which lie directly on top of each other, and will be used to stitch the meshes together.

It may be useful (particularly for deleteEls) to have access to element coordinates. A weighted average can be obtained from the function

```
[ex, ey, ez] = getElCents(m);
```

where *ex, ey* and *ez* correspond to the x, y and z (if 3D) coordinates of each elements.

5.4.3 Other values

As explained in Sect. 5.4.1, there are many other parameters which need to be set as part of the model. There is a helper function (see below) for signal generation, but most others need to be set by the script.

5.4.4 Signal generation

A Hann windowed toneburst (see Sect. 3.2.3) can be added with the following command

```
[ mOut ] = genPogoHannSignal( m, nCyc, freq[, whichSig[, whichFrame]] );
```

where *m* is the input model, *mOut* is the output model. *nCyc* defines the number of cycles and *freq* defines the frequency. *whichSig* and *whichFrame* define which signal number and which frame number respectively; both default to 1 if not defined. Note that the parameters *m.nt* and *m.dt* must both be defined prior to running genPogoHannSignal.

5.4.5 Absorbing boundary generation

As discussed in Sect 3.3.3.1, in many cases, it is important to include absorbing boundaries into the model. A highly automated function has been written to do this:

```
[ mAbs ] = addAbsBound( m, xLims, yLims, zLims, nAbsVals, K, c, f0 );
```

Here, *m* and *mAbs* are the input and output models respectively. *xLims, yLims* and *zLims* define the absorbing boundary extents in x, y and z respectively. Each consists of four numbers: a, b, c and d. Between b and c, the material is unchanged, while from b to a and c to d, the damping is increased according to a cubic power law. Outside the range a to d the damping is at its maximum value. *nAbsVals* defines the number of absorbing materials used; this defaults to 60, which is sufficient for most problems. K is the damping parameter used. If undefined, it will be estimated from the values provided for c (wave speed) and f (frequency). If none of these parameters are defined, c and f are estimated from the model m. Typically the absorbing boundary should be around 3 wavelengths in size.

5.5 Abaqus conversion

If an Abaqus .inp file is available, it is possible to convert it into a Pogo format using the command:

```
pogoFromAb modelName
```

which will take an Abaqus input file 'modelName.inp' and generate a Pogo input file 'modelName.pogo-inp'. However, it should be noted that this does not provide access to the latest Pogo features, and it only supports a subset of Abaqus features. While the converter does its best to flag up any issues while converting there is no guarantee that it will capture everything, so please use caution. It is far better to use a dedicated Pogo input file generation tool.

Chapter 6

Blocking

In order to run the model on a GPU, it must be separated into separate blocks. This enables memory access to be easily managed and optimised, which is critical for quick solving of the problem on GPUs. This problem is specific to discretisations which allow free meshes, which is key to the flexibility of the FE method. More information about the FE discretisation approach can be found in Sect. 3.3.

For general meshes, it is necessary to have an automated method for subdividing the nodes into suitable blocks which can be run on the GPU. Once the blocks have been defined, the memory then needs to be arranged such that each block can access memory from the adjacent blocks.

There are two different algorithms which are available for the subdivision stage. Full details of these can be found in [1]. The first one, developed specifically for Pogo, is the 'aligned' blocker. This only works in 2D. The boundary of the domain is split into lengths of around 32 nodes; each of these becomes the base for a different block. All of the blocks are then advanced simultaneously. Techniques are employed to detect convex and concave edges, and to split and merge blocks accordingly as they advance, so that they all remain well aligned. Because they are well aligned, it is easy to arrange the memory to enable loading from adjacent blocks.

The alternative subdivision routine is the 'greedy' partitioner. This simply starts with the first block, and takes as many nodes as possible until it is 'full', i.e. at the maximum block size. Then it takes the next block, repeating the process until all the nodes are taken.

Pogo also includes an automatic routine to detect structured grids and partition using a structured pattern. This can speed up the blocking process. Since this will automatically be detected by Pogo, the user does not need to make any changes, although the process can be prevented if desired with the command line flag '–ignoreStruct' (see Sect. A.3).

The second stage of the blocking routine is to arrange the memory. Pogo allows a certain number of different sized loads from adjacent blocks. If it is not possible to load from all necessary adjacent blocks within the loads provided, then it will adjust the block structure by splitting the block in question and repeating the memory arrangement approach.

The block information is stored in a .pogo-block file, which is subsequently read by the solver. The solver performs a full range of checks to ensure that the block file is suitable for the model provided, and will quickly give an error message if an unsuitable block file is used.

It should be noted that the block file depends purely on the topology of the mesh, i.e. how the nodes are linked to each other by the elements. This means that the mesh can be distorted (for example by offsetting certain nodes) and the same block file will be sufficient. In any case, since the solver undertakes a rigorous check of the provided block file, if unsure if the topology has changed, it is safe to test with an existing block file.

The programs can be run with:

```
pogoBlock modelName
pogoBlockGreedy modelName
pogoBlockGreedy3d modelName
```

where modelName.pogo-inp is the name of the Pogo input file. These will each produce a block file called modelName.pogo-block. The first is the standard 2D aligned blocker. The second is the 2D greedy blocker and the last one is the greedy blocker in 3D.

Chapter 7

Solver

7.1 Running on GPUs

Having generated a .pogo-inp file and a .pogo-block file, the problem must be passed to the solver. This can be achieved through one of the basic commands:

```
pogoSolve modelName
pogoSolve64 modelName
pogoSolve3d modelName
```

The first two of these run 2D models, the first in single precision and the second in double precision. The third must be used for 3D models. Various flags can be used as described in Sect. A.4.

