VirtualLab

Rhydian Lewis; Benjamin Thorpe; Llion Marc Evans

Jan 19, 2024

CONTENTS:

1 Installation **3**
- 1.1 Quick Installation . . . 3
- 1.2 Standard install . . . 4
- 1.3 Instalaltion on HPC system . . . 5
- 1.4 Testing . . . 7
- 1.5 MPI . . . 7

2 Containers **9**
- 2.1 What are they? . . . 9
- 2.2 Why do we use them? . . . 9
- 2.3 Available containers . . . 10
- 2.4 Container build status . . . 11
- 2.5 References . . . 11

3 Code Structure **13**
- 3.1 Input . . . 13
- 3.2 Scripts . . . 13
- 3.3 Materials . . . 14
- 3.4 RunFiles . . . 14
- 3.5 docs . . . 15
- 3.6 Output . . . 15
- 3.7 Containers . . . 15
- 3.8 Code configuration . . . 15

4 Virtual Experiments **17**
- 4.1 Tensile Testing . . . 17
- 4.2 Laser Flash Analysis . . . 19
- 4.3 HIVE . . . 19

5 Running VirtualLab **23**
- 5.1 The RunFile Explained . . . 23
- 5.2 Launching VirtualLab . . . 32

6 Tutorials **35**
- 6.1 Mechanical . . . 36
- 6.2 Thermal . . . 47
- 6.3 Thermo-mechanical . . . 57
- 6.4 Image-Based Simulation . . . 64
- 6.5 Mesh Voxelisation . . . 66
- 6.6 Machine learning (HIVE) . . . 76
- 6.7 Irradiation Damage . . . 102

6.8 Simulated X-ray imaging with GVXR . . . 121
6.9 Performing X-Ray CT Reconstruction . . . 135

7 Adding to VirtualLab **141**
7.1 Scripts . . . 141
7.2 Experiments . . . 141
7.3 Containers and Methods . . . 142
7.4 Amending Available Methods . . . 143
7.5 Adding New Methods . . . 143
7.6 Adding New Containers . . . 143
7.7 Contributing to VirtualLab . . . 147

8 About **151**
8.1 Support . . . 151
8.2 Attribution . . . 151
8.3 Contribute . . . 152
8.4 License . . . 152

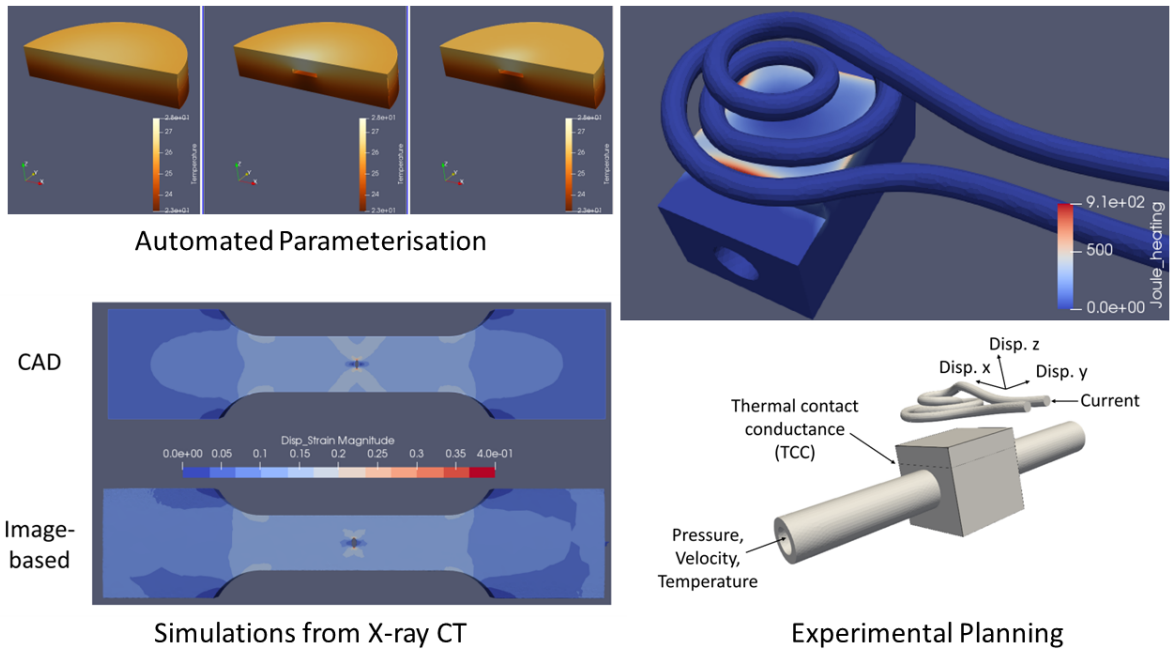
VirtualLab is a modular platform which enables the user to run simulations of physical laboratory experiments, i.e., their 'virtual counterparts'.

The motivation for creating a virtual laboratory is manyfold, for example:

- Planning and optimisation of physical experiments.
- Ability to directly compare experimental and simulation data, useful to better understand both physical and virtual methods.
- Augment sparse experimental data with simulation data for increased insight.
- Generating synthetic data to train machine learning models.

The software is mostly written in python, and is fully parametrised such that it can be run in 'batch mode', i.e., non-interactively, via the command line. This is in order to facilitate automation and so that many virtual experiments can be conducted in parallel.

Due to the modularity of the platform, by nature, **VirtualLab** is continually expanding. The bulk of the 'virtual experiments' currently included are carried out in the FE solver Code_Aster. However, there are also modules to simulate X-ray computed tomography, irradiation damage of materials and electromagnetics.



The pre and post processing is carried out using various software, for example:

- SALOME: Mesh generation
- Cad2Vox: Mesh voxelisation

- CIL: CT reconstruction
- SuRVoS: Image segmentation
- iso2mesh: Image-based meshing
- PyTorch: Data analytics
- ParaVis: Data visualisation
- Paramak: Neutronics cad developement
- OpenMC: Neutronics simulation
- MoDELib: Dislocation dynamics simulation

While this platform has been written for use from the command line, some capabilities have been included to use GUIs offered by the various software for debugging and training.

CHAPTER

ONE

INSTALLATION

The **VirtualLab** platform has been designed so that only a small number of prerequisites are needed for its use. Containers are used to house various codes and software, meaning that a containerisation tool is required, with Apptainer the chosen tool. The **VirtualLab** python package only requires gitpython (and therefore git), and can be used with either native python (including pip) or conda.

VirtualLab Supports the following operating systems:

Table 1.1: Supported OS's

Operating System	Version	Notes
Linux	Mint 19/Ubuntu 20.04+	Any reasonably modern distro should work. **VirtualLab** has been tested on various desktops and laptops running Ubuntu 20.04 LTS and a supercomputer running Redhat Linux enterprise 9. However, as with all things Linux results may vary on other distros.

As Apptainer is only available on Linux this is currently the only officially supported OS.

Note: Apptainer's website does contain instructions for using it Windows and MacOs. However, this through a virtual machine which prevents the use of GPUs for modules that support them. It also has a negative impact on performance as such we don't recommend using Apptainer on non Linux systems.

1.1 Quick Installation

You may run the following 'one line' command on a fresh installation of a supported Linux distro to install **VirtualLab** and the containerisation tool. The only requirement for this installation is that you have either python installed (including pip), or conda.

Warning: If you are not using a fresh OS it is highly recommended that you read the rest of this page before trying this method, to better understand how the installation is carried out in case you have pre-installed dependencies which might brake.

This 'one line' commant has only been tested on Ubuntu 20.04 LTS.

Terminal:

```
wget https://gitlab.com/ibsim/virtuallab/-/raw/master/Scripts/Install/Host/Install_main.
↪sh && \
chmod 755 Install_main.sh && \
./Install_main.sh -y  && \
rm Install_main.sh && \
source ~/.VLprofile
```

Running the above will install both git and apptainer along with the VirtualLab python package using standard python.

Note: Install_main.sh contains a couple of sudo commands, therefore you will be promted to enter your password.

The `-y` in the third line of the above signals to skip the installation dialogue and install **VirtualLab** in the default location `/home/$USER/VirtualLab`. This flag can be removed if youd require a non-standard install.

Note: If youd like to install **VirtualLab** using conda or to get the latest development version see here.

To test out the install follow the steps outlined here.

1.2 Standard install

To use the install/update script you will need to install git. This can be easily done by either following the instructions on git's website or, on Ubuntu based distros, you can run the following in a terminal.

```
sudo apt install git
```

Next you will need to install Apptainer. This can either be installed using the following:

```
wget https://gitlab.com/ibsim/virtuallab/-/raw/master/Scripts/Install/Host/Install_
↪Apptainer-bin.sh && \
chmod 755 Install_Apptainer-bin.sh && \
sudo ./Install_Apptainer-bin.sh -y  && \
rm Install_Apptainer-bin.sh
```

or by following the most up-to-date instructions from their website:

- Quick start
- Install Apptainer

VirtualLab is primarily command line only so you will need to run the following commands in a terminal to install the **VirtualLab** python package

```
BRANCH=master
wget https://gitlab.com/ibsim/virtuallab/-/raw/${BRANCH}/Scripts/Install/Host/Install_
↪VirtualLab.sh && \
chmod 755 Install_VirtualLab.sh && \
./Install_VirtualLab.sh -B $BRANCH  && \
rm Install_VirtualLab.sh && \
source ~/.VLprofile
```

The installer will then take you through a series of menus and download the latest version of the code.

Note: If you'd prefer to use conda instead of native python you will need to add `-I conda` to the third line, e.g.

```
./Install_VirtualLab.sh -I conda
```

This will create an environment named VirtualLab.

Note: The above will install the most recent version of **VirtualLab** from the master branch. For the most recent development version change BRANCH=dev in the above.

At this point the **VirtualLab** package will have been installed, however none of the containers it requires have yet been downloaded. These will be installed as and when they are needed for the analysis in question.

The size of these containers can be quite large. As standard, these containers will be saved to a directory named 'Containers' in the VirtualLab directory. If you'd prefer these containers be saved elsewhere, this can be changed in `VLconfig.py` file in the VirtualLab directory, see code configuration for more details.

To test out the install follow the steps outlined here.

1.3 Instalaltion on HPC system

Due to the limited number of prerequisits, installation of **VirtualLab** on HPC clusters is straight forward. The steps outlined here are for the sunbird cluster, however a similar procedure should work for other HPC systems.

Firstly, a setup file is required which contains all of the packages required for **VirtualLab** to work correctly. For sunbird, this file looks like this

```
module load anaconda/2023.03 git/2.19.2 apptainer/1.0.3 mpi/mpich/3.2.1

# sunbird specific bug fix to ensure conda works correctly
source /apps/local/languages/anaconda/2023.03/etc/profile.d/conda.sh # must match loaded␣
↪anaconda version
set -a
. /apps/local/languages/anaconda/2023.03/etc/profile.d/conda.sh
set +a

# sources .VLprofile (if its available) to ensure paths such as the VirtualLab directory␣
↪are discovarable
if [ -f "$HOME/.VLprofile" ] ; then
  source $HOME/.VLprofile
fi
```

This loads anaconda, git, apptainer and mpich (which is needed for multi-node use, see MPI for details). This file also fixes a small bug with conda specific to sunbird. It then sources `.VLprofile`, which is created as part of the **VirtualLab** install.

This file can be easily downloaded and sourced with the following

```
cd # more convenient if this file is in home directory, but doesn't have to be
wget "https://gitlab.com/ibsim/virtuallab_bin/-/raw/dev/VL_setup.sh"
source VL_setup.sh
```

On other HPC systems the versions for the packages would need to be changed to reflect the versions available.

Next **VirtualLab** can be installed with the following

```
BRANCH=master # branch to install from
wget "https://gitlab.com/ibsim/virtuallab/-/raw/"$BRANCH"/Scripts/Install/Host/Install_
↪VirtualLab.sh"
chmod 755 Install_VirtualLab.sh
./Install_VirtualLab.sh -B $BRANCH -I conda
rm Install_VirtualLab.sh
```

This installation is exactly the same as that outlined here, where more details can be found.

Test out that the installation has worked correctly with the following

```
VirtualLab --test
```

see here for more details on this.

Note: On sunbird the computing nodes do not have access to the internet, therefore any containers required for an analysis will need to be pulled and built using the login node. Therefore the above test will need to be performed on the login node.

Performing analysis using **VirtualLab** on sunbird can then performed using the a SLURM script such as this

```
#!/bin/bash --login
#SBATCH --job-name=VirtualLab
#SBATCH --output=VL%J
#SBATCH --time=0-00:10
#SBATCH --ntasks=1
#SBATCH --mem-per-cpu=6000
#SBATCH --account=scwXXXX

module purge
source ~/VL_setup.sh # gets all the necessary packages

VirtualLab -f ~/VirtualLab/Run.py # runs analysis
```

The first few lines specify the resources required to run the analysis, followed by sourcing VL_setup.sh to ensure the required packages are loaded. Following this the analysis outlined `Run.py` in the `VirtualLab` directory is performed, which is a simple tensile test.

Note: This analysis requires **salome meca** to create meshes and perform FEA, therefore this container will need to be built on the login node before hand.

A convenient method of building container is by using the `-C` option followed by the name of the containers. To build the **salome meca** container use the following command

```
VirtualLab -C SalomeMeca
```

The above SLURM script can then be submitted with the following

```
sbatch #PATH/#TO/#SLURM_FILE
```

1.4 Testing

To test out that the installation has worked as expected run the following command

`VirtualLab --test`

This will download **VirtualLab**'s manager container along with a small test container to make sure things are set up correctly. It also spits out a randomly selected programming joke as a nice whimsical bonus.

For more on how to use **VirtualLab** we recommend working through the Tutorials section.

1.5 MPI

VirtualLab is able to perform analysis on multi-node systems as well as personal computers. For this MPI is required, and needs to be compatible with the MPI installed within **VirtualLab**'s manager container, which is MPICH. To install MPICH run the following command

```
sudo apt install mpich
```

To test out that **VirtualLab** is compatible with MPI run the following

```
VirtualLab -f RunFiles/MPI_test.py
```

You should see an output similar to this (order will differ)

```
Hello! I'm rank 0 from 5 running in total...
Hello! I'm rank 2 from 5 running in total...
Hello! I'm rank 1 from 5 running in total...
Hello! I'm rank 4 from 5 running in total...
Hello! I'm rank 3 from 5 running in total...
```

Warning: GlibC issues with Ubuntu 22.04+

We note, at this stage, that there is a known bug with Salome-Meca Running in VirtualLab with Ubuntu 22.04, along with some newer versions of Fedora. If you are using these you may find you get an error containing something similar to the following: `version `GLIBC_2.34' not found (required by /.singularity.d/libs/libGLX.so.0)`

The issue is a bug in the way that the `--nv` flag loads nvidia libraries. The short version is that the `--nv` flag isn't very sophisticated when it comes to libraries. It looks for a list of library files on the host which is defined in `nvliblist.conf`. The issue is that the latest version(s) of Ubuntu are compiled against a newer version of libGLX than is included within the Salome container. This causes problems in Apptainer.

To fix this you have two options. Firstly, you can use the `-N` option to turn off the nvidia libraries. The drawback to this is that you will be running in 'software rendering mode' and thus you will not benefit from any GPU acceleration.

The second option is to use the following workaround.

1. Search for a file named `nvliblist.conf` in your installation. It should be under your Apptainer installation directory. By default this is under `/etc/apptainer`.
2. Make a back-up of this file `mv nvliblist.conf nvliblist.conf.bak`.
3. Open the file `nvliblist.conf` using a text editor.

4. Delete all of the following lines that appear `libGLX.so`, `libGLX.so.0`, `libglx.so`, `libglx.so.0` and `libGLdispatch.so`. Note, depending on you exact system, the file may not contain all of them.

Try running the Salome container again, it should work this time.

Reference: https: //github.com/apptainer/apptainer/issues/598

One caveat with this workaround, however, is that involves messing with configs that apply system wide. As such, it may have unintended side-effects with other software/containers that use Apptainer. Our team have not yet reported any issues. However, this does not mean they do not exist. Therefore, we cannot 100% guarantee you won't have any issues. This is also the reason we recommend backing up your original config in step 2, just in case. Also, for future reference, these fixes where applied to ubuntu 22.04 with Apptainer version 1.0.5. Your millage may vary with future updates.