7.2 Output files

In general, two output files are generated by pogoSolve, modelName.pogo-hist and modelName.pogo-field. These contain different types of data. The history data is generally sparse in space, but dense in time, while field data is the other way around: sparse in time but dense in space. History data is intended to store time traces from specific nodes (or a collection of nodes) in a similar way to a transducer outputting a single time trace. Field data is data output across the whole set of nodes (or a significant fraction of it) at particular time steps. This can be used for animating the field and checking wave behaviour. Physically this could correspond to a laser dopler vibrometer scanning across the surface of the object and producing images of the wavefield across the surface.

The file format for both of these is given in Appendix C. Functions are available in Matlab to load in both file formats. Generally, a tool like Matlab is very useful for post-processing the history signals into whatever form is needed. Viewing field data is less straightforward; this is best done via the GUI combined with Paraview - more information can be found in Sect. 2.

7.3 Multiple GPUs

A notable drawback of running simulations on a GPU is the limited memory available. The maximum amount of memory on a single card is typically far below the amount of RAM which can be installed on a motherboard. This is overly restrictive for many applications, particularly in 3D, where there are more connections between the nodes (greater linking is possible in 3D compared to 2D), and also there are 3×3 stiffness coefficients between a pair vs 2×2 in 2D, due to the degrees of freedom. To address this, a multiple-GPU version of Pogo has been developed, which enables larger problems to be subdivided and run across several cards. The total memory available is now significantly greater than before.

7.3.1 Overheads

Inevitably with parallel computing, there are overheads associated with having to communicate between different GPUs. However, a relatively small amount of data needs to be communicated compared to the total amount stored. The data to be communicated consists of the displacements along the boundaries of the domains between the different GPUs, while the majority of the total stored data is taken up by the K matrix stiffness coefficients. Taking a practical example, we consider an 8 million node model, arranged into a cube with cubic elements between a mesh of $200 \times 200 \times 200$ nodes, and we wish to subdivide this into two 4 million node sections, along a central plane of the model. The two lots of 200×200 boundary displacements need to be communicated, making $80,000$ total floating point transfers. By comparison, between any pair of nodes linked by elements there will be 3×3 coefficients between the degrees of freedom, and for this model there will be 27 pairs (including the link to the node itself), making nearly 2 billion floating point values which need to be stored for K, making a ratio of storage to transfer of nearly 25000.

While few examples in reality are likely to be so neat, it is clear that the amount of data to transfer must be very low for all but the most extreme cases, and the overhead associated with it should be low. It should be recognised, however, that memory transfers between GPUs suffer from low transfer speeds, so Pogo aims to minimise the effect of this. The approach used by Pogo is to initially calculate the values for the blocks at the boundary of the domain. The data from these can then be transferred while the remainder of the simulation is continuing on the other blocks. Typically the transfers are fairly transparent to the user, and efficiency is generally quite high. It is left to the users to test this for their own systems, since it is likely to be very system dependent. Note that various command-line flags exist to specify how many or which GPUs to use for a simulation.

7.3.2 Usage

Pogo aims to use multiple GPUs in as much of a transparent way as possible. If the user has not specified which cards to use, then it will scan through the GPUs and select as many cards as it needs to run the job. It will try to use the fastest cards available with the biggest memory first. In homogeneous systems (i.e. with identical GPUs), with many jobs running simultaneously, it is advisable to use the –noSmart flag to avoid 'race conditions' where one process will select the GPUs to run on, then another process will select some (or all) of the same GPUs before the first process has the opportunity to allocate onto them. The –noSmart flag loops through and allocates immediately onto the first set of GPUs needed in order to solve the problem, which addresses this issue. Note that versions of Pogo prior to v1.1 needed to use the separate pogoDiv commands - documentation for these can be provided if requested.

7.4 Compression

In many scenarios, large sections of uniform elements may be used in a model. This forms some redundancy in the K, C and M matrices - many of the coefficients will be the same. These coefficients could be stored once, then pointed to, so they can be accessed when needed. Pogo has the ability to detect these uniform elements and exploit to make the problem smaller, and hence take up less memory. This enables larger models to be run on the same hardware, and also can increase the speed. Compression is normally turned on by default. Flags to turn it on or off are –compresson and –compressoff - details can be seen in Sect. A.4. Pogo will automatically detect whether compression can be used, however, this can be slow for large models, so it may be desirable to turn it off.

7.4.1 Consequences of using compression

While compression gives an important improvement in capability, there are some potential issues that the user should be aware of. The most significant is the issue of rounding errors; details about the cause of these errors is given in Sect. 3.3.4.3. Essentially these errors mean that even though coefficients should in theory be identical, in practice they are not. This is not a major issue in most FE implementations; the differences are small, and with (say) 20 elements per wavelength, the variations are effectively averaged out as the wave propagates.

The issue is that in copying the coefficients, there is no longer any averaging. If there were two regions which had different materials (as given to Pogo) with the same material properties, each of these would copy a different set of coefficients. A wave travelling from one to the other will show some scattering from the small impedance mismatch at the boundary, caused by using different coefficients. However, it is unlikely that many users will ever need to use multiple identical materials in this way; in practice it is likely that any material property difference will be much larger than the rounding errors.

Chapter 8

Post-processing

Pogo produces two file outputs: a field file (.pogo-field) and a history file (.pogo-hist). The field file contains data needed to visualise the field, while the history file contains time trace data from the model. Both file formats are described in Appendix C.

8.1 Field data

Visualisations can provide valuable data to people modelling ultrasonic problems. It is strongly advised to view the field in order to check that the wave propagation makes physical sense in relation to the intended input, and hence identify many errors which can occur at the model generation stage. A visual output of the wavefield can also help to identify the cause of a particular measured signal; for example one may see an unexpected wavepacket within the signal, and visualising the wavefield shows a surface wave arriving at the same time that the wavepacket appears in the signal.

The preferred solution is to visualise the wavefield via the Pogo.Pro GUI and Paraview, following the instructions of Sect. 2. In this, Pogo.Pro will convert the .pogo-field file into an xdmf file which can be loaded and visualised in Paraview. It is also possible to use Paraview to generate animations and images for papers, presentations etc.

8.1.1 Loading and visualisation with Matlab

Matlab can also be used to visualise the field data. Firstly, the field data must be loaded via

```
f = loadPogoField(fileName);
```

As with the model, the field data is contained within a single struct, f. This contains a number of fields:

- nodeLocs - positions of nodes, dimension × nodeNum
- times - the time values when the field values are recorded
- ux, uy, uz - displacements in each degree of freedom (uz only for 3D) for each time step (nodeNum × timeStep)
- nodeNums - the node numbers corresponding to the output

For structured meshes, this is straightforward to plot; the grid of nodes can be directly mapped to a grid of pixels to produce an image. For free meshes, the output displacements need to somehow be interpolated down to the pixel grid. This can be done in conjunction with the model file to provide element information about how the nodes are linked, or Matlab has built-in functions for remapping from arbitrary positioned points onto a grid. The function

```
viewPogoField( f, npix, tImages, plotDir, plane, offset, wm )
```

uses the latter. The inputs for this are:

- f - the field struct, as loaded via loadPogoField()
- npix, the number of pixels along the longest axis (default 100)
- tImages - which images to plot, a vector containing the numbers (default all)
- plotDir - which direction of displacement to plot; x = 1, y = 2, z = 3, 0 = magnitude (default)
- plane - (3D only) which plane of the model to plot; x-y = 1, x-z = 2, y-z = 3, auto = 0 (default). Auto plots the two dimensions with the largest range.
- offset - (3D only) position along the other axis (default = 0)
- wm - wave map to use - passed directly to colormap so see ‘help colormap’ for details. Defaults to wave ‘red-white-blue’ map.

Since many of the values have default values, it is possible to generate an animation via

```
f = loadPogoField('example.pogo-field');
viewPogoField(f);
```

It should be noted that the approach is not particularly fast when compared to dedicated tools like Paraview.

8.2 History data

Utilising the history data is very dependent on the application. For example, if the model is to be used to test an imaging algorithm, the data will need to be post-processed and passed to the algorithm, in whatever language this is written. The open file format provides a mechanism to achieve this. Section C.3 provides this format.

Pogo.Pro can again be used to visualise the history output. Details on how to do this are given in Sect. 2; again the data is converted to a format compatible with Paraview, which is then used to visualise the data.

8.2.1 Matlab processing

The history data can be loaded into Matlab via the command

```
h = loadPogoHist( fileName );
```

where, as before, h is a struct containing the history data and accompanying information. This struct contains the following fields

- nt - the number of measurement times
- dt - the time spacing between measurement times (s)
- startMeas - the time for the first time point (s)
- sets - a struct inside h, containing each set name (default is 'main' if unspecified), which is itself a struct, containing:
  - name - the defined name matching that specified in the input file
    - Note that the field name in the struct has to have non-word characters removed to conform with Matlab naming conventions; this string has the true version
  - nodeNums - the numbers of the nodes at which measurements are given
  - nodeDofs - the degree of freedom for each measurement
  - nodePos - location of each node; dimension × node number
  - histTraces - the measurements. Time × node number

This could be visualised in Matlab with the following example code

```
h = loadPogoHist('example.pogo-hist');

%make a vector containing the time values:
```

```
t = h.startMeas+(0:h.nt-1)*h.dt;

%now plot all the time traces in 'main'
figure
plot(t, h.sets.main.histTraces)
```

Chapter 9

Examples

9.1 Matlab example

Below is an example Matlab script used to generate a model. This aims to use as many of the provided functions as possible.

```
clear

%define element size
dx = 0.2e-3;
%lambda = 4mm so this is 20 els per wavelength at 1 MHz

%define nodes along the bottom of the free meshed domain
nodex = (0.1:dx:0.2).';

%put into a points array, along with another point at the top to
  make a triangle
points = [ [ nodex zeros(length(nodex),1)]
    0.15 0.06].';

%add points inside for a rectangle to be deleted
rectPoints = [ 0.14 0.01
    0.16 0.01
    0.16 0.025
    0.14 0.025].';
%combine:
points = [points rectPoints];
```

```

nPointsOuter = length(nodex)+1;

%define the segments to joint the points around the outside
   together
segsOut = [ (1:nPointsOuter).' [2:nPointsOuter 1].'].';
%this matrix will be:
%1,2
%2,3
%3,4
% ...
%n-1, n
%n, 1

%segments for the internal rectangle
segsRect = [(nPointsOuter+1:nPointsOuter+4).' [(nPointsOuter+2:
   nPointsOuter+4) nPointsOuter+1].'].';
%combined segments
segs = [segsOut segsRect];

%one hole which will be deleted
holes = [0.148 0.015].';

%save into a poly format
savePoly( 'temp.poly', points, segs, holes );

%IMPORTANT
%This might need to be changed depending on your system setup
%Essentially you want to run pogoMesh with the file just
   generated
%and maximum length set to dx*1.5
system(sprintf('pogoMesh temp.poly -l %f',dx*1.5))

%having done the meshing can delete the file
delete('temp.poly')

%load in mesh
m = loadVtuMesh('temp.vtu');
%can delete mesh file now loaded
delete('temp.vtu')

%generate a 2D grid
%centre locations for grid
% - note that we want to make the top of this line up with the
   nodes from
```

```
% the free mesh
cx = 0.15;
cy = -0.05;
%points in x and y
nx = 0.1/dx+1;
ny = 0.1/dx+1;
%actually make grid
m2 = genGrid2D(nx,ny,dx,dx,cx,cy);

%get the overlapping node numbers from each model; the top line
  from 2, the
%bottom line from 1
n2 = (1:nx) + (ny-1)*nx;
n1 = 1:nx;

if 0
    %can plot the two models if desired
    figure
    plot(m.nodePos(1,:),m.nodePos(2,:),'k.')
    hold on
    plot(m2.nodePos(1,:),m2.nodePos(2,:),'bx')
    plot(m.nodePos(1,n1),m.nodePos(2,n1),'go')
    plot(m2.nodePos(1,n2),m2.nodePos(2,n2),'r+')
    axis equal
end

m3 = combinePogoMesh(m,m2,n1,n2);

%get number of nodes in model 1, the free meshed model
nNodes1 = size(m.nodePos,2);

%source and receiver points - offset by nNodes1 from the start,
  then simple
%arithmetic in the grid to set two locations
sPoint = nNodes1+(nx+1)/2+(ny+1)/2*nx;
rPoint = nNodes1+(nx+1)/2+(ny-5)*nx;

%define material properties
E = 210e9;
G = 80e9;
nu = E/2/G-1;
rho = 8000;

courant = 0.3; % mix of sizes so be conservative with stability