CHAPTER

TWO

CONTAINERS

VirtualLab utilises software and codes placed in containers to perform analysis.

2.1 What are they?

If you're unfamiliar with containers, here's a quick overview from opensource.com[1]:

> *"Containers, in short, contain applications in a way that keep them isolated from the host system that they run on. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. And they are designed to make it easier to provide a consistent experience as developers and system administrators move code from development environments into production in a fast and replicable way.*
>
> *In a way, containers behave like a virtual machine. To the outside world, they can look like their own complete system. But unlike a virtual machine, rather than creating a whole virtual operating system, containers don't need to replicate an entire operating system, only the individual components they need in order to operate. This gives a significant performance boost and reduces the size of the application. They also operate much faster, as unlike traditional virtualization the process is essentially running natively on its host, just with an additional layer of protection around it."*

2.2 Why do we use them?

We have chosen containers as the main way of distributing **VirtualLab** for a number of reasons:

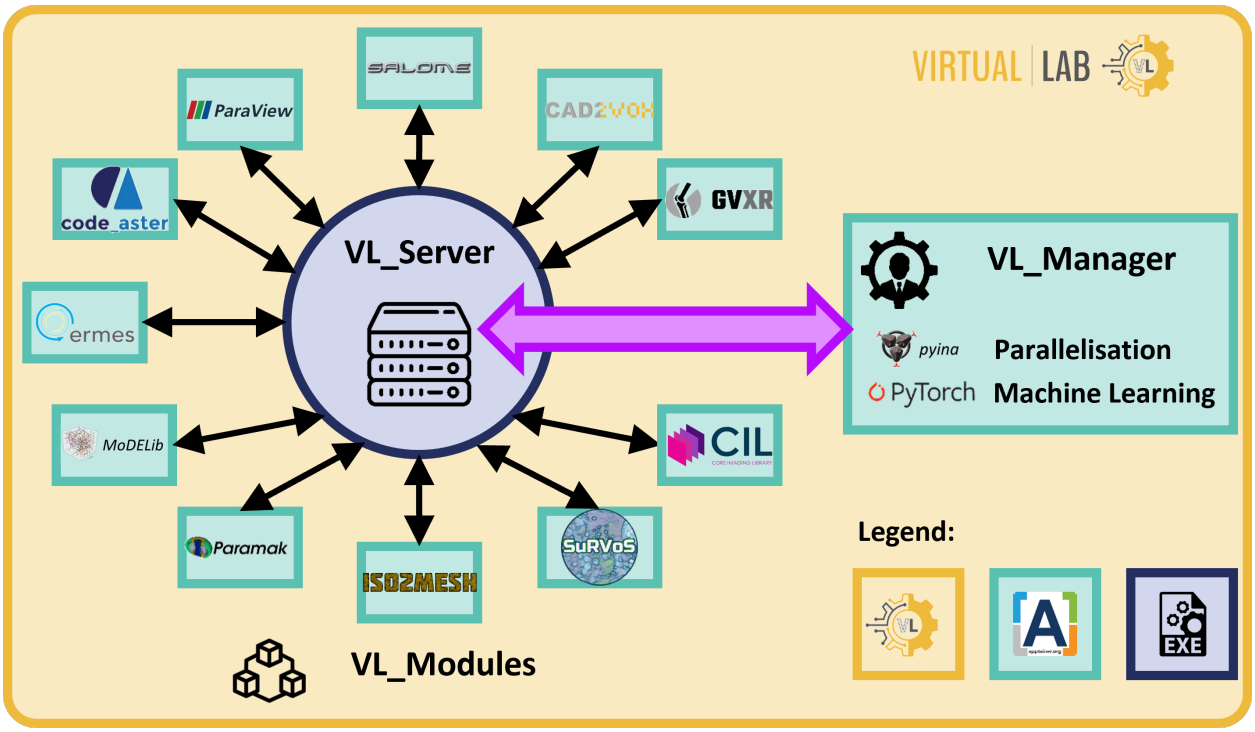
- We, the developers, take on the effort to ensure that all software dependencies are met meaning that the users can focus on getting up and running as quickly as possible.
- The portability of containers means that, whether working on a laptop or a HPC cluster, a container pull (or download) is all that's required and users' workflows can be easily moved from machine to machine when scaling up to a larger resource is required.
- The small impact on performance is far outweighed by the benefits of easy installation compared with a local installation.
- Containers offer superior performance compared with virtual machines and can make use of hardware acceleration with GPUs.
- Containers allow us to install external modules each with their own dependencies isolated from one another.

[1] What are linux containers? https://opensource.com/resources/what-are-linux-containers. Accessed: 2020-11-20.

2.3 Available containers

2.3.1 Manager

The central component of the **VirtualLab** platform is the 'Manager' container. This uses the VirtualLab python package to execute the steps of the RunFile, passing jobs to other containers via a server which runs on your local machine.



2.3.2 SalomeMeca

Container which includes the 2019 version of SalomeMeca. **SalomeMeca** is the pre and post-processing software SALOME with the Finite Element (FE) solver Code_Aster integrated within it.

This container also includes the electro-magnetic FE solver ERMES.

2.3.3 Cad2Vox

Contains the Cad2Vox package for mesh voxelisation.

2.3.4 CIL

Contains the CIL package for reconstruction of CT data.

2.4 Container build status

The below table gives an overview of the containers which are, in some way, linked with **VirtualLab** and their current build status.

Image Name	Docker Pull	Build Status	Software	Version
vl_manager	docker://ibsim/virtuallab		VirtualLab	22.0.1
vl_paramak	docker://ibsim/vl_paramak		Paramak	0.8.6
vl_openmc	docker://ibsim/vl_openmc		OpenMC	0.13.2
vl_paraview	docker://ibsim/vl_paraview		ParaView	5.11
vl_modelib_v1	docker://ibsim/vl_modelib_v1		MoDELib	1.0
vl_iso2mesh	docker://ibsim/vl_iso2mesh		iso2mesh	1.9.6
vl_cad2vox	docker://ibsim/vl_cad2vox		CAD2Vox	1.26
vl_gvxr	docker://ibsim/vl_gvxr		gVXR	2.0.2
vl_cil	docker://ibsim/vl_cil		CIL	22.1.0
vl_salomemeca	docker://ibsim/vl_salomemeca		Salome-Meca	2019.0.3
vl_aster_v14_6	docker://ibsim/vl_aster_v14_6		Code_Aster	14.6
vl_coms_test	docker://ibsim/vl_coms_test		Utils	1.0

2.5 References

CHAPTER

THREE

CODE STRUCTURE

In the **VirtualLab** directory you will find a number of sub-directories used to run the package. *Input*, *Scripts* and *Materials* are essential for any analysis performed. In addition to these you will find *RunFiles* and *docs*. Results generated by **VirtualLab** are written to the *Output* directory, which is created when the first analysis is performed.

3.1 Input

`Input` contains the parameters which will be used for running simulations, such as dimensions to create meshes and boundary conditions and materials for FE simulations.

Input has a sub-directory for each simulation type, and within those are sub-directories for each Project, e.g. `Input/$SIMULATON/$PROJECT`.

Here you will find `$PARAMETERS_MASTER.py` and `$PARAMETERS_VAR.py` files, which are explained in more detail here.

For the example of `Input/Tensile/Tutorials`:

- `$PARAMETERS_MASTER.py` = `TrainingParameters.py`
- `$PARAMETERS_VAR.py` = `Parametric_1.py`

3.2 Scripts

This directory includes the scripts required to install and launch **VirtualLab**.

3.2.1 Install

The directory `Install` contains the scripts used by the non-manual installation, which will install and configure **VirtualLab** and its dependencies such as python, **Code_Aster**, **SALOME** and **ERMES**.

3.2.2 Methods

The directory `Methods` contains files for the different methods available in **VirtualLab**, such as 'Mesh' and 'Sim'.

3.2.3 Experiments

The directory `Experiments` contain simulation-specific scripts for each experiment available, which currently are Tensile, LFA and HIVE. Inside the experiment directory are scripts, grouped by their method type, required to run that specific virtual experiment. In `Mesh` you will find **SALOME** python scripts which create the mesh of the testpiece, while in `Sim` you will find the **Code_Aster** command scripts which outline the steps followed to setup the FE simulation. Alongside these files used for PreAster and PostAster can be found. `DA` contains scripts used for data analysis.

Other simulation-specific sub-directories may also be included in the experiment directory here, such as *Laser* for the LFA simulation which contains different laser pulse profiles measured experimentally.

3.2.4 Common

`Common` contains scripts used by **VirtualLab** for any type of experiment. These includes setting up the environment through creating directories and inerfacing with the various packages incorporated, such as **SALOME** and **Code_Aster**.

3.3 Materials

This directory contains the material properties used for FE simulations.

Within this directory are sub-directories, the name of which are the different materials available. Within these sub-directories are files for the different type of material properties, e.g. 'Youngs.dat' contains information about the Youngs modulus of a material.

The data stored in these files can be a single number (used to perform linear analysis) or a list of two numbers, the first column is a varying property (e.g. Temperature) while the second column is the value of the material property at that quantity.

3.4 RunFiles

This directory contains the driver files to launch virtual experiments, referred to as a RunFile.

This directory contains a number of templates which the user may customise for their own applications, including ones specifically for each of the tutorials in `RunFiles/Tutorials`. A detailed template file `Run.py` is also included in the top-level directory of **VirtualLab** i.e. the installation location.

3.5 docs

The files required to create this documentation.

3.6 Output

This directory will be created when the first **VirtualLab** analysis is performed.

This directory is structured similarly to the *Input* directory, where you will find a directory for the *Simulation* type followed by one of the *Project* name.

The 'project directory' (`Output/$SIMULATON/$PROJECT`) will hold all data generated for the *Project*, such as: meshes; simulation results; visualisation images; analysis reports. The structure of the project directory is detailed in this section.

3.7 Containers

This directory will be created when a container is built using **VirtualLab**. This will occur when the first analysis is performed.

3.8 Code configuration

As standard, the code structure is set up for the above directories to be located in the `VirtualLab` directory, however this can be altered using `VLconfig.py` (also in the VirtualLab directory). For example, if you wanted the containers to be saved to an alternative location, then 'ContainerDir' would need to be changed to the desired location.

CHAPTER

FOUR

VIRTUAL EXPERIMENTS

4.1 Tensile Testing

A tensile test is a common test used for mechanical characterisation of materials. A sample of a carefully controlled geometry is gripped from two ends and put under tension. The load can be applied as a controlled force whilst measuring the displacement or as a controlled displacement whilst measuring the required load. This provides information about mechanical properties such as Young's modulus, Poisson's ratio, yield strength, and strain-hardening.

This methodology is so widely used that there exist many international testing standards for various materials and applications. Our initial implementation of this physical experiment as a virtual test is focused on emulating the standard BS EN ISO 6892-1:2016 for testing of metallic materials at room temperature and specifically for 'dog-bone' shaped samples. The parameterised nature of **VirtualLab** facilitates using our implementation as a template for tensile testing according to other standards or a custom test setup.

To accompany the virtual offering of the platform, the research group have also undertaken a physical experimental campaign with a batch of samples with varying parameters for direct comparison of the experimental and simulation results. This data will be made publicly available in due course.

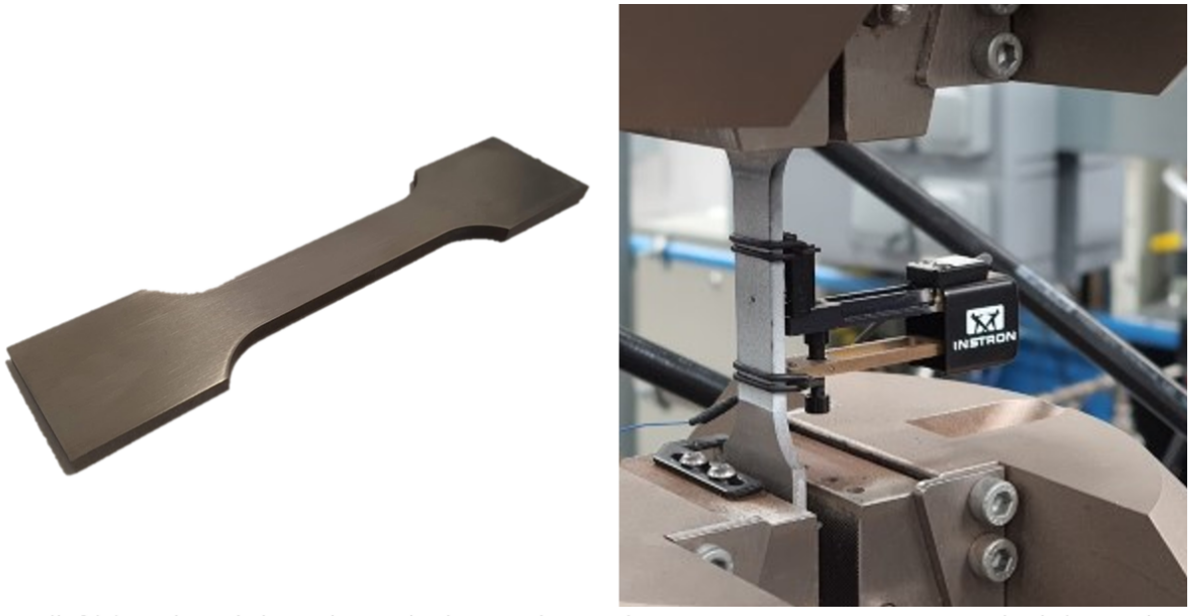


Fig. 4.1: **(left)** 'Dog bone' shaped sample designed according to BS EN ISO 6892-1:2016; **(right)** sample loaded into testing apparatus with strain measured by a gauge and digital image correlation.

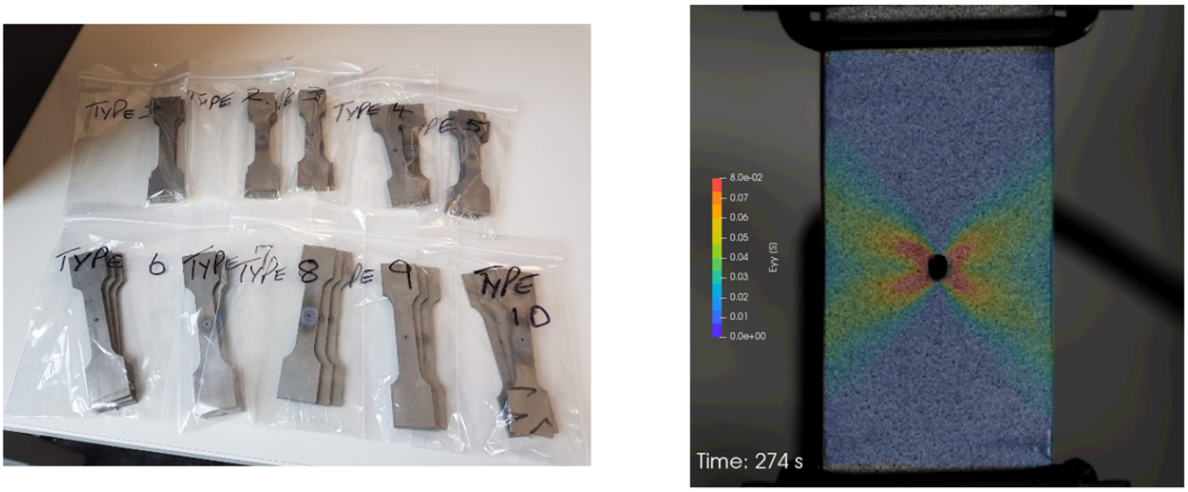


Fig. 4.2: **(left)** Photograph of batch of samples manufactured with varying parameters to test physically and compare directly with their virtual counterparts; **(right)** experimental results from digital image correlation measurements.