```

```
%define frequency
freq = 1500e3;
%number of cycles in toneburst
nCyc = 3;
%sound speeds
c0 = sqrt(E*(1-nu)/(rho*(1+nu)*(1-2*nu)));
cSh = sqrt(E/(2*rho*(1+nu)));

%get wavelengths
lambda = c0/freq;
lambdaSh = cSh/freq;

%set dt and nt
m3.dt = dx/c0*courant;
%want 30e-6 s long simulation
m3.nt = round(30e-6/m3.dt);

%get number of elements
nEls = size(m3.elNodes,2);
%define the material references (set them all to material 1)
m3.matTypeRefs = ones(nEls,1);
%then define material 1 with the properties from before
m3.matTypes{1}.paramsType = 0;
m3.matTypes{1}.paramValues = [E, nu, rho];

%put a Hann signal on sig 1, frame 1
m3 = genPogoHannSignal(m3,nCyc,freq,1,1);

%say what nodes to apply it to
m3.frames{1}.sigs{1}.nodeSpec = [sPoint sPoint];
%and which degrees of freedom
m3.frames{1}.sigs{1}.dofSpec = [1 2];

%set the signal amplitudes in each direction
m3.frames{1}.sigs{1}.sigAmps = [-0.3 0.6];
%set the type to 0, for force
m3.frames{1}.sigs{1}.sigType = 0;

%define the measurement frequency and start time
m3.measFreq = 2;
m3.measStart = 1;

%define the measurement sets
m3.measSets{1}.name = 'main';
m3.measSets{1}.measNodes = [rPoint,rPoint];
```

```
m3.measSets{1}.measDof = [1 2];
%receive in x and y at node rPoint

%calculate time increments to store the field at
gap = round(m3.nt/80); % do at around 80 time steps
if gap < 1
    gap = 1;
end
m3.fieldStoreIncs = 1:gap:m3.nt;

%delete some elements
[ex,ey] = getElCents(m3); %get element centroids
%find which elements to delete based on coordinates
elsDel = find(ex < 0.15 & ex > 0.14 & ey > -0.09 & ey < -0.08);
%delete them
m3 = deleteEls(m3,elsDel);

%put an absorbing boundary on the bottom edge
xLims = [];
yLims = [-0.1 -0.1+0.012  100 102];
%^only use first two values here - the last two are set to very
   large so
%that they are outside the model. Here we're using 3 wavelength
   wide
%boundaries.

m3 = addAbsBound(m3,xLims,yLims,[],[],[],c0,freq);

%duplicate some nodes to decouple some elements - model a very
   thin crack
px = m3.nodePos(1,:);
py = m3.nodePos(2,:);
nodesDupe = find(px < 0.1801 & px > 0.1799 & py > -0.05 & py <
   -0.04);
n = nodesDupe;
m3 = decoupleEls(m3,nodesDupe,[1 0]);
%direction set to x direction

%save the file - this is done in the slightly older v1.07 format
savePogoInp('example.pogo-inp',m3,1.07);
```

This example defines a triangular free meshed section, with a rectangular region subtracted from the middle of it. Subsequently, a grid is attached to the bottom. This example is included with the package of different Matlab functions.

Appendix A

Pogo commands reference

This section provides an outline of the command line options available for the four different command line components for Pogo – *pogoFromAb*, *pogoMesh*, *pogoBlock* and *pogoSolve*. Here | is used to indicate 'or', and square brackets [] indicate optional arguments. In all cases, the Linux names of the commands are given; for Windows *.exe* will need to be added.

A.1 pogoFromAb

This program converts an Abaqus *.inp* file into a Pogo *.pogo-inp* file, enabling Abaqus models to be run using Pogo.

```
pogoFromAb model [-h|--help|--version] [-vN|--verbosity N] [--ts] ...
    [--noparse] [-p pogoModelName |--pogooutput pogoModelName]
```

The options are outlined below

- *model* identifies the file *model.inp* in the current working directory which contains the Abaqus input file. If no period (.) is present in the name given, it is assumed that the extension *.inp* needs to be added to the file. By default, the output Pogo model name will be the same as the input name without the *.inp* present.
- *-h, –help*: Output a brief help and exit. Note that the model name need not be included to request help.
- *–version*: Output the version and exit.

- *-vN, –verbosity N*: Set the verbosity to N. By default this is set to 2, which gives a reasonable level of output.
- *–noparse*: Ignore any usage and processing of the **PARAMETER* keyword from the Abaqus keyword for speed.
- *-p pogoModelName, –pogooutput pogoModelName*: set the output file to *pogoModelName*, so the output file name will be *pogoModelName.pogo-inp*.
- *–ts*: output time stamps to the terminal at appropriate locations for logging purposes.

A.2 pogoMesh

This is a 2D mesher which takes as its input a set of points and line segments and generates a triangular mesh from them. See Sect. 5.3 for details.

```
pogoMesh file [-h|--help|--version] [-vN|--verbosity N] [--ts]...
    [-s S] [-a A] [-l L] [-d D] [--noStruct] [--noSmooth] [--nSmoothIts]
```

The options are outlined below

- *file* identifies the file *file.poly* in the current working directory which contains a suitable poly definition file. If no period (.) is present in the name given, it is assumed that the extension *.poly* needs to be added to the file.
- *-h, –help*: Output a brief help and exit.
- *–version*: Output the version and exit.
- *-vN, –verbosity N*: Set the log level (verbosity) to N. By default this is set to 2, which gives a reasonable level of output for most applications.
- *–ts*: output time stamps to the terminal at appropriate locations for logging purposes.
- *-s S*: set the desired mesh size to Sm. It is advised not to use this in combination with any of the following geometry parameters.
- *-a A*: set the maximum area of any element to Am^2.
- *-l* L: set the maximum side length of any element to Lm.
- *-d* D: set the minimum angle within any element to D degrees.
- –noStruct: do not try to use uniform structured meshes where possible.
- –noSmooth: turn off smoothing applied at the end (can greatly speed up meshing, but less neat results).

- –nSmoothIts: number of iterations of smoothing (default 10).

The output will be in the VTU format, in *file.vtu*.

A.3 pogoBlock, pogoBlockGreedy, pogoBlockGreedy3d

This program takes a *.pogo-inp* file and partitions it according to the methods in [1] to generate a *.pogo-block* file, needed to run the simulation.

```
pogoBlock|pogoBlockGreedy|pogoBlockGreedy3d model ...
    [-h|--help|--version] [-vN|--verbosity N] [--ts] ...
    [--ignoreStruct]
```

pogoBlock is used for partitioning 2D models using the aligned partitioner. *pogoBlockGreedy* partitions 2D models with the greedy partitioner, and *pogoBlockGreedy3d* partitions 3D models with the greedy partitioner.

The options are outlined below

- *model* identifies the file *model.pogo-inp* in the current working directory which contains the Pogo input file. If no period (.) is present in the name given, it is assumed that the extension *.pogo-inp* needs to be added to the file.
- *-h, –help*: Output a brief help and exit. Note that the model name need not be included to request help.
- *–version*: Output the version and exit.
- *-vN, –verbosity N*: Set the verbosity to N. By default this is set to 2, which gives a reasonable level of output.
- *–ts*: output time stamps to the terminal at appropriate locations for logging purposes.
- *–ignoreStruct*: do not make any checks to see whether the mesh is structured. This is normally done; if a structured mesh is detected then a structured block arrangement can be used, which is generally faster to generate and more efficient.

A.4 pogoSolve, pogoSolve3d, pogoSolve64

This program runs the simulation on the GPU, using Pogo.

```
pogoSolve|pogoSolve3d|pogoSolve64 model ...
    [-b|--blockfile blockfilename] [--outfile outfile] ...
    [--nosavefield] [--startFrame P] [--stopFrame Q] ...
    [-o] [--noSmart][--ngpus N|--device X,Y,Z] ...
    [-vN|--verbosity N] [--ts] [-h|--help|--version|--test]
```

pogoSolve is used for solving 2D models in single precision, and *pogoSolve3d* solves 3D models. *pogoSolve64* is used for solving 2D models in double precision.

The options are outlined below

- *model* identifies the files *model.pogo-inp* and *model.pogo-block* in the current working directory which contains the Pogo input file and block file respectively. No extension should be included with this.
- *–blockfile, -b*: specify a different block file from the default one stored in *model.pogo-block*.
- *–outfile*: Define the main section of the output file names. *.pogo-hist* and *.pogo-field* will be appended to this to generate file names for the history data and the field data respectively.
- *–nosavefield*: Specify not to save the field data. In many cases the most important data is the history data, but saving the field for many times takes a lot of resources and can significantly slow the simulation down. This option allows the field saving to be skipped from the command line.
- *–startFrame P* and *–stopFrame Q*: for models containing multiple frames, start at the specified frame P and finish upon reaching Q. This in inclusive.
- *-o*: indicate that Pogo should automatically overwrite any existing output files without prompting - useful for batch processes.
- *–noSmart*: prevents 'smart' allocation of GPUs. This can avoid race conditions where GPUs can be allocated to multiple jobs if started around the same time (see Sect. 7.3).
- *–ngpus N*: will force N GPUs to be used for the model, if they exist on the system and they are large enough for the model.
- *–device* X,Y,Z: give a list of GPUs which should be used for the simulation. No spaces should be included in the list of numbers, and the numbers should just be separated by commas. The numbers are as given in the output from the 'nvidia-smi' command.
- *–compresson, –compressoff*: enable and disable compression. See Sect. 7.4 for details of this approach.
- *-h, –help*: Output a brief help and exit. Note that the model name need not be included to request help.
- *–version*: Output the version and exit.

- *-vN, –verbosity N*: Set the verbosity to N. By default this is set to 2, which gives a reasonable level of output.
- *–ts*: output time stamps to the terminal at appropriate locations for logging purposes.
- *–test*: this will perform a simple test of the GPU and licensing hardware to check that the installation is working.

Appendix B

Element and materials library

This document contains information about the elements available in Pogo. Each element is assigned a material and, optionally, an orientation. The elements, orientations, materials have different parameters depending on whether they are isotropic, orthotropic etc; details of the parameters necessary are given in this document.

To define parameters in the input file, as given by the file format in Appendix C, a number specifying the parameter type is stored, for example indicating whether the parameters indicate Young's modulus etc. or whether they contain constitutive matrix terms for an anisotropic medium. The numbers are given in this document.

B.1 Elements

Node numbering is as defined in Fig. B.1.

B.1.1 CPE3/CPS3

Plane strain (E) and plane stress (S), 3-noded linear triangular elements. These match the Abaqus elements of the same name.