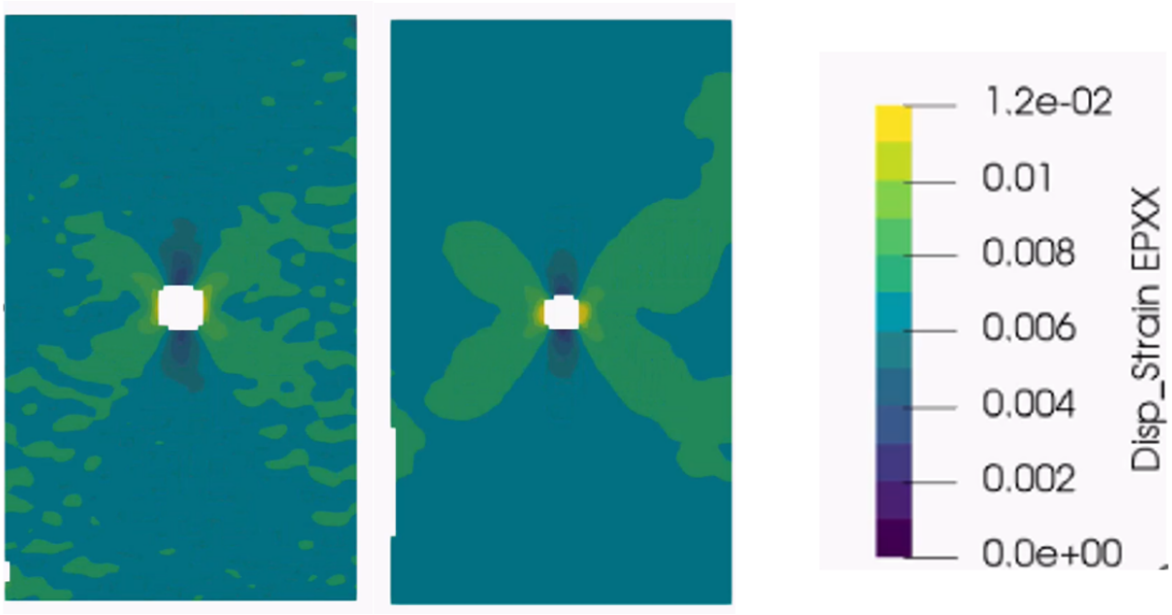


Fig. 4.3: Direct comparison of test results from the physical **(left)** and virtual **(right)** labs.

4.2 Laser Flash Analysis

Similarly, Laser flash analysis (LFA) is a commonly used test for thermal characterisation of materials. A disc shaped sample has a short laser pulse incident on one surface, whilst the temperature change is tracked with respect to time on the opposing surface. This is used to measure thermal diffusivity, which is used to calculate thermal conductivity.

We based our implementation on the testing standards ASTM E1461 / ASTM E2585 for the determination of the thermal diffusivity of primarily homogeneous isotropic solid materials. Other standards can be modelled by varying the parameters of our template.

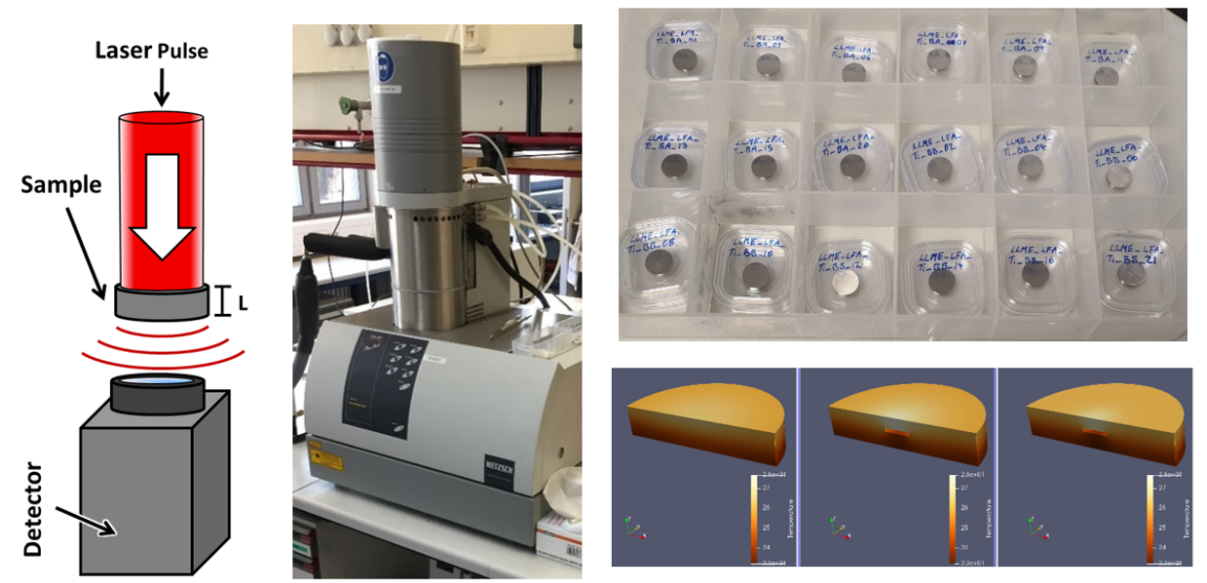


Fig. 4.4: **(left)** Schematic of LFA experimental setup; **(centre)** photograph of LFA apparatus from a physical laboratory; **(top-right)** batch of LFA samples; **(bottom-right)** results from a parameterised virtual LFA experiment.

4.3 HIVE

Heat by Induction to Verify Extremes (HIVE) is an experimental facility at the UK Atomic Energy Authority's (UKAEA) Culham site. It is used to expose plasma-facing components to the high thermal loads they will be subjected to in a fusion energy device. In this experiment, samples are thermally loaded on one surface by induction heating whilst being actively cooled with pressurised water. Further information about this custom experiment can be found in this scientific publication.

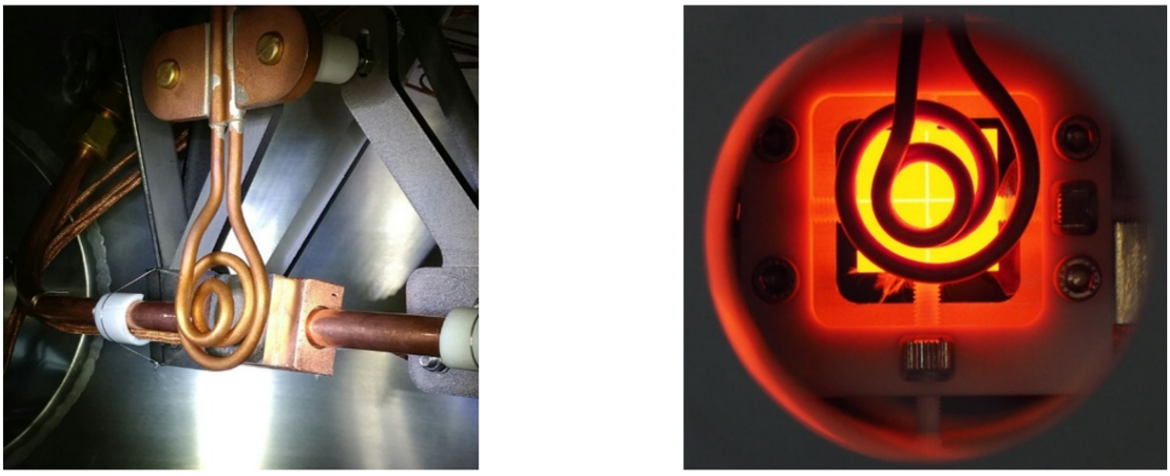


Fig. 4.5: **(left)** Photograph of sample mounted under induction coil within HIVE; **(right)** photograph of sample heated during a physical test.

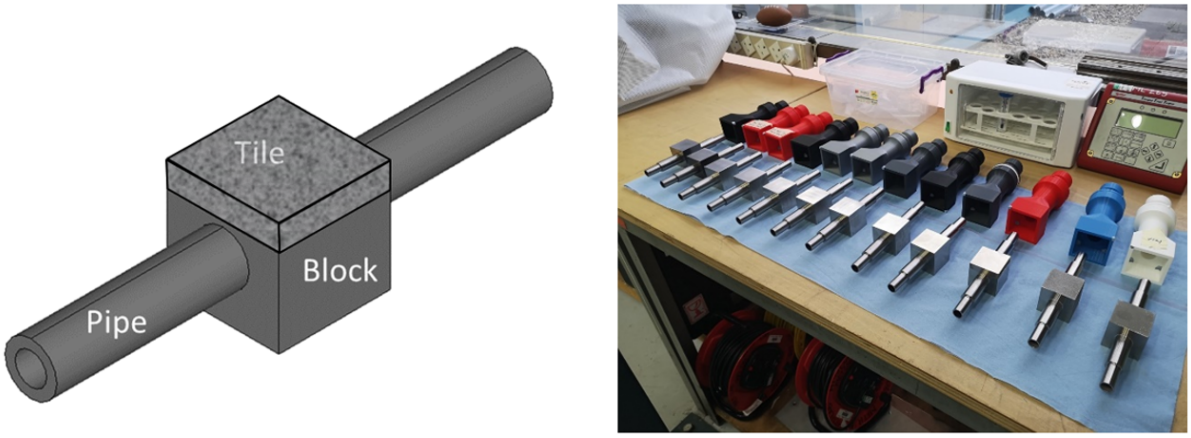


Fig. 4.6: **(left)** Schematic of sample manufactured for parameterised physical and virtual testing within HIVE; **(right)** photograph of a batch of manufactured HIVE samples.

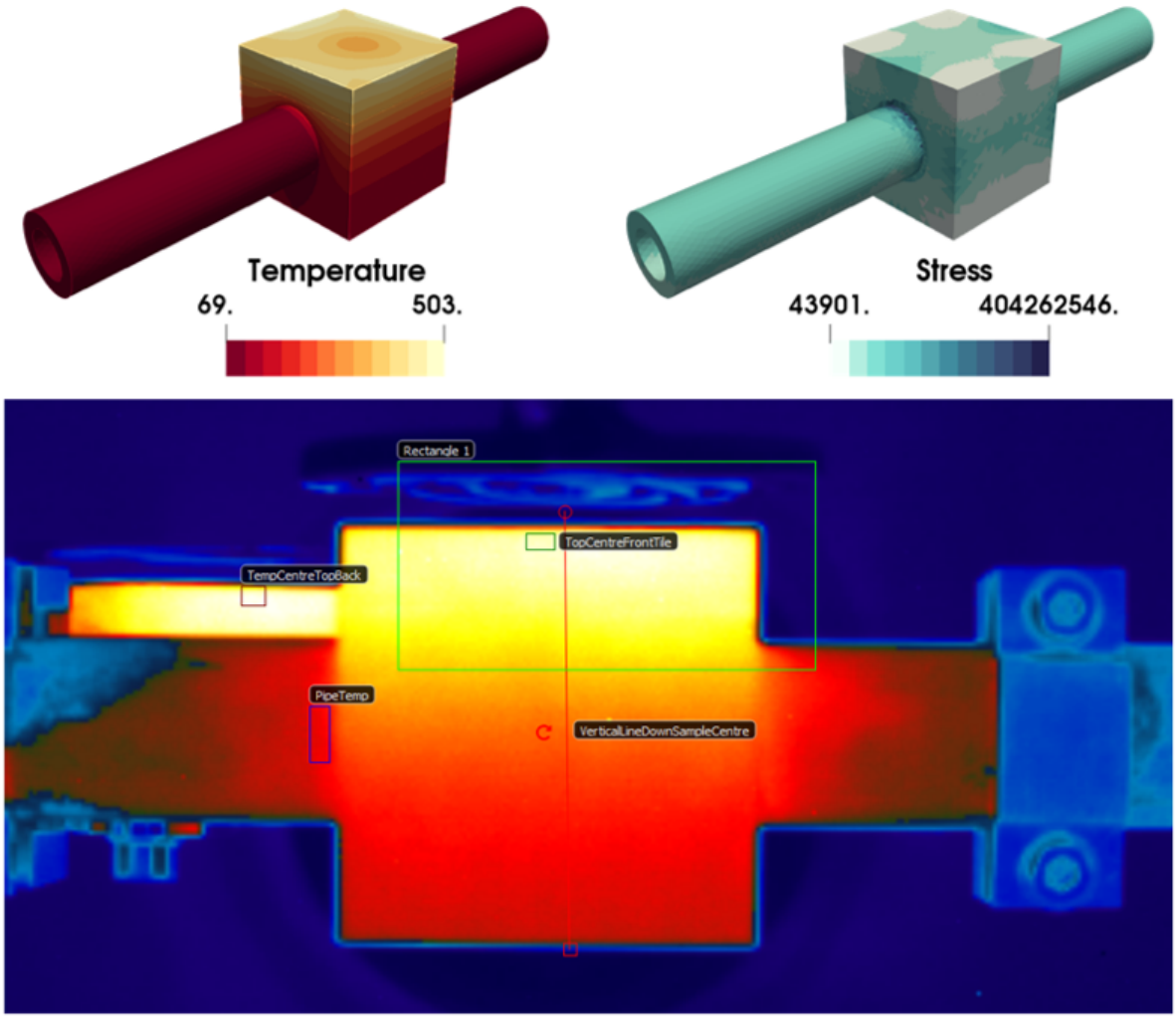


Fig. 4.7: **(top-left & top-right)** virtual testing results for temperature and stress respectively; **(bottom)** physical testing results for temperature measured by an infra-red camera.

CHAPTER

FIVE

RUNNING VIRTUALLAB

This section outlines how to run analyses using VirtualLab.

5.1 The RunFile Explained

The *RunFile* contains all the necessary information to launch analyses using **VirtualLab**. The *RunFile* is executed in strict order such that it is possible to build up a complex workflow with conditional dependencies. This does mean that it is important to carefully consider the order of the *RunFile* sections and sub-sections.

5.1.1 Template

Template

A template *RunFile* for **VirtualLab**:

```
#!/usr/bin/env python3
#=============================================================================
# Header
#=============================================================================

import sys
sys.dont_write_bytecode=True
from Scripts.Common.VirtualLab import VLSetup

#=============================================================================
# Definitions
#=============================================================================

Simulation='$TYPE'
Project='$USER_STRING'
Parameters_Master='$FNAME'
Parameters_Var='$FNAME'/None

#=============================================================================
# Environment
#=============================================================================

VirtualLab=VLSetup(
```

(continues on next page)

(continued from previous page)

```
            Simulation,
            Project
            )

VirtualLab.Settings(
            Mode='$TYPE',
            Launcher='$TYPE',
            NbJobs=$INTEGER
            )

VirtualLab.Parameters(
            Parameters_Master,
            Parameters_Var,
            RunMesh=bool,
            RunSim=bool,
            RunDA=bool,
            RunVoxelise=bool
            )

#==============================================================================
# Methods
#==============================================================================

VirtualLab.Mesh(
            ShowMesh=bool,
            MeshCheck='$MESH_NAME'/None
            )

VirtualLab.Sim(
            RunPreAster=bool,
            RunAster=bool,
            RunPostAster=bool,
            ShowRes=bool
            )

VirtualLab.DA()

VirtualLab.Voxelize()
```

5.1.2 Header

At the top of each *RunFile* is the header, common for all analyses, which includes various commands e.g. importing libraries. It is unlikely that you will need to amend this section.

```
#!/usr/bin/env python3
#==============================================================================
# Header
#==============================================================================

import sys
```

(continues on next page)

(continued from previous page)

```
sys.dont_write_bytecode=True
from Scripts.Common.VirtualLab import VLSetup
```

5.1.3 Definitions

Following this is the definitions section, where variables are defined which are compulsory to launch **VirtualLab** successfully.

Simulation

Usage:

```
Simulation = '$TYPE'
```

This is used to select the 'type' of virtual experiment to be conducted.

Types available:

`Tensile`
`LFA`
`HIVE`

For further details on each simulation see Virtual Experiments.

Project

Usage:

```
Project = '$USER_STRING'
```

User-defined field to specify the name of the project being worked on.

All data for a project is stored in the project directory located at `Output/$SIMULATION/$PROJECT`. Here you will find the sub-directory 'Meshes' which contain the meshes generated for the project, alongside results from simulations and data analyses conducted. The output generated would be:

`Output/$SIMULATION/$PROJECT/Meshes/$Mesh.Name`
`Output/$SIMULATION/$PROJECT/$Sim.Name`
`Output/$SIMULATION/$PROJECT/$DA.Name`

Parameters_Master

Usage:

```
Parameters_Master = '$FNAME'
```

Name of the file which includes values for all the required variables for the selected virtual experiment. This file must be in the directory `Input/$SIMULATION/$PROJECT`.