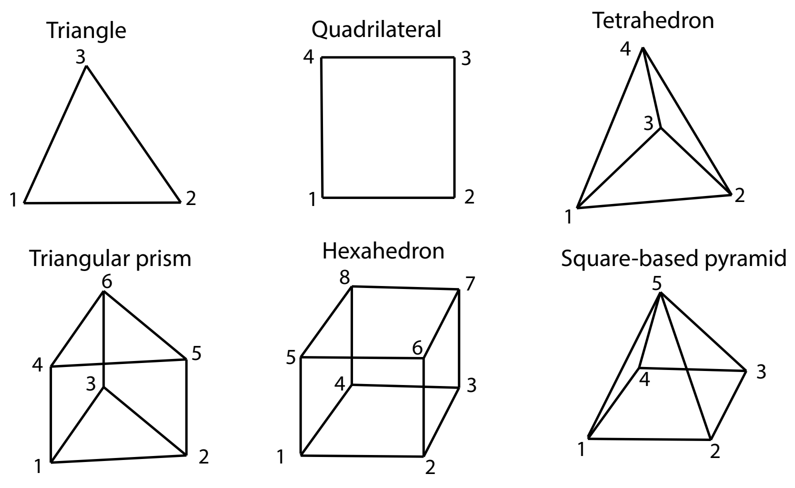


Figure B.1: Node numbering for all elements. Note that no pyramidal elements are currently supported in Pogo at present.

B.1.2 CPE4R/CPS4R

Plane strain and plane stress, 4-noded bilinear quadrilateral elements. Use reduced integration with hourglass control. These match the Abaqus elements of the same name. Hourglass control parameter is set to 0.05 by default - can be changed by setting element parameter 1 under element parameter type 1.

B.1.3 CPE4/CPS4

Plane strain and plane stress, 4-noded bilinear quadrilateral elements. Fully integrated, so slower to formulate than CPE4R/CPS4R, and can also suffer from shear locking. Not supported in Abaqus Explicit, but can be included in Abaqus input file and will be translated to Pogo input file.

B.1.4 C3D4

Tetrahedral element. Matches equivalent Abaqus element.

B.1.5 C3D8R

8 noded brick element; reduced integration with hourglass control. Hourglass control parameter is set to 0.05 by default - can be changed by setting element parameter 1 under parameter type 1. Matches equivalent Abaqus element.

B.1.6 C3D8

8 noded brick element, fully integrated. Matches equivalent Abaqus element.

B.1.7 C3D6

6 noded triangular prism element, fully integrated.

B.1.8 C3D6R

6 noded triangular prism element, reduced integration. Matches Abaqus equivalent.

B.2 Materials

B.2.1 Type 0 - Isotropic

Parameters: E, ν, ρ, α (damping); damping optional.

B.2.2 Type 1 - Orthotropic

Parameters: D_{1111}, D_{1122}, D_{2222}, D_{1133}, D_{2233}, D_{3333}, D_{1212}, D_{1313}, D_{2323}, ρ, α (damping); damping optional.

B.2.3 Type 2 - Anisotropic

Parameters: D_{1111}, D_{1122}, D_{2222}, D_{1133}, D_{2233}, D_{3333}, D_{1112}, D_{2212},

D_{3312},D_{1212},D_{1113},D_{2213},D_{3313},D_{1213},D_{1313},D_{1123},

D_{2223},D_{3323},D_{1223},D_{1323},D_{2323},

ρ, α (damping); damping optional.

B.3 Orientations

B.3.1 Type 0

Parameters: a_x, a_y, a_z, b_x, b_y, b_z, rotation axis, rotation angle(degrees). Note that the only supported configuration is for 2D, so $a_x = 1$, $a_y = 0$, $a_z = 0$, $b_x = 0$, $b_y = 1$, $b_z = 0$, with the rotation axis being set to 3 (out of plane).

Appendix C

File formats

File formats - input, hist, field and poly. Note that all file formats should be backwards compatible, i.e. the most recent version of Pogo should be able to load in any version of the file.

C.1 General notes

The order of the dimensions is given by the order of the parameters in the bytes column. For example, prec*nDims*nNodes means the prec bytes making up each value is the fastest variable, followed by the dimensions, followed by the number of nodes. Everything is 0 indexed unless specified otherwise.

C.2 Problem input file

Extension: .pogo-inp. File details for v1.10 with indications about the older formats. Some parts of the input file are repeated, for example to define several sources. The [bracket in the first column is used to indicate which parts are being repeated and how much nesting of this is happening.

For DOF encoding, the two least significant bits encode the local DOF (e.g. x, y, z) and the remaining higher bits correspond to the node number (i.e. effectively multiplied by 4). From v1.10 (64 bit storage of DOFs) this means 62 bits for the node, in v1.09 and before this meant 30 bits. This is the case for all situations referring to DOF below unless explicitly stated.

	Variable Name	Bytes	Type	Description
	header	20	char	File header, must match '%pogo-inp1.10'
	prec	4	int	Precision, in bytes (4 or 8), of floating point data
	nDims	4	int	Number of dimensions for node locations
	nDofPerNode	4	int	Number of DOF per node (should match nDims at present)
	notes	1024	char	Optional text data about the file
	runName	80	char	Null terminated string specifying the name for the model[*]
	nt	4	int	Number of time steps
	dt	prec	float	Length of time step (s)
	nNodes	4	uint	Number of nodes
	pos	prec*nDims*nNodes	float	Nodal positions (m)
	nEls	4	uint	Number of elements
	nNodesPerEl	4	int	Maximum number of nodes per element
	elTypeRefs	4*nEls	int	Element reference
	matTypeRefs	4*nEls	int	Material reference
	orientRefs	4*nEls	int	Orientation reference[†]
	elNodes	4*nNodesPerEl*nEls	uint	List of nodes making up each element[‡]
	nPmlSets	4	int	Unsupported. Set this to zero.
	nPmlParams	4	int	Set to zero.
	nElTypes	4	int	Number of element types
		For each of the nElTypes element types:		
[	elType	20	char	Null terminated string indicating the element type
[	elParamsType	4	int	Defines what form the parameters take. Usage depends on element.

[*] Unused at present.
[†] Unused values set to -1
[‡] Unused values set to $32^2 - 1$

[	nElParams	4	int	Number of element params
[	elParamValues	prec*nElParams	float	Define the parameter values according to the element definition (see element library)
	nMatTypes	4	int	Number of material types
		For each of the nMatTypes material types:		
[	matParentType	4	int	Which is the parent material[*] (used if absorbing boundaries present). If no parent, set to -1.
[	matParamsType	4	int	Defines what form the parameters take[†]
[	nMatParams	4	int	Number of material params
[	matParamValues	prec*nMatParams	float	Define the parameter values
	nOrientations	4	int	Number of orientations
		For each of the nOrientations:		
[	orParamsType	4	int	Defines what form the parameters take
[	nOrParams	4	int	Number of orientation params
[	orParamValues	prec*nOrParams	float	Define the parameter values
	nFixDof	4	uint64[‡]	Number of degrees of freedom fixed
	fixDof	4*nFixDof	uint64[§]	Which DoF are fixed[¶]
	nTieSets	4	int	Number of sets of ties[‖]
		Then have nTieSets sets:		

[*]From v1.09 onwards.
[†]Details of parameters are given in Appendix B.
[‡]From v1.10 onwards. Before, was int32.
[§]From v1.10 onwards. Before, was int32.
[¶]See encoding information above
[‖]All tie params are only included from v1.06 onwards

[	tieTransform	nDofPerNode*nDofPerNode*prec	float	Matrix describing how the coordinates are transformed between each node pair
[	nTies	4	int	Number of node ties in this set
[	masterNodes	4*nTies	int	Master nodes
[	slaveNodes	4*nTies	int	Slave nodes
	nDoFGroups	4	int	Number of DoFGroups (link together degrees of freedom and weight them for sources and receivers)[*]
		Then have nDoFGroups sets of:		
[	nDoF	4	int	Number of degrees of freedom in this group
[	dofSpec	4*nDoF	int	Which degrees of freedom
[	dofWeight	prec*nDoF	float	Weighting for each DoF
	nFrames	4	int	Number of frames [†]
		Then have nFrames sets of signals:		
[	nSigs	4	int	Number of source signals
[	ntSig	4	int	Number of values in each source signal
[	dtSig	prec	float	Time step in each source signal
[		Then have nSigs sets of:		
[[	nDofForSig	4	int	Number of DOF this signal affects
[[	sigType	4	int	0 - force, 1 - displacement
[[	isDoFGroup	1	int8	Should the DoF be interpreted as DoF groups[‡]
[[	dofSpec	4*nDofForSig	int	Which DoF to apply the signal to, or which DoF group number
[[	sigAmps	prec*nDofForSig	float	Amplitude multiplier for each specified DoF
[[	sig	prec*ntSig	float	Source signal

[*] v1.08 onwards
[†] v1.07 onwards. Before this, just a single frame was used.
[‡] from v1.08

		End sig sets and frame sets		
	nMeasSets	4	int	Number of history measurement sets*
	measFreq	4	int	Number of increments between measurements being recorded
	measStart	4	int	Increment at which to start recording†
		For each of nMeasSets:		
[	measSetName	20	char	Null terminated string indicating the measurement set name‡
[	nMeas	4	int	Number of traces in the set
[	isDoFGroup	1	int8	Should the measDof values be interpreted as Dof groups §
[	measDof	4*nMeas	int	Ordered list of measurement DOFs
	nFieldStores	4	int	Number of times we store the field
	fieldStoreIncs	4*nFieldStores	int	Increments where we store the field values (i.e. all dofs).
	nFieldStoreNodes	4	uint	Number of nodes from which field data is stored; set to 0 for none, 0xFFFFFFFF for all.¶
	fieldStoreNodes	4*nFieldStoreNodes	uint	Nodes where data is stored; unused all or none set in nFieldStoreNodes

*From v1.05. Before this, was just nMeas, number of measurement DOF.
†From v1.04
‡This and next values are omitted pre-v1.05
§Since v1.08
¶From v1.05

C.3 History output file

Extension: .pogo-hist

Variable Name	Bytes	Type	Description
header	20	char	File header, must match '%pogo-hist1.03'
prec	4	int	Precision, in bytes (4 or 8), of floating point data
nDims	4	int	Number of dimensions
nMeasSets	4	int	Number of measurement sets [*]
ntMeas	4	int	Number of values in each signal
dtMeas	prec	float	Time step in each signal
startMeas	prec	float	Starting time value[†]
	Then have nMeasSets of:		
setName	20	char	Null terminated string indicating the measurement set name[‡]
nMeas	4	int	Number of measurements in the set
	within this, have nMeas sets of:		
nodeNum	4	int	Node number or DoFGroup number[§]
nodeDoF	4	int	Node degree-of-freedom
loc	prec*nDims	float	Coordinates of node
historyTrace	prec*ntMeas	float	Time trace - displacement

C.4 Field output file

Extension: .pogo-field

Variable Name	Bytes	Type	Description

[*] Number of signals pre v1.02
[†] from v1.01
[‡] This and next values are omitted pre v1.02
[§] DoFGroup number from v1.03 onwards, nodeDoF set to -1, and loc to zero

header	20	char	File header, must match '%pogo-field1.03'
prec	4	int	Precision, in bytes (4 or 8), of floating point data
nDims	4	int	Number of dimensions
nDofPerNode	4	int	Number of DOF per node (should match nDims at present)
nNodes	4	int	Number of nodes at which we record data
pos	prec*nDims*nNodes	float	Nodal positions
nodeNums	4*nNodes	int	Node numbers*
nFieldStores	4	int	Number of frames recorded from simulation
	Then have nFieldStores sets of:		
time	prec	float	Time of frame
u	prec*nNodes	float	Displacement in certain DoF at each node
	Repeat u for all nDofPerNode		