Note: Do not include the '.py' file extension as part of $FNAME.

The variables in this file are assigned to different `Namespaces`, which is essentially an empty class that variables can be assigned to.

Mesh

The `Mesh` namespace defines the parameters required by **SALOME** to construct a mesh, such as geometric dimensions or mesh fineness. The script `$Mesh.File.py` is executed in **SALOME** using the attributes of `Mesh` to create the geometry and subsequent mesh. This script must be in directory `Scripts/Experiments/$SIMULATION/Mesh`. The meshes will be stored in `MED` format under the name `Mesh.Name` in the 'Meshes' directory of the *Project*, i.e. `Output/$SIMULATION/$PROJECT/Meshes`.

Sim

The `Sim` namespace define the parameters needed by **Code_Aster** to perform a FE simulation. The command file `$Sim.AsterFile.comm` is executed in **Code_Aster** using the attributes of `Sim` to initiate the simulation. This script must be in directory `Scripts/Experiments/$SIMULATION/Sim`. Optional pre- and post-processing scripts can be run by specifying them in `Sim.PreAsterFile` and `Sim.PostAsterFile` respectively. These scripts, which are executed before and after the **Code_Aster** are also found in `Scripts/Experiments/$SIMULATION/Sim`. Simulation information and data will be stored in the sub-directory `Sim.Name` of the project directory, i.e. `Output/$SIMULATION/$PROJECT/$Sim.Name`.

DA

The `DA` namespace define the parameters needed to perform data analyses (DA) on the data collected from simulations. These are generally python scripts. These files can be found in `Scripts/Experiments/$SIMULATION/DA`. Like with the simulations, results for the data analyses are saved to `Output/$SIMULATION/$PROJECT/$DA.Name`.

Note: `Mesh.Name`, `Sim.Name` and `DA.Name` can be written as paths to save in to sub folders of a project directory, i.e. `Sim.Name` = 'Test/Simulation' will create a sub-directory 'Test' in the project directory.

Parameters_Var

Usage:

```
Parameters_Var = {'$FNAME'/None}
```

Name of the file which includes value ranges for particular variables of the user's choice. This is used in tandem with *Parameters_Master*.

Variables defined here are usually a sub-set of those in *Parameters_Master*, with the values specified here overwriting those in the master.

Value ranges for given variables are used to perform parametric analyses, where multiple 'studies' are conducted.

As in *Parameters_Master*, values will be assigned to the `Namespaces` `Mesh`, `Sim` and `DA`. This file is also in `Input/$SIMULATION/$PROJECT`.

If set to `None` a single study is run using the values defined in *Parameters_Master*.

Please see the Tutorials to see this in action.

Note: Do not include the '.py' file extension as part of $FNAME.

5.1.4 Environment

The next section is for setting the **VirtualLab** environment. That is, how the user would like to interact with **VirtualLab** and how it should make use of the available hardware. It is necessary to create the environment before starting any *Methods*. However, it is possible to change the envrionment later in the *RunFile* as part of the workflow. For example, it may be desirable to only have a single job during meshing but multiple jobs for the simulation if performing a parameter sweep of boundary conditions with the same geometry.

VLSetup

`VLSetup` takes the previously set *Definitions* to start building the environment. It is unlikely that you will need to amend this section.

```
VirtualLab=VLSetup(
           Simulation,
           Project
           )
```

VirtualLab.Settings

This is an optional attribute of **VirtualLab** where settings can be changed.

```
VirtualLab.Settings(
           Mode='Headless',
           Launcher='Process',
           NbJobs=1
           )
```

Mode

Usage:

```
Mode = '$TYPE' (str, optional)
```

This dictates how much information is printed in the terminal during the running of **VirtualLab**. Options available are:

- 'Interactive' - Prints all output to individual pop-up terminals (currently not in use due to a change in X-Window implementation).
- 'Terminal' - Prints all information to a single terminal.
- 'Continuous' - Writes the output to a file as it is generated.
- 'Headless' - Writes output to file at the end of the process. (Default)

Launcher

Usage:

```
Launcher = '$TYPE' (str, optional)
```

This defines the method used to launch the **VirtualLab** study. Currently available options are:

- 'Sequential' - Each operation is run sequentially (no parallelism).
- 'Process' - Parallelism for a single node only. (Default)
- 'MPI' - Parallelism over multiple nodes.

NbJobs

Usage:

```
NbJobs = $INTEGER (int, optional)
```

Defines how many of the studies that will run concurrently when using either the 'process' or 'MPI' launcher. Default is 1.

VirtualLab.Parameters

This function creates the parameter files defined using *Parameters_Master* and *Parameters_Var*. It also performs some checks, such as checking defined files exist in their expected locations, i.e., *Parameters_Master*, *Parameters_Var* and the files specified therein (Mesh.File, Sim.AsterFile etc.).

```
VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=True,
           RunSim=True,
           RunDA=True,
           RunVoxelise=True
           )
```

In addition to the parameter files and performing checks of associated file, it is possible to define whether particular *Methods* should run or not. By default, any method which is included in the later method section will run unless explicitly defined not to here.

Usage:

```
Run$METHOD = bool (optional)
```

Indicates whether or not the method will be run. Default is `True`. Currently available options are:

- Mesh - For geometry creation and meshing.
- Sim - For running simulations.
- DA - For data analysis of results.
- Vox - For voxelisation of meshes.

5.1.5 Methods

This section is where the bulk of the activity of **VirtualLab** occurs. That is, until now, we have only put in place the necessary information to initiate a task. The methods section controls precisely which tasks **VirtualLab** will perform. They can be simple one step sequential tasks or highly complex parallelised tasks making use of multiple software packages.

`VirtualLab.Mesh`

This is the meshing routine. In fact, this routine first generates the CAD geometry from a set of parameters and then meshes it ready for simulation. The mesh(es) defined using `Mesh` in *Parameters_Master* and *Parameters_Var* are created and saved to the sub-directory 'Meshes' in the project directory along with a file detailing the variables used for their creation. If RunMesh is set to `False` in *VirtualLab.Parameters* then this routine is skipped. This may be useful when different simulation parameters are to be used on a pre-existing mesh.

```
VirtualLab.Mesh(
            ShowMesh=False,
            MeshCheck=None
            )
```

ShowMesh

Usage:

```
ShowMesh = bool (optional)
```

Indicates whether or not to open created mesh(es) in the **SALOME** GUI for visualisation to assess their suitability. **VirtualLab** will terminate once the GUI is closed and no simulation will be carried out. Default is `False`.

MeshCheck

Usage:

```
MeshCheck = '$MESH_NAME'/None (optional)
```

'$MESH_NAME' is constructed in the **SALOME** GUI for debugging. Default is None.

`VirtualLab.Sim`

This function is the simulation routine. The simulation(s) defined using `Sim` in *Parameters_Master* and *Parameters_Var* are carried out with the results saved to the project directory. This routine also runs the pre- and post-processing scripts, if they are provided. If RunSim is set to `False` in *VirtualLab.Parameters* then this routine is skipped.

```
VirtualLab.Sim(
            RunPreAster=True,
            RunAster=True,
            RunPostAster=True,
            ShowRes=False
            )
```

RunPreAster

Usage:

```
RunPreAster = bool (optional)
```

Indicates whether or not to run the optional pre-processing script provided in *Sim.PreAsterFile*. Default is `True`.

RunAster

Usage:

```
RunAster = bool (optional)
```

Indicates whether or not to run the **Code_Aster** script provided in `Sim.AsterFile`. Default is `True`.

RunPostAster

Usage:

```
RunPostAster = bool (optional)
```

Indicates whether or not to run the optional post-processing script provided in `Sim.PostAsterFile`. Default is `True`.

ShowRes

Usage:

```
ShowRes = bool (optional)
```

Visualises the .rmed results file(s) produced by **Code_Aster** through the **ParaVis** module in **SALOME**. Default is `False`.

VirtualLab.DA

This function is the data analysis routine. The analyses, defined using the namespace `DA` in *Parameters_Master* and *Parameters_Var*, are carried out. The results are saved to `Output/$SIMULATION/$PROJECT`. If RunDA is set to `False` in *VirtualLab.Parameters* then this routine is skipped.

VirtualLab.Voxelize

This function is the routine to call **Cad2Vox**. The parameters used for the Voxelization process are defined in the namespace `Vox` in *Parameters_Master* and *Parameters_Var*. The resultant output images are saved to `Output/$SIMULATION/$PROJECT/Voxel-Images`. If RunVoxelise is set to `False` in *VirtualLab.Parameters* then this routine is skipped.

5.1.6 Example

Example

An example *RunFile* for **VirtualLab** which will run a virtual tensile test:

```
#!/usr/bin/env python3
#=============================================================================
# Header
#=============================================================================

import sys
sys.dont_write_bytecode=True
from Scripts.Common.VirtualLab import VLSetup

#=============================================================================
# Definitions
#=============================================================================

Simulation='Tensile'
Project='Tutorials'
Parameters_Master='TrainingParameters'
Parameters_Var=None

#=============================================================================
# Environment
#=============================================================================

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Terminal',
           Launcher='Process',
           NbJobs=1
           )

VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=True,
           RunSim=True,
           RunDA=False,
           RunVoxelise=False
           )

#=============================================================================
# Methods
#=============================================================================

VirtualLab.Mesh()
```

(continues on next page)

(continued from previous page)

```
VirtualLab.Sim(
           ShowRes=True
           )

VirtualLab.DA()

VirtualLab.Voxelize()
```

5.2 Launching VirtualLab

5.2.1 Command Line Interface

If **VirtualLab** has been installed correctly, the main program will have been added to your system <path>. In this case, it is possible to call **VirtualLab** from the terminal (also known as command line interface, CLI) or a bash script from any location in your system. To facilitate automation, **VirtualLab** has purposefully been designed to run without a graphical user interface (GUI).

Usage of **VirtualLab**:

```
VirtualLab -f <path>
```

More options:

`-f <path>` : Where <path> points to the location of the python RunFiles (this must be either an absolute path or relative to the current working directory).

`-K <Name=Value>`: Overwrite the value specified for variables/keyword arguments specified in the *Run* file.

`-N` : Flag to turn on/off NVIDIA GPU support.

`--tcp_port`: tcp port to use for server communication, default is 9000.

`--dry-run` : Flag to update containers without running simulations.

`--debug` : print debug messages for networking.

`--test` : Launch a small container to test installation and communication.

`-h` : Display the help menu.

Note: The default behaviour is to exit if no <path> is given.

Batch Mode

In batch mode, rather than launching the command directly it is normally entered within a script which is sent to a job scheduler (or workload manager). The command is then put in a queue to be executed when the requested resources become available. Apptainer is often the container platform of choice for shared HPC resources because it can be used without the user needing admin privileges. This is an example for the slurm job scheduler on Supercomputing Wales's sunbird system.

Apptainer

```
#!/bin/bash --login
#SBATCH --job-name=VirtualLab
#SBATCH --output=logs/VL.out.%J
#SBATCH --error=logs/VL.err.%J
#SBATCH --time=0-01:00
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --mem-per-cpu=6000

# Example batch script used to run VirtualLab.

module purge
module load apptainer/#VersionNumber
module load mpi/#MPIVersion # Optional, only required if one wants to run on multiple␣
↪nodes (set launcher to mpi if so)
source ~/.VLprofile # make sure VirtualLab/bin is in $PATH

VirtualLab -f <Path/To/File> -K Mode=H NbJobs=4 Launcher=(process/mpi)
```

5.2.2 Virtual Machines

Once logged into the VM the user is presented with an Ubuntu desktop environment which can be used identically to a native Linux installation. That is, with the use of the CLI in a terminal **VirtualLab** may be launched as detailed in *Usage*. The main limitation is that no methods requiring a GPU will be possible, and those which can use a GPU for acceleration will not be able to make use of this option.

CHAPTER

SIX

TUTORIALS

The tutorials in this section provide an overview in to running a 'virtual experiment' using **VirtualLab**.

These examples give an overview of:

- how meshes and simulations can be created parametrically without the need for a graphical user interface (GUI).
- the available options during simulations that give the user a certain degree of flexibility.
- methods of debugging.
- **VirtualLab**'s in-built pre- and post-processing capabilities.

There is a tutorial for each of **VirtualLab**'s virtual experiments.

Before starting the tutorials, it is advised to first read the Code Structure and Running VirtualLab sections for an overview of **VirtualLab**. Then it is best to work through the tutorials in order as each will introduce new tools that **VirtualLab** has to offer.

These tutorials assume a certain level of pre-existing knowledge about the finite element method (FEM) as a prerequisite. Additionally, these tutorials do not aim to teach users on how to use the **Code_Aster** software itself, only its implementation as part of **VirtualLab**. For **Code_Aster** tutorials we recommend the excellent website feaforall.com. Because **VirtualLab** can be run completely from scripts, without opening the **Code_Aster** graphical user interface (GUI), **VirtualLab** can be used without being familiar with **Code_Aster**.

'Setting up data for visualisation' is outside the scope of these tutorials. The **ParaVis** module within **SALOME** is based on another piece of open-source software called **ParaView**. If you would like to learn more about how to visualise datasets with **SALOME** it is recommended that you follow the tutorials available on feaforall.com and paraview.org.

Each tutorial is structured as follows: firstly, the experimental test sample (i.e. geometry domain) is introduced followed by an overview of the boundary conditions and constraints being applied to the sample to emulate the physical experiment. Then a series of tasks are described to guide the user through various stages with specific learning outcomes.

Simulations are initiated by launching **VirtualLab** in the command line with a RunFile specified using the flag `-f`:

```
VirtualLab -f </PATH/TO/RUNFILE>
```

`Run.py` in the **VirtualLab** top level directory is a template of a *RunFile* which is used to launch **VirtualLab**. Additional examples of *RunFiles* are available in the RunFiles directory, where the file `RunTutorials.py` is located which will be used for these tutorials.

Note: To help with following the tutorials, certain keyword arguments (referred to as kwargs) have been changed from their default values in `RunTutorials.py`. In VirtualLab.Settings *Mode* has been changed to 'Interactive', while in VirtualLab.Sim *ShowRes* is set to `True`.

Tip: You may wish to save a backup of `RunTutorials.py` such that you may return to the default template without needing to re-download it.

6.1 Mechanical

6.1.1 Introduction

A virtual experiment of the standard mechanical tensile test is performed using a linear elastic model.

In this experiment a 'dog-bone' shaped sample is loaded either through constant force, measuring the displacement, or constant displacement, measuring the required load. This provides information about mechanical properties such as Young's elastic modulus.

Action

The *RunFile* `RunTutorials.py` should be set up correctly for this simulation:

```
#==========================================================================
# Definitions
#==========================================================================
Simulation='Tensile'
Project='Tutorials'
Parameters_Master='TrainingParameters'
Parameters_Var=None

#==========================================================================
# Environment
#==========================================================================

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbJobs=1
           )

VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=True,
           RunSim=True,
           RunDA=True
           )

#==========================================================================
# Methods
```

(continues on next page)

(continued from previous page)

```
#============================================================================

VirtualLab.Mesh(
            ShowMesh=False,
            MeshCheck=None
            )

VirtualLab.Sim(
            RunPreAster=True,
            RunAster=True,
            RunPostAster=True,
            ShowRes=True
            )

VirtualLab.DA()
```

The setup above means that the path to the *Parameters_Master* file used is `Input/Tensile/Tutorials/TrainingParameters.py`. Open this example python file in a text editor to browse its structure.

Before any definitions are made, you will notice the import statement:

```
from types import SimpleNamespace as Namespace
```

A `Namespace` is essentially an empty *class* that *attributes* can be assigned to.

The `Namespace` `Mesh` and `Sim` are created in *Parameters_Master* in order to assign attributes to for the meshing and simulation stages, respectively. Since `DA` is not defined in *Parameters_Master* no data analysis will take place.

6.1.2 Sample

`Mesh` contains all the variables required by **SALOME** to create the CAD geometry and subsequently generate its mesh.

```
Mesh.Name = 'Notch1'
Mesh.File = 'DogBone'
```

Mesh.File defines the script used by **SALOME** to generate the mesh, which in this case is `Scripts/Experiments/Tensile/Mesh/DogBone.py`.

Once the mesh is generated it will be saved to the sub-directory `Meshes` of the project directory as a `MED` file under the user specified name set in *Mesh.Name*. In this instance the mesh will be saved to `Output/Tensile/Tutorials/Meshes/Notch1.med`.

The attributes of `Mesh` used to create the sample geometry in `DogBone.py` are:

```
# Geometric Parameters
Mesh.Thickness = 0.003
Mesh.HandleWidth = 0.024
Mesh.HandleLength = 0.024
Mesh.GaugeWidth = 0.012
Mesh.GaugeLength = 0.04
Mesh.TransRad = 0.012
Mesh.HoleCentre = (0.0,0.0)
```

(continues on next page)

(continued from previous page)

```
Mesh.Rad_a = 0.0005
Mesh.Rad_b = 0.001
```

The interpretation of these attributes in relation to the sample is shown in Fig. 6.1.

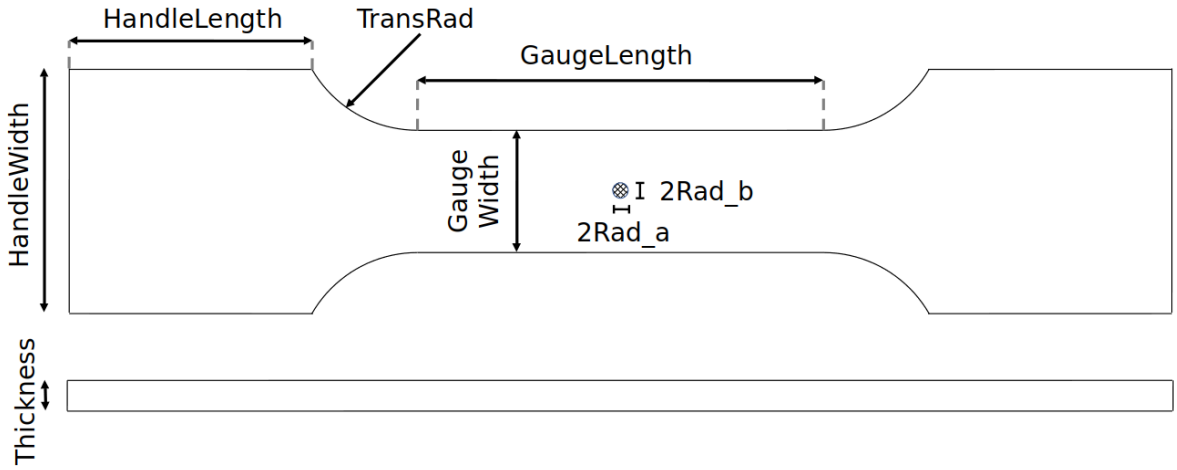


Fig. 6.1: Drawing of the 'dog-bone' sample with the attributes of `Mesh` used to specify the dimensions.

2Rad_a and 2Rad_b refer to the radii of an elliptic hole machined through a point offset from the centre by *HoleCentre*. The attribute *TransRad* is the radius of the arc which transitions from the gauge to the handle.

The remaining attributes relate to the mesh refinement parameters:

```
# Meshing Parameters
Mesh.Length1D = 0.001
Mesh.Length2D = 0.001
Mesh.Length3D = 0.001
Mesh.HoleSegmentN = 30
```

Length1D, *2D* and *3D* specify the discretisation size (or target seeding distance) along the edges, faces and volumes respectively, while *HoleSegmentN* specifies the number of segments the circumference of the hole is divided into.

The attributes of `Mesh` used to create the CAD geometry and its mesh are stored in `Notch1.py` alongside the `MED` file in the `Meshes` directory.

6.1.3 Simulation

The attributes of `Sim` are used by **Code_Aster** and by accompanying pre/post-processing scripts:

```
Sim.Name = 'Single'
Sim.AsterFile = 'Tensile'
```

Sim.Name specifies the name of the sub-directory in `Output/Tensile/Tutorials/` into which all information relating to the simulation will be stored. The file `Parameters.py`, containing all attributes of `Sim`, is saved here along with the output generated by **Code_Aster** and any pre/post-processing stages.

The attribute *Sim.AsterFile* specifies the file used by **Code_Aster** to run a virtual experiment, which in this case is `Scripts/Experiments/Tensile/Sim/Tensile.comm`. The extension `.comm` is short for command, which is the file extension for scripts used by the **Code_Aster** software.

The attributes used by **Code_Aster** are:

```
Sim.Mesh = 'Notch1'
Sim.Force = 1000000
Sim.Displacement = 0.01
Sim.Materials = 'Copper'
```

Sim.Mesh specifies which mesh is used in the simulation.

The attribute *Force* specifies the magnitude, in Newtons, which is used to load the sample during the force-controlled simulation, while *Displacement* specifies the enforced displacement, in metres, which is applied during the forced displacement simulation.

Note: If both *Force* and *Displacement* are attributed to `Sim` then both force-controlled and displacement-controlled simulations are run. If, for example, you only wish to run a constant force simulation, then this can be achieved either by removing the attribute *Displacement* or by setting it to zero.

The attribute *Materials* specifies the material the sample is composed of.

In this instance, since `Sim` has neither the attributes *PreAsterFile* or *PostAsterFile*, no pre or post processing will be carried out.

6.1.4 Task 1: Running a simulation

Due to *Parameters_Var* being set to `None`, a single mesh and simulation will be run using the information from *Parameters_Master*.

The mesh generated for this simulation is 'Notch1', while the name for the simulation is 'Single', given by *Sim.Name*. All information relating to the simulation will be saved to the simulation directory `Output/Tensile/Tutorials/Single`.

Since *Force* and *Displacement* are attributes of `Sim` a force-controlled simulation (with magnitude 1000000N) is run, along with a displacement controlled simulation (with enforced displacement 0.01m). The material properties of copper will be used for the simulation.

With *Mode* set to 'Interactive' in the setup section of `RunTutorials.py`, when launching **VirtualLab** firstly you will see information relating to the mesh printed to the terminal, e.g. the number of nodes and location the mesh is saved, followed by the **Code_Aster** output messages for the simulation printing in a separate xterm window, see Fig. 6.2.

Note: Due to a change in X-Window implementation, 'Interactive' is currently not in use within **VirtualLab**. However, if set, it will behave the same as using 'Terminal'. That is, output will be printed to the main terminal.

Action

Launch your first **VirtualLab** simulation by executing the following command from command line (CL) of the terminal whilst within the **VirtualLab** directory:

```
VirtualLab -f RunFiles/RunTutorials.py
```

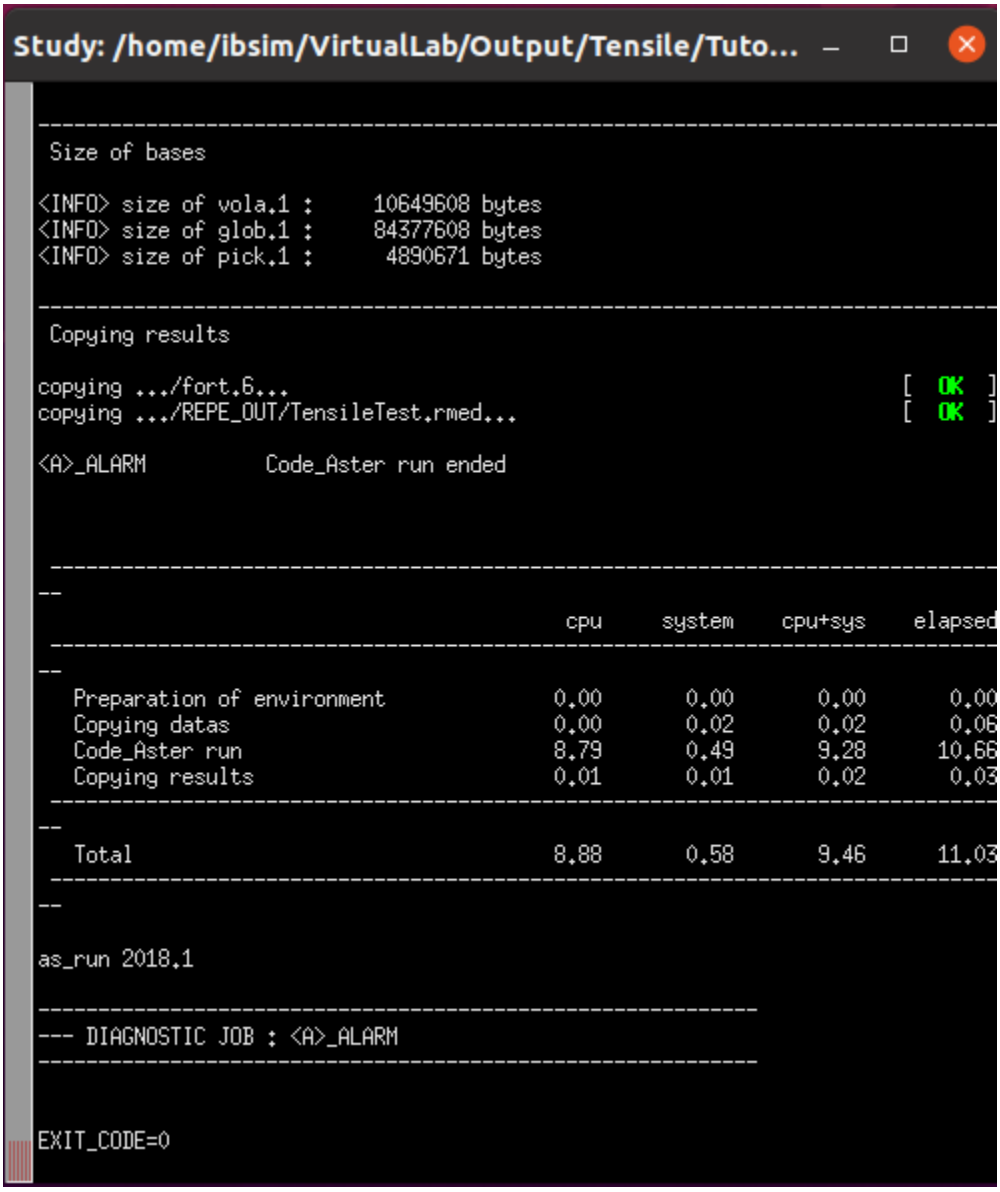


Fig. 6.2: Xterm window which opens if **VirtualLab** is set to run with *Mode* as 'Interactive'.

Running this simulation will create the following outputs:

- `Output/Tensile/Tutorials/Meshes/Notch1.med`
- `Output/Tensile/Tutorials/Meshes/Notch1.py`
- `Output/Tensile/Tutorials/Meshes/Notch1.log`
- `Output/Tensile/Tutorials/Single/Parameters.py`
- `Output/Tensile/Tutorials/Single/Aster/Export`
- `Output/Tensile/Tutorials/Single/Aster/AsterLog`
- `Output/Tensile/Tutorials/Single/Aster/TensileTest.rmed`
- `Output/Tensile/Tutorials/Single/Output.log`

The first two output files relate to the mesh generated. The `.med` file contains the mesh data, while the attributes of `Mesh` are saved to the `.py` file.

The remaining outputs are all saved to the simulation directory. `Parameters.py` contains the attributes of `Sim` which has been used for the simulation.

The file `Aster/Export` was used to launch **Code_Aster** and contains information on how it was launched. `Aster/AsterLog` is a log file containing the **Code_Aster** output, which is the same information shown in the xterm window. The file `Aster/TensileTest.rmed` contains the results generated by **Code_Aster**. Since both *Force* and *Displacement* attributes were specified the results for both are stored in this file.

Note: The file extension `.rmed` is short for 'results-MED' and is used for all **Code_Aster** results files.

Because *ShowRes* is set to `True` in VirtualLab.Sim, `TensileTest.rmed` is opened in **ParaVis** for visualisation automatically. Here you will be able to view the following fields, see Fig. 6.3 and Fig. 6.4:

`Force_Displacement` - Displacement for constant force simulation.

`Force_Stress` - Stress for constant force simulation.

`Disp_Displacement` - Displacement for constant displacement simulation.

`Disp_Stress` - Stress for constant displacement simulation.

Note: You will need to close the xterm window once the simulation has completed for the results to open in **ParaVis**.

File structure hierarchy

Location of the key files and directories for Task 1 of this tutorial:

```
| VirtualLab
| ├── .log
| ├── Config
| ├── Containers
| ├── Input
|     ├── HIVE
|     ├── LFA
|     └── Tensile
|         └── Tutorials
|             └── TrainingParameters.py
```

(continues on next page)

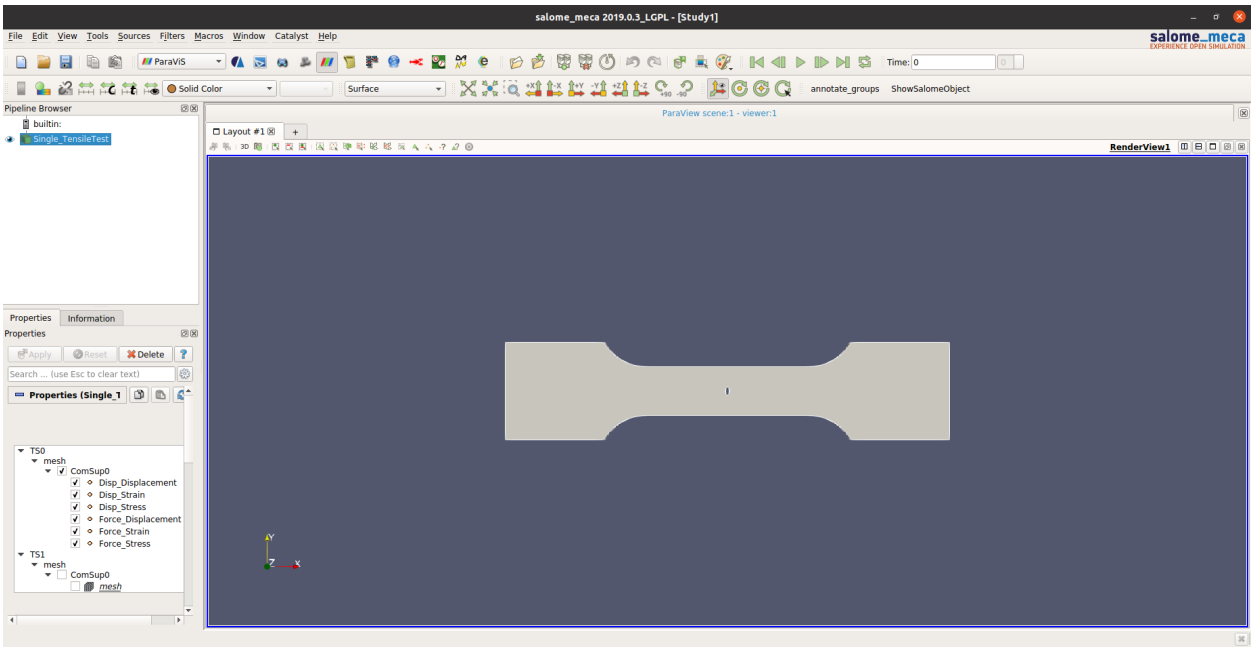


Fig. 6.3: **ParaVis** visualisation of sample as seen when opened automatically with *ShowRes* set to `True`.

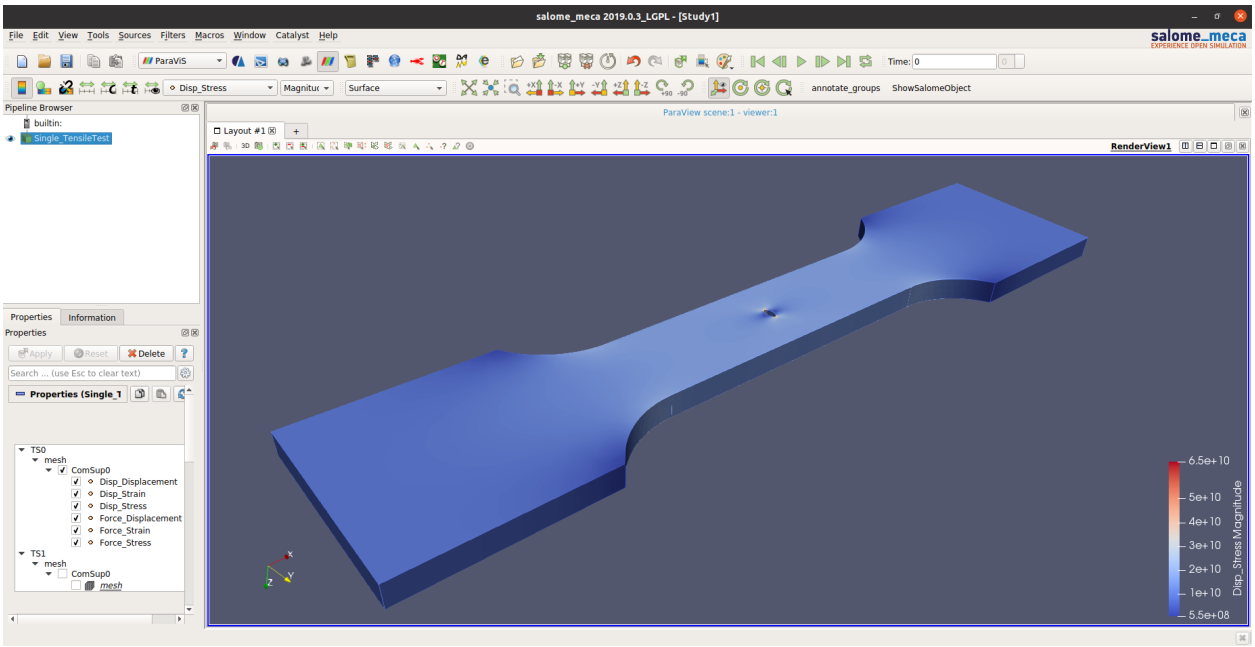


Fig. 6.4: **ParaVis** visualisation of Von Mises stress in the sample for a displacement controlled virtual experiment.

(continued from previous page)

```
|   ├── Materials
|   |   ├── Copper
|   |   |   ├── Alpha.dat
|   |   |   ├── Cp.dat
|   |   |   ├── Lambda.dat
|   |   |   ├── PoisRat.dat
|   |   |   ├── Rho.dat
|   |   |   └── Youngs.dat
|   |   ├── Copper_NL
|   |   ├── Tungsten
|   |   └── Tungsten_NL
|   ├── Output
|   |   └── Tensile
|   |       └── Tutorials
|   |           ├── Meshes
|   |           |   ├── Notch1.log
|   |           |   ├── Notch1.med
|   |           |   └── Notch1.py
|   |           └── Single
|   |               ├── Aster
|   |               |   ├── AsterLog
|   |               |   ├── Export
|   |               |   └── TensileTest.rmed
|   |               ├── Output.log
|   |               └── Parameters.py
|   ├── RunFiles
|   |   └── RunTutorials.py
|   ├── Scripts
|   |   ├── Common
|   |   ├── Experiments
|   |   |   ├── HIVE
|   |   |   ├── LFA
|   |   |   └── Tensile
|   |   |       ├── Mesh
|   |   |       |   └── DogBone.py
|   |   |       └── Sim
|   |   |           └── Tensile.comm
|   |   ├── Install
|   |   └── Methods
|   ├── bin
|   ├── docs
|   └── tests
|
|
```

6.1.5 Task 2: Running Multiple Simulations

The next step is to run multiple simulations. This is achieved using *Parameters_Var* in conjunction with *Parameters_Master*.

The *Parameters_Var* file `Input/Tensile/Tutorials/Parametric_1.py` will be used to create two different meshes which are used for simulations. Firstly, you will see value ranges for *Mesh.Rad_a* and *Mesh.Rad_b* along with the *Name* for each mesh:

```
Mesh.Name = ['Notch2','Notch3']
Mesh.Rad_a = [0.001,0.002]
Mesh.Rad_b = [0.001,0.0005]
```

Any attributes of `Mesh` which are not included in the *Parameters_Var* file will instead use the values from *Parameters_Master*. For example, 'Notch2' will have the attributes:

```
Mesh.Name = 'Notch2'
Mesh.File = 'DogBone'

Mesh.Thickness = 0.003
Mesh.HandleWidth = 0.024
Mesh.HandleLength = 0.024
Mesh.GaugeWidth = 0.012
Mesh.GaugeLength = 0.04
Mesh.TransRad = 0.012
Mesh.HoleCentre = (0.0,0.0)
Mesh.Rad_a = 0.001
Mesh.Rad_b = 0.001

Mesh.Length1D = 0.001
Mesh.Length2D = 0.001
Mesh.Length3D = 0.001
Mesh.HoleSegmentN = 30
```

Simulations will then be performed for each of these samples:

```
Sim.Name = ['ParametricSim1', 'ParametricSim2']
Sim.Mesh = ['Notch2', 'Notch3']
```

In this instance, only the simulation geometry (hole radii) will differ between 'ParametricSim1' and 'ParametricSim2'.

The results for both simulations will be opened in **ParaVis**. The results will be prefixed with the simulation name for clarity, see Fig. 6.5.

Action

Change *Parameters_Var* in the *RunFile*:

```
Parameters_Var='Parametric_1'
```

Launch **VirtualLab**:

```
VirtualLab -f RunFiles/RunTutorials.py
```

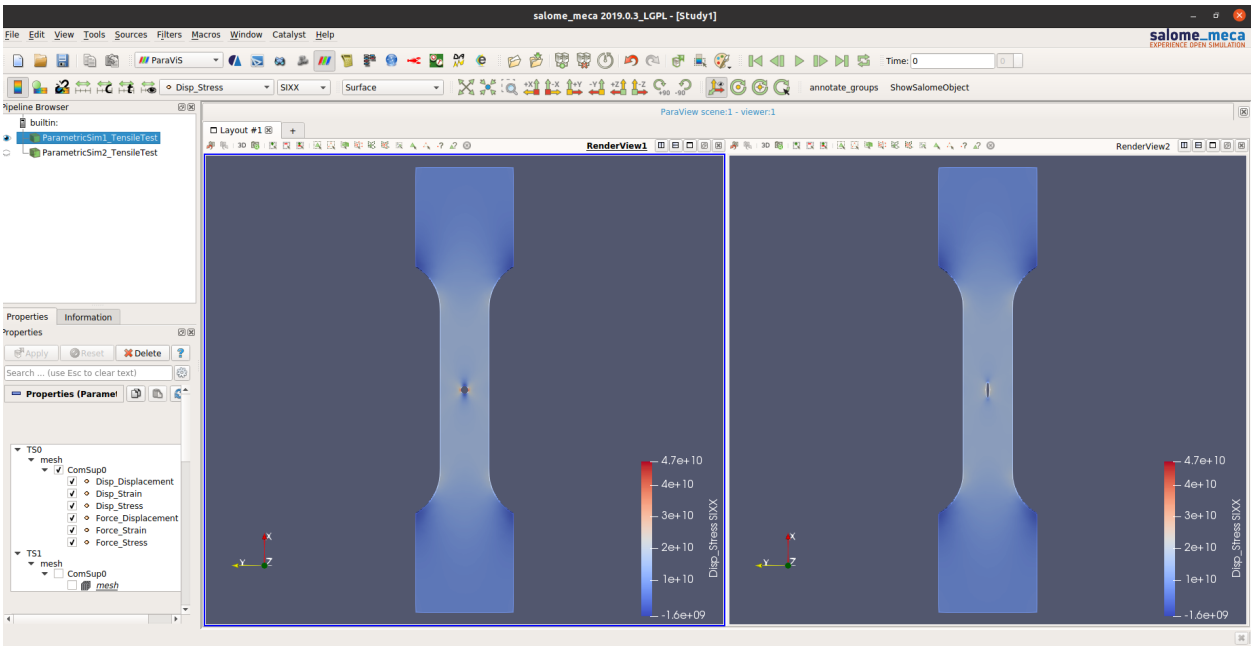


Fig. 6.5: **ParaVis** visualisation of parametric analysis of sample where the dimensions of the void in its centre were varied.

Compare `Notch2.py` and `Notch3.py` in the *Meshes* directory. You should see that only the values for *Rad_a* and *Rad_b* differ. Similarly, only `Mesh` will be different between `ParametricSim1/Parameters.py` and `ParametricSim2/Parameters.py` in the project directory.

> **Warning:** The number of entries for attributes of `Mesh` and `Sim` must be consistent.
>
> For example, if *Mesh.Name* has 3 entries then every attribute of `Mesh` in *Parameters_Var* must also have 3 entries.

6.1.6 Task 3: Running Multiple Simulations Concurrently

The last task introduced you to running multiple simulations, however both the meshing and simulations were run sequentially. For more complex meshes and simulations this would be very time consuming. **VirtualLab** has the capability of running meshes and simulations concurrently, enabling a substantial speed up when running multiple simulations.

In VirtualLab.Settings you will see the `kwarg` *NbJobs* which specify how many tasks VirtualLab is to run concurrently.

Note: The number you specify for *NbJobs* will depend on a number of factors, including the number of CPUs available and the RAM.

For example, the fineness of the mesh is an important consideration since this can require a substantial amount of RAM.

Action

In the *RunFile* change *NbJobs* to 2:

```
VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbJobs=2
           )
```

Launch **VirtualLab**.

You should now see that 'Notch2' and 'Notch3' are created simultaneously, followed by one *xterm* window opening, with the *Name* of each simulation written on top left. You can switch between simulations and compare them. Additionally, it is possible to open two simulations side by side.

6.1.7 Task 4: Simulation Without Meshing

After running the simulation, you realise that the wrong material was used - you wanted to run analysis on a tungsten sample. You are happy with the meshes you already have and only want to re-run the simulations.

This can be accomplished by using the *RunMesh* `kwarg` in VirtualLab.Parameters. By setting this flag to `False` **VirtualLab** will skip the meshing routine.

Action

Change the material in *Parameters_Master* to 'Tungsten':

```
Sim.Materials = 'Tungsten'
```

Change the name of the simulations in *Parameters_Var* also:

```
Sim.Name = ['ParametricSim1_Tungsten', 'ParametricSim2_Tungsten']
```

In the *RunFile* ensure that *RunMesh* is set to False:

```
VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=False,
           RunSim=True,
           RunDA=True
           )
```

Launch **VirtualLab**.

You should notice the difference in stress and displacement for the tungsten sample compared with that of the copper sample.

Tip: If you have interest in developing your own scripts then it would be worthwhile looking at the scripts `DogBone.py` and `Tensile.comm` which have been used by **SALOME** and **Code_Aster** respectively for this analysis.

6.2 Thermal

6.2.1 Introduction

The Laser flash analysis (LFA) experiment consists of a disc shaped sample exposed to a short laser pulse incident on one surface. During the pulse, and for a set time afterwards, the temperature change is tracked with respect to time on the opposing surface. This is used to measure thermal diffusivity, which is consequently used to calculate thermal conductivity.

This example introduces some of the post-processing capabilities available in **VirtualLab**. The results of the simulation will be used to calculate the thermal conductivity of the material, while images of the heated sample will be produced using **ParaVis**.

Action

Because this is a different simulation type, *Simulation* will need to be changed:

```
#=============================================================================
# Definitions
#=============================================================================
Simulation='LFA'
Project='Tutorials'
Parameters_Master='TrainingParameters'
Parameters_Var='Parametric_1'

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbJobs=2
           )

VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=True,
           RunSim=True,
           RunDA=True
           )

#=============================================================================
# Methods
#=============================================================================

VirtualLab.Mesh(
           ShowMesh=False,
           MeshCheck=None
           )
```

(continues on next page)

(continued from previous page)

```
VirtualLab.Sim(
            RunPreAster=True,
            RunAster=True,
            RunPostAster=True,
            ShowRes=True
            )

VirtualLab.DA()
```

In the *Parameters_Master* file `Inputs/LFA/Tutorials/TrainingParameters.py` you will again find namespace `Mesh` and `Sim` along with *DA*.

6.2.2 Sample

The file used by **SALOME** to create the geometry and generate the mesh is `Scripts/Experiments/LFA/Mesh/Disc.py`. The attributes required to create the sample geometry, referenced in Fig. 6.6, are:

```
Mesh.Radius = 0.0063
Mesh.HeightB = 0.00125
Mesh.HeightT = 0.00125
Mesh.VoidCentre = (0,0)
Mesh.VoidRadius = 0.000
Mesh.VoidHeight = 0.0000
```

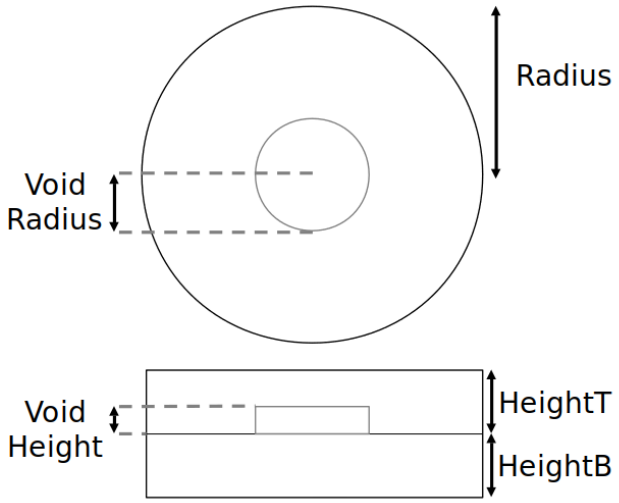


Fig. 6.6: Drawing of the disc shaped sample with the attirubtes of `Mesh` used to specify the dimensions.

The centre of the void is offset from the centre of the disc by *VoidCentre*. Entering a negative number for *VoidHeight* will create a void in the bottom half of the disc as apposed to the top half.

The attributes used for the mesh refinement are similar to those used in the Tutorial #1 tutorial:

```
Mesh.Length1D = 0.0003
Mesh.Length2D = 0.0003
Mesh.Length3D = 0.0003
Mesh.VoidSegmentN = 30
```

6.2.3 Simulation

Because this is a transient (time-dependant) simulation, additional information is required by **Code_Aster**, such as the initial conditions (IC) of the sample and the temporal discretisation.

The time-stepping is defined using the attribute *dt*. This is a list of TUPLES (A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.), where the first entry specifies the timestep size, the second the number of time steps and the third is the frequency of how often the results are stored (optional, default is 1). Further 'time sections' may be defined by additional entries in the list.

For example:

```
Sim.dt = [(0.1,5,1),(0.2,10,2)]
```

Would result in:

```
# Time steps
0,0.1,0.2,0.3,0.4,0.5,0.7,0.9,1.1,1.3,1.5,1.7,1.9,2.1,2.3,2.5
# Results stored at
0,0.1,0.2,0.3,0.4,0.5,0.9,1.3,1.7,2.1,2.5
```

The total number of timesteps, N_tsteps, is the sum of the second entry in each time section:

$$N\_tsteps = N\_tsteps_1 + ... + N\_tsteps_m$$

The end time of the simulation, T, is the sum of the product of timestep size and number of timesteps for each time section:

$$T = t_0 + dt_1 \times N\_tstep_1 + ... + dt_m \times N\_tstep_m$$

The number of timestep results stored, N_Res, is the sum of the number of timesteps divided by the storage frequency for each time section plus one for the initial conditions at t_0:

$$N\_Res = 1 + \frac{N\_tstep_1}{freq_1} + ... + \frac{N\_tstep_m}{freq_m}$$

The attribute *Theta* dictates whether the numerical scheme is fully explicit (0), fully implicit (1) or semi-implicit (between 0 and 1).

For this simulation the temporal discretisation is:

```
Sim.dt = [(0.00002,50,1), (0.0005,100,2)]
Sim.Theta = 0.5
```

When *Theta* is 0.5 the solution is inherently stable and is known as the Crank-Nicolson method.

For this virtual experiment, the time-step size has been set to be smaller initially to capture the larger gradients present

during the laser pulse at the start of the simulation.

$$N\_tsteps = 50 + 100 = 150$$

$$T = 0.00002 \times 50 + 0.0005 \times 100 = 0.051$$

$$N\_Res = 1 + \frac{50}{1} + \frac{100}{2} = 101$$

The sample is set to initially have a uniform temperature profile of 20° C.

`Sim` also has attributes relating to the power and profile of the laser pulse.

```
Sim.Energy = 5.32468714
Sim.LaserT= 'Trim'
Sim.LaserS = 'Gauss'
```

Energy dictates the energy (J) that the laser will provide to the sample. The temporal profile of the laser is defined by *LaserT*, where the different profiles can be found in `Scripts/Experiments/LFA/Laser`, see Fig. 6.7. 'Coarse', 'Fine' and 'Trim' are versions of experimentally measured data whereas 'Hat' and 'HatMid' are idealised profiles. The spatial profile, *LaserS*, can be either 'Uniform' or 'Gaussian', see Fig. 6.8.

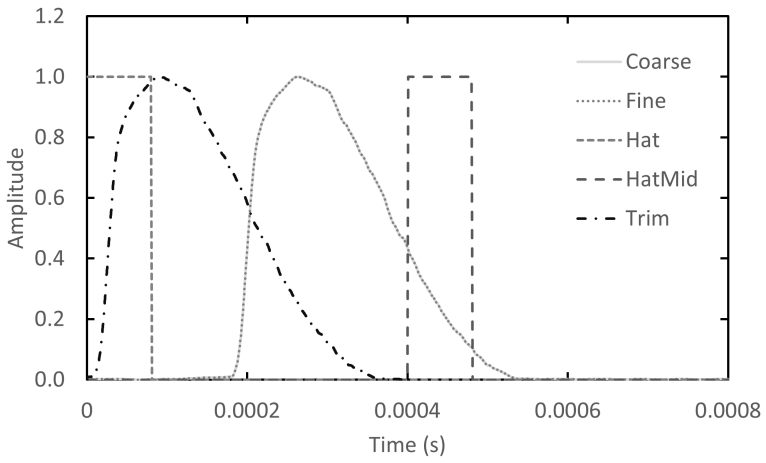


Fig. 6.7: Plot of various laser temporal profiles available.

A convective boundary condition (BC) is also applied by defining the heat transfer coefficient (HTC) and the external temperature:

```
Sim.ExtTemp = 20
Sim.BottomHTC = 0
Sim.TopHTC = 0
```

The attribute *Sim.Materials* in this example is a python dictionary whose `keys` are the names of the mesh groups and their corresponding `values` are the material properties which will be applied to those groups:

```
Sim.Materials = {'Top':'Copper', 'Bottom':'Copper'}
```

This allows different material properties to be applied to different parts of the sample in **Code_Aster**.

As previously mentioned, this tutorial introduces post-processing in **VirtualLab**.

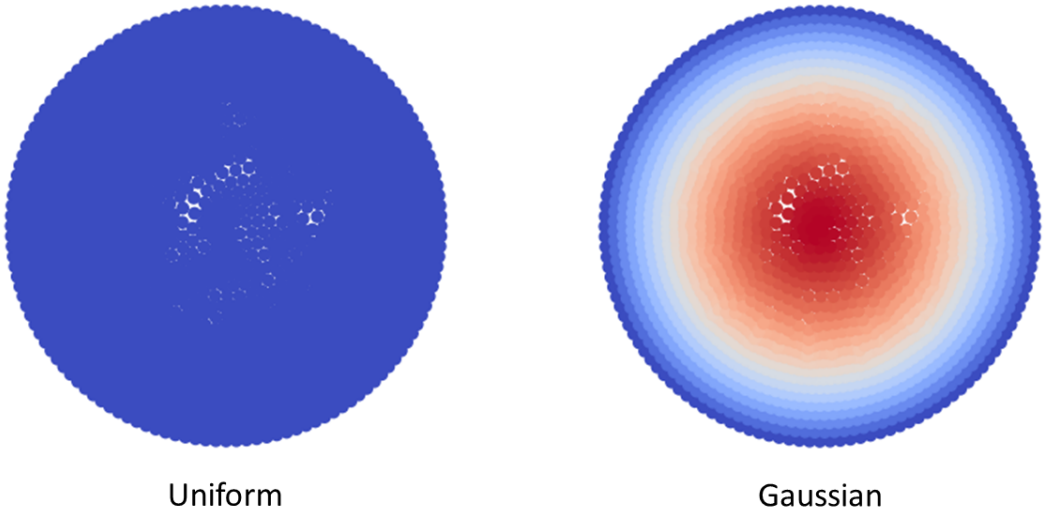


Fig. 6.8: Plot of laser spatial profiles available.

```
Sim.PostAsterFile = 'ConductivityCalc'
Sim.Rvalues = [0.1, 0.5]
```

The script `Scripts/Experiments/LFA/Sim/ConductivityCalc.py` is used to create plots of the temperature distribution over time and calculate the thermal conductivity from the simulation data.

The variables associated with *DA* will be discussed shortly.

6.2.4 Task 1: Checking Mesh Quality

Open the *Parameters_Var* file `Input/LFA/Tutorials/Parametric_1.py` in a text editor. The parameters used here will create two meshes, one with a void and one without, for use in three simulations.

In the first simulation, a Gaussian laser profile is applied to the disc without a void. The second and third simulation apply a Gaussian and uniform laser profile, respectively, to the disc now containing a void.

Suppose you are interested in seeing the meshes prior to running the simulation. To do this, the `kwarg` *ShowMesh* is used in VirtualLab.Mesh. Setting this to `True` will open all the generated meshes in the **SALOME** GUI to visualise and assess their suitability.

Action

In the *RunFile* change the `kwargs` *ShowMesh* to `True`:

```
VirtualLab.Mesh(
            ShowMesh=True,
            MeshCheck=None
            )
```

NbJobs should still be set to 2 from the Tensile tutorial.

Launch **VirtualLab**:

```
VirtualLab -f RunFiles/RunTutorials.py
```

You will notice that each mesh has the group 'Top' and 'Bottom' in *Groups of Volumes* in the object browser (usually located on the left-hand side). These groups are the `keys` defined in *Sim.Materials*.

Once you have finished viewing the meshes you will need to close the **SALOME** GUI. Since this `kwarg` is designed to check mesh suitability, the script will terminate once the GUI is closed, meaning that no simulations will be run.

6.2.5 Task 2: Transient simulation

You decide that you are happy with the quality of the meshes created for your simulation.

You will notice in the *Parameters_Var* file `Input/LFA/Tutorials/Parametric_1.py` that *Sim.Name* are:

```
Sim.Name = ['Linear/SimNoVoid','Linear/SimVoid1','Linear/SimVoid2']
```

VirtualLab supports writing names in this manner so that simulations can be grouped together in sub-directories. This is designed to give the user flexibility in how results are stored.

Action

In the *RunFile* change *ShowMesh* back to its default value `False` and set *RunMesh* to `False` to ensure that the simulations are run without re-meshing. Also set *RunDA* to `False` for the time being.

Since 3 simulations are to be run you can set *NbJobs* to 3 (if you have the resources available):

```
VirtualLab.Settings(
            Mode='Interactive',
            Launcher='Process',
            NbJobs=3
            )

VirtualLab.Parameters(
            Parameters_Master,
            Parameters_Var,
            RunMesh=False,
            RunSim=True,
            RunDA=False
            )

VirtualLab.Mesh(
            ShowMesh=False,
            MeshCheck=None
            )
```

You will notice a sub-directory named 'Linear' has been created in the project directory which contains the 3 simulations which ran. See Fig. 6.9 for an example visualisation of the results.

In the *Aster* directory for each of the 3 simulations, you will find: `AsterLog`; `Export`; and **Code_Aster** `.rmed` files, as seen in the first tutorial. You will also find the file `TimeSteps.dat` which lists the timesteps used in the simulation.

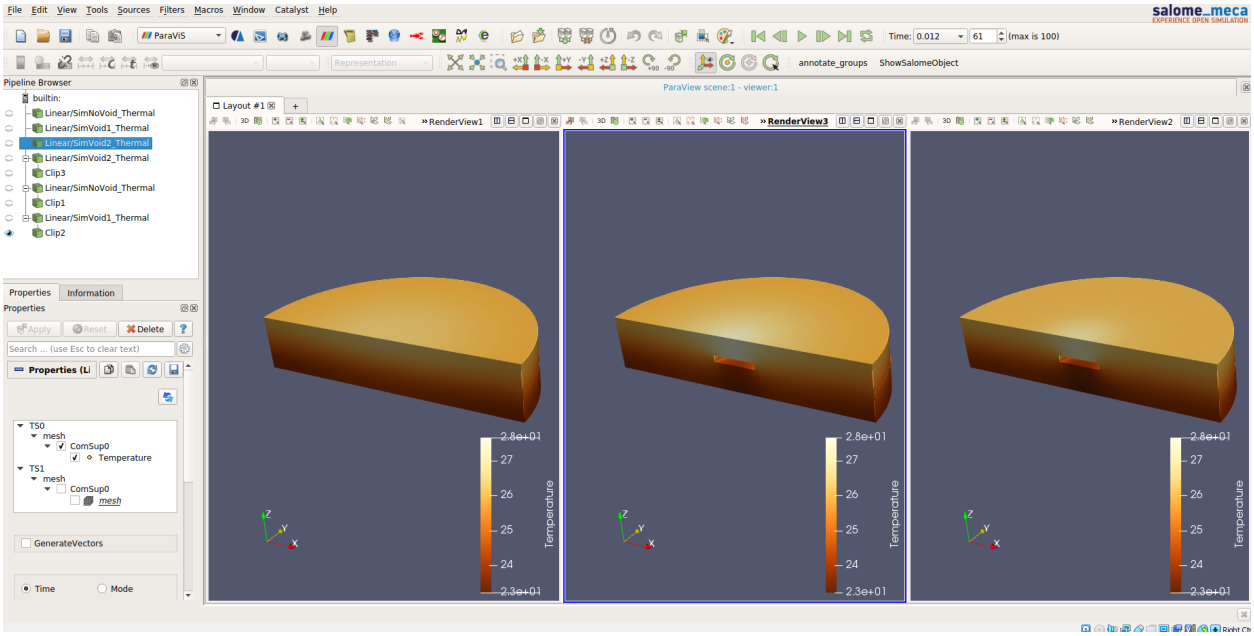


Fig. 6.9: Visualisation of three instances of the LFA simulation results, showing a cross-sectional view for a comparison of the internal temperature profile.

In the *PostAster* directory for each simulation you will find the following files generated by `ConductivityCalc.py`:

- `LaserProfile.png`
- `AvgTempBase.png`
- `Summary.txt`

`LaserProfile.png` shows the temporal laser profiles (top) along with the spatial laser profile (bottom) used in the simulation. The temporal profile shows the flux (left) and the subsequent loads applied to each node (right).

`AvgTempBase.png` shows the average temperature on the base of the sample over time. If values have been specified in *Sim.Rvalues* then this plot will also contain the average temperature on differently sized areas of the bottom surface. An R value of 0.5 takes the average temperatures of nodes within a half radius of the centre point on the bottom surface. An R value of 1 would be the entire bottom surface.

The curves for an Rvalue of 0.1 show the rise in average temperature with respect to time over the central most area of the disc's bottom surface. It can be seen that this temperature rises more rapidly for the 'SimNoVoid' simulation compared with the 'SimVoid1' and 'SimVoid2' simulations. This is due to the void creating a thermal barrier in the centre-line of the sample, i.e., directly between the thermal load and the area where the average temperature is being measured. Differences can also be observed between the profiles for the 'SimVoid1' and 'SimVoid2' simulations despite the geometries being identical, which is due to the different spatial profile of the laser. These images are created using the python package matplotlib.

`Summary.txt` contains the calculated thermal conductivity, along with the accuracy of the spatial (mesh fineness) and temporal discretisation (timestep sizes) used by the simulation.

6.2.6 Task 3: Re-running Sub-sets of Simulations

You realise that you wanted to run the 'SimNoVoid' simulation with a uniform laser profile, rather than the Gaussian profile you used. Running particular sub-sets of simulations from *Parameters_Var* can be achieved by including *Sim.Run* in the file. This list of Booleans will specify which simulations are run. For example:

```
Sim.Run=[True,False,True,False]
```

included within a *Parameters_Var* file would signal that only the first and third simulation need to be run.

Since 'SimNoVoid' is the first entry in *Sim.Name* in `Parametric_1.py` the corresponding entry in *Sim.Run* will need to be `True` with the remaining entries set to `False`.

Action

In the *Aster* section of `Parametric_1.py` add *Sim.Run* with the values shown below and change the first entry in *Sim.LaserS* to 'Uniform':

```
Sim.Run = [True,False,False]
Sim.LaserS = ['Uniform','Gauss','Uniform']
```

There is no need to change the value for *NbJobs*.

Launch **VirtualLab**.

Note: *Sim.Run* is optional and does not need to be included in the *Parameters_Master* file.

You should see only the simulation 'SimNoVoid' running. From the temperature results displayed in **ParaVis** it should be clear that a uniform laser profile is used.

Tip: Similarly, certain meshes from *Parameters_Var* can be chosen by including *Mesh.Run* in the file in the same manner as *Sim.Run* was added above. For example, adding:

```
Mesh.Run = [True,False]
```

to `Parametric_1.py` and re-running the mesh would result only in 'NoVoid' being re-meshed since this is the first entry in *Mesh.Name*.

6.2.7 Task 4: Collective Post-Processing

We would like to create images of the simulation we have run using **ParaVis**. Given that we want to compare the 3 simulations it is essential that all are plotted using the same temperature range for the colour bar.

Up until now the post-processing carried out has been for each simulation individually as we've used *PostAsterFile* within `Sim`, however sometimes we will need access to multiple sets of results simultaneously, e.g., for comparison.

This is possible using the VirtualLab.DA Method. This is primarily used to analyse data collected from simulations, where machine learning could be used to gain insight, for example.

In the *Parameters_Master* file `TrainingParameters.py` you will see the Namespace *DA* with the following attributes.

```
DA.Name = 'Linear'
DA.File = 'Images'
DA.CaptureTime = 0.01
# DA.PVGUI = True
```

The data analysis will be performed on the results in the directory specified by *DA.Name*. The file `Scripts/Experiments/LFA/DA/Images.py` captures images of the simulations at time *CaptureTime*.

Warning: Due to issues with the **ParaVis** module incorporated in **SALOME**, off-screen rendering is not possible with the use of VMs. The commented attribute *PVGUI* forces **ParaVis** to run the script in the GUI where the rendering works fine. If you're using a VM, uncomment this line by deleting the hash character, i.e., *#*.

However, both off-screen and GUI rendering will currently fail for systems without a screen, e.g., HPC clusters. We hope to apply a fix for this in future.

Action

Given that all the simulations are now correct there is no need to re-run them. In the *RunFile* set the `kwarg` *RunSim* to `False` and change *RunDA* to `True`:

```
VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=False,
           RunSim=False,
           RunDA=True
           )
```

You will need to manually close the GUI once the imaging is complete.

Launch **VirtualLab**.

Note: Creating images using **ParaVis** will produce 'Generic Warning' messages in the terminal. They are caused by bugs within **SALOME** and can be ignored.

You should now see the following images added to the `PostAster` directory for each simulation:

- `Capture.png`
- `ClipCapture.png`
- `Mesh.png`
- `MeshCrossSection.png`

The images `Mesh.png` and `MeshCrossSection.png` show the mesh used in the simulation and its cross-section, respectively. The images `Capture.png` and `ClipCapture.png` show the heat distribution in the sample at the time specified by the *CaptureTime*. The colour bar range used in these image uses the min and max temperature over all the simulations for consistency, see Fig. 6.10.

Note: If errors have occurred when creating images in **ParaVis** uncomment *DA.PVGUI* in `TrainingParameters.py` as advised in the *Warning* section for VMs above.

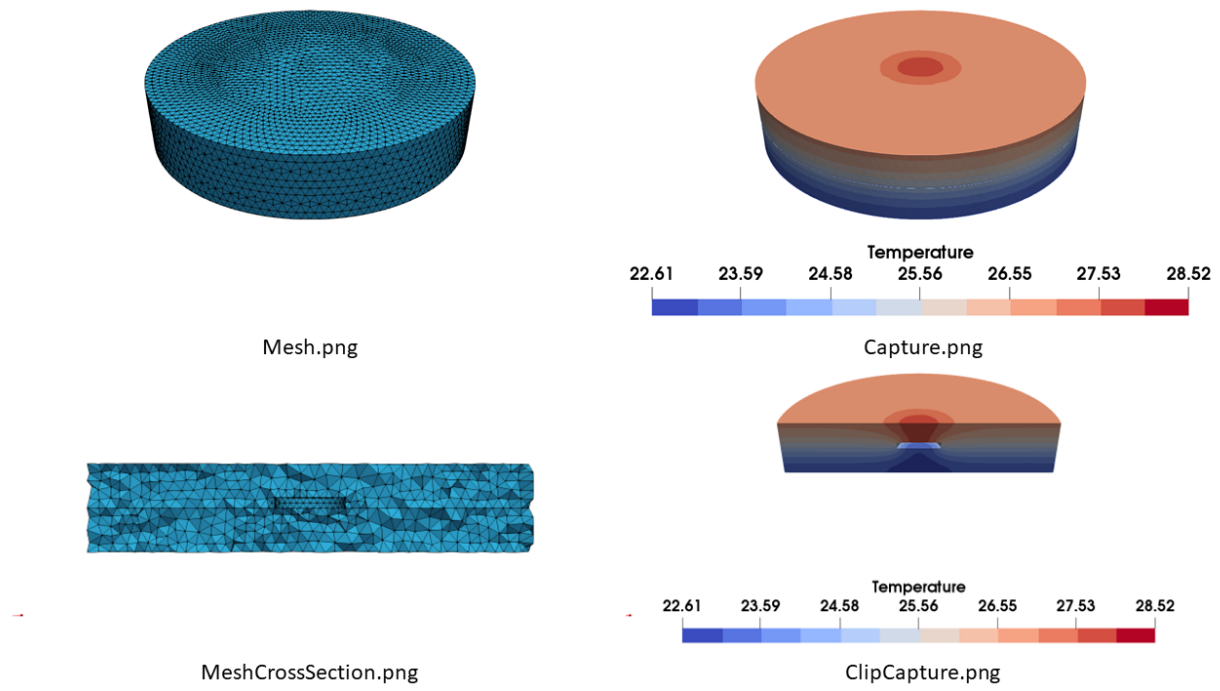


Fig. 6.10: Images generated for the LFA virtual experiment as part of the DA method.

Also feel free to uncomment this attribute if you are interested in seeing how **ParaVis** is used to generate images.

6.2.8 Task 5: Non-linear Simulations

Thus far, the script used by **Code_Aster** for the Laser Flash Analysis has been `Disc_Lin.comm`, which is a linear simulation. The command script `Disc_NonLin.comm` allows the use of non-linear, temperature dependent, material properties in the simulation.

The collection of available materials can be found in the Materials directory. Names of the non-linear types contain the suffix '_NL'.

Action

In the *RunFile RunSim* will need to be changed back to `True`. As we will create images of the simulations you can change *ShowRes* to `False`

```
VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=False,
           RunSim=True,
           RunDA=True)

VirtualLab.Sim(
           RunPreAster=True,
           RunAster=True,
```

(continues on next page)

(continued from previous page)

```
            RunPostAster=True,
            ShowRes=False)
```

We want to save the results of the nonlinear simulations separately. In `Parameteric_1.py` change the simulation names in *Sim.Names*:

```
Sim.Name = ['Nonlinear/SimNoVoid','Nonlinear/SimVoid1','Nonlinear/SimVoid2']
```

In `TrainingParameters.py` change *Sim.AsterFile* to 'Disc_NonLin' and modify *Sim.Materials* to use non-linear materials:

```
Sim.AsterFile = 'Disc_NonLin'
Sim.Materials = {'Top':'Copper_NL', 'Bottom':'Copper_NL'}
```

As the results will now be stored in a sub-directory named 'Nonlinear' you will need to change *DA.Name* to reflect this:

```
DA.Name = 'Nonlinear'
```

Launch **VirtualLab**.

Note: Linear material properties can also be used in `Disc_NonLin.py`

Notice that the **Code_Aster** terminal output is different in the non-linear simulation compared with the linear one. This is due to the Newton iterations which are required to find the solution in non-linear simulations.

The default maximum number of Newton iterations is 10. This can be altered by adding the attribute *MaxIter* to the `Sim` namespace.

Tip: If you are interested in developing post-processing scripts look at `Sim/ConductivityCalc.py` and `DA/Images.py`.

6.3 Thermo-mechanical

6.3.1 Introduction

Heat by Induction to Verify Extremes (HIVE) is an experimental facility at the UK Atomic Energy Authority (UKAEA) to expose plasma-facing components to the high thermal loads that they will experience in a fusion reactor. Samples are thermally loaded by induction heating whilst being actively cooled with pressurised water.

While **Code_Aster** has no in-built ElectroMagnetic coupling, having a python interpreter and being open source makes it easier to couple with external solvers and software compared with proprietary commercial FE codes.

In **VirtualLab**, the heating generated by the induction coil is calculated by using the open-source EM solver **ERMES** during the pre-processing stage. The results are piped to **Code_Aster** to be applied as boundary conditions (BC).

The effect of the coolant is modelled as a 1D problem using its temperature, pressure and velocity along with knowing the geometry of the pipe. This version of the code is based on an implementation by Simon McIntosh (UKAEA) of Theron D. Marshall's (CEA) Film-2000 software to model the Nukiyama curve[1] for water-cooled fusion divertor

[1] T. D. Marshall, D. L. Youchison and L. C. Cadwallader. Modeling the nukiyama curve for water-cooled fusion divertor channels. *Fusion Technol.*, 35:8–16, 2001. http://film2000.free.fr/TOFE.pdf.

channels, which itself was further developed by David Hancock (also UKAEA). The output from this model is also piped to **Code_Aster** to apply as a BC.

Because this is a multi-physics problem, the setup of `Sim` is slightly different with additional keywords RunCoolant and RunERMES:

Action

For this tutorial the *RunFile* should have the values:

```
Simulation='HIVE'
Project='Tutorials'
Parameters_Master='TrainingParameters'
Parameters_Var=None

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbJobs=1
           )

VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=True,
           RunSim=True,
           RunDA=True
           )

VirtualLab.Mesh(
           ShowMesh=False,
           MeshCheck=None
           )

VirtualLab.Sim(
           RunPreAster=True,
           RunCoolant=True,
           RunERMES=True,
           RunAster=True,
           RunPostAster=True,
           ShowRes=True
           )

VirtualLab.DA()
```

6.3.2 Sample

The sample selected to use in this tutorial is an additive manufactured sample which was part of the EU FP7 project "Additive Manufacturing Aiming Towards Zero Waste & Efficient Production of High-Tech Metal Products" (AMAZE, grant agreement No. 313781). The sample is a copper block on a copper pipe with a tungsten tile on the top.

The file used to generate the mesh is `Scripts/Experiments/HIVE/Mesh/Monoblock.py`. The geometrical parameters, referenced in Fig. 6.11, are:

```
Mesh.BlockWidth = 0.03
Mesh.BlockLength = 0.05
Mesh.BlockHeight = 0.02
Mesh.PipeCentre = [0,0]
Mesh.PipeDiam = 0.01
Mesh.PipeThick = 0.001
Mesh.PipeLength = Mesh.BlockLength
Mesh.TileCentre = [0,0]
Mesh.TileWidth = Mesh.BlockWidth
Mesh.TileLength = 0.03
Mesh.TileHeight = 0.005

Mesh.Fillet = 0.0005
```

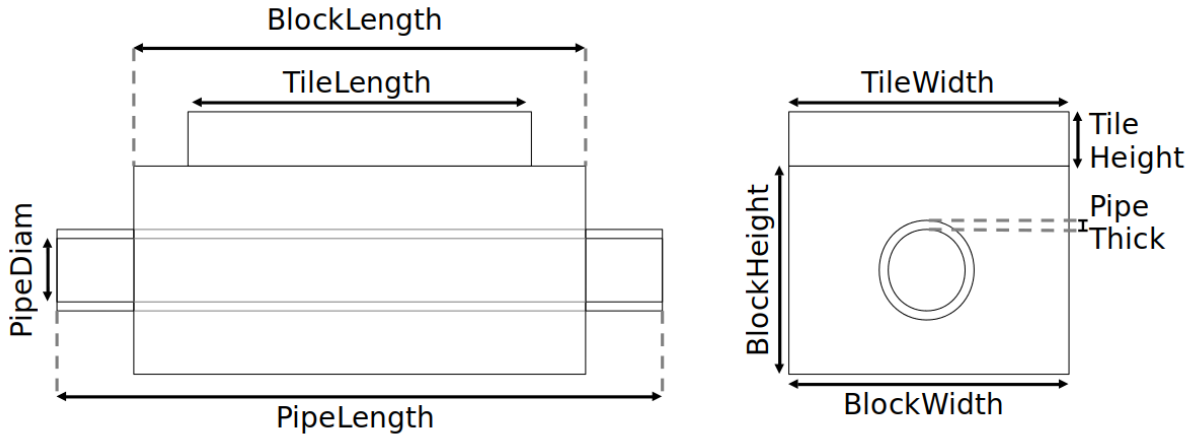


Fig. 6.11: Drawing of the AMAZE sample with the attributes of `Mesh` used to specify the dimensions.

The centre of the pipe is offset from the centre of the co-planar block face by *PipeCentre*. Similarly, the centre of the tile is offset from the centre of the block face by *TileCentre*. The current implementation of **ERMES** leads to singularities at sharp corners. To overcome this the edges belonging to the face adjacent to the induction coil are filleted (or smoothed). The size of the fillet is measured in metres and is given by *Fillet*.

```
# Mesh parameters
Mesh.Length1D = 0.005
Mesh.Length2D = 0.005
Mesh.Length3D = 0.005

Mesh.PipeSegmentN = 20
Mesh.SubTile = [0.002, 0.002, 0.002]
Mesh.Deflection = 0.01
```

The attributes *Length1D-3D* again specify the global mesh sizes. The mesh on the pipe is refined using *PipeSegmentN*, while *SubTile* specifies the mesh size on the tile. This is the part of the component which the coil interacts with, therefore the mesh needs to be finer here. *Deflection* refers to the mesh refinement along the fillet.

6.3.3 Simulation

The coolant is accounted for through the script `Scripts/Experiments/HIVE/Sim/Coolant_1D.py`. This calculates the heat flux between the pipe and the coolant dependent on the temperature on the wall of the pipe. This is usually referred to as the boiling curve.

```
Sim.Pipe = {'Type':'smooth tube', 'Diameter':0.01, 'Length':0.05}
Sim.Coolant = {'Temperature':30, 'Pressure':1, 'Velocity':10}
```

The dictionary *Pipe* specifies information about the geometry of the pipe, while *Coolant* provides properties about the fluid in the pipe.

To calculate the thermal loading arising from the induction coil the file `Scripts/Experiments/HIVE/Sim/EM_Analysis.py` is used which performs the necessary **ERMES** analysis:

```
Sim.CoilType = 'Test'
Sim.CoilDisplacement = [0,0,0.0015]

Sim.Frequency = 1e4
```

ERMES requires a mesh of the induction coil and surrounding vacuum which must conform with the mesh of the component.

The attribute *CoilType* specifies the coil design to be used. Currently available options are:

- 'Test'
- 'HIVE'
- 'Pancake'

CoilDisplacement dictates the x, y and z components of the displacement of the coil with respect to the sample. The z-component indicates the gap between the upper surface of the sample and the coil and must be positive. The x and y components indicate the coil's offset about the centre of the sample, see Fig. 6.12.

Frequency is used by **ERMES** to produce a range of EM results, such as the Electric field (E), the Current density (J) and Joule heating. These results are stored in the sub-directory *PreAster* within the simulation directory.

The Joule heating profile is used by **Code_Aster** to apply the thermal loads. A mesh group is required for each individual volumetric element within the mesh to apply the heat source, however doing so substantially increases the computation time.

To speed this step up the Joule heating values are clustered into N-number of 'bins'. The 1D k-means algorithm (also known as the Jenks optimisation method) find the N optimal value to group the distribution in to. The Goodness of Fit Value (GFV) describes how well the clustering represents the data, ranging from 0 (worst) to 1 (best).

The attribute *NbClusters* specifies the number of groups to cluster the data in to. In this analysis 100 clusters are used. The attribute *Current* specifies the current in the input terminal of the induction coil. Although this parameter technically relates to the **ERMES** analysis the results scale linearly with this, therefore this value can be altered without having to re-run the entire ERMES analysis.

```
Sim.Current = 1000
Sim.NbClusters = 100
```

Because the loads are not time-dependent this can be treated as a steady state thermal problem, with the command file `Monoblock_Steady.comm` used (Steady State). A transient version of this simulation is also available, `Monoblock_Transient.comm`.

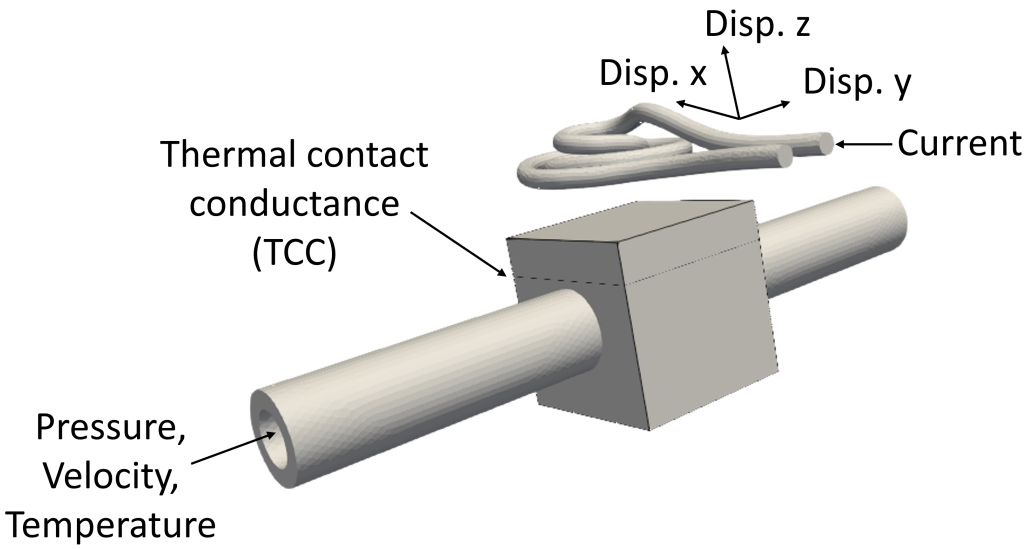


Fig. 6.12: Schematic of the HIVE simulation setup, showing some of the variable parameters.

6.3.4 Task 1: Running 1D Coolant

In this task, firstly, the mesh of the AMAZE sample is created. This will be saved to the meshes directory under the name 'AMAZE'.

Following this, the coolant analysis will be performed. A sub-directory named 'Examples' will have been created in the project directory, inside which a results directory 'Test_Coil' can be found. The boiling curve is shown in `PreAster/HeatTransfer.png`, with important values highlighted. This data is also saved to `PreAster/HeatTransfer.dat` to be passed to CodeAster during the analysis.

Action

Change the *RunERMES* and *RunAster* `kwarg` to `False` in VirtualLab.Sim because we are only interested in the coolant aspect of the work at this stage:

```
VirtualLab.Parameters(
            Parameters_Master,
            Parameters_Var,
            RunMesh=True,
            RunSim=True,
            RunDA=True
            )

VirtualLab.Sim(
            RunPreAster=True,
```

(continues on next page)

(continued from previous page)

```
            RunCoolant=True,
            RunERMES=False,
            RunAster=False,
            RunPostAster=True,
            ShowRes=True
            )
```

Launch **VirtualLab**.

6.3.5 Task 2: Running an ERMES simulation

Action

Change *RunMesh* to `False` in VirtualLab.Parameters because we are using the same mesh. As we don't need to perform the coolant analysis again change *RunCoolant* to `False` in VirtualLab.Sim. Finally, change RunERMES to `True` in VirtualLab.Sim