C.5 Poly input file

The .poly file format is used to describe geometry which is then passed to the mesher. For more information, see Sect. 5.3. This format is a text format, and each row in the table below describes one line (repeated as described). Each value is separated by spaces. Each number should be entered as is. Each variable should be replaced with a number. 'pointNumber', 'segNumber' etc. should be 1, 2, 3... etc. Everything is 1-indexed.

nPoints	2	0	0	
pointNumber	x	y		[repeats nPoints times]
nSegs	0			
segNumber	startPoint	endPoint		[repeats nSegs times]
nHoles				
holeNumber	x	y		[repeates nHoles times]

*From v1.03

Appendix D

Abaqus feature support

Pogo is a standalone product. It does not require any other specific software packages to be installed (beyond libraries etc.). However, it is recognised that many users are familiar with Abaqus and wish to use Pogo to perform simulations of their Abaqus models. To this end, a program *pogoFromAb* has been written (see Sect. A.1), which enables an Abaqus input file to be converted to a Pogo input file. Clearly, however, Pogo can only support a subset of features from Abaqus; this section aims to outline what features Pogo has implemented and their limitations. It should be noted that while pogoFromAb is provided as a convenience, its behaviour is not guaranteed to work in all circumstances; users are expected to validate their own results when using it to ensure that it is doing the right thing.

D.1 Known limitations

Firstly, it must be stressed exactly what Pogo is. It is an explicit time domain elastodynamic solver in 2D and 3D, while Abaqus supports many different physical effects with various different solution techniques. The specific limitations are as follows:

- Multiple DOF requests. The converter will not allow more than one source to be applied to each single degree of freedom in the model (although note that different degrees of freedom, e.g. x and y, on a single node are not an issue). Similarly, when requesting time history values, each degree of freedom can only be requested once.
  - Note that this is now just a limitation with the converter. Recent versions of Pogo support these requests, however, these must be put into the input file through an alternative method to using the Abaqus converter.

- Axisymmetric elements. These are currently unsupported in Pogo, but may be supported at a future date subject to demand.
- Abaqus parts. These are unsupported, and are unlikely to be supported by Pogo in future.
- Multiple steps. Only a single step is supported by Pogo.
- Non-linearity. Pogo only supports linear problems (both for geometry and materials).
- Ties. Unsupported in Pogo.

D.2 Supported Abaqus keywords

Supported Abaqus keywords are listed below, along with their known limitations.

- *Node
  - Only rectangular cartesian coordinate system can be used for this
- *Nset
- *Element
- *Elset
- *Include
  - Note that Pogo only accepts single files, so pogoFromAb compiles multiple Abaqus input files into a single *.pogo-inp* file
- *Solid section
- *Material
  - Material types supported are outlined in Appendix B
- *Density
- *Elastic
  - Support for isotropic, orthotropic and fully anisotropic
- *Damping
  - Beta damping not supported
- *Orientation

  - Only used for orientating anisotropic materials in 2D at present.
  - Pogo only supports ‘definition=coordinates’ for this.
  - Values 2-4 must be 0, and values 1 and 5 must be equal (typically 1). Value 6 (rotation axis) must be 3 and value 7 must be the rotation angle in degrees.

- *Amplitude
  - Only ‘definition=tabular’ supported (‘definition=equally spaced’ should work but is untested)
- *Dynamic
  - Must specify ‘explicit’ parameter
  - Must use ‘Direct user control’ for time step
- *Bulk viscosity
  - Values must be set to zero
- *Boundary
  - ‘type=displacement’ is the only one supported
  - Specified displacements must be zero
- *Cload
  - Must have an ‘amplitude’ associated with it
- *Output, field
  - ‘number interval’ or ‘time interval’ should be specified
  - Number of time steps (defined with *dynamic keyword) must have been set previously
- *Output, history
  - ‘time interval’ or ‘frequency’ should be specified
- *Node output
- *Parameter
  - Defines parameters for use throughout the model. If the *–noparse* option (see Sect. A.1) specified, these will be ignored.

[1] P. Huthwaite, "Accelerated finite element elastodynamic simulations using the GPU," *Journal of Computational Physics*, vol. 257, no. A, pp. 687–707, 2014.

[2] R. Cook, D. Malkus, M. Plesha, and R. Witt, *Concepts and Applications of Finite Element Analysis.* 2002.

[3] P. Rajagopal, M. Drozdz, E. A. Skelton, M. J. S. Lowe, and R. V. Craster, "On the use of absorbing layers to simulate the propagation of elastic waves in unbounded isotropic media using commercially available Finite Element packages," *NDT & E International*, vol. 51, pp. 30–40, 2012.

[4] J. R. Pettit, A. Walker, P. Cawley, and M. J. S. Lowe, "A Stiffness Reduction Method for efficient absorption of waves at boundaries for use in commercial Finite Element codes.," *Ultrasonics*, vol. 54, pp. 1868–79, sep 2014.

[5] M. Drozdz, *Efficient Finite Element Modelling Of Ultrasound Waves in Elastic Media.* PhD thesis, Imperial College London, 2008.

[6] J. R. Shewchuk, "Triangle: {E}ngineering a {2D} {Q}uality {M}esh {G}enerator and {D}elaunay {T}riangulator," in *Applied Computational Geometry: Towards Geometric Engineering* (M. C. Lin and D. Manocha, eds.), vol. 1148 of *Lecture Notes in Computer Science*, pp. 203–222, Springer-Verlag, may 1996.

[7] P.-O. Persson and G. Strang, "A simple mesh generator in matlab," *SIAM review*, vol. 46, no. 2, pp. 329–345, 2004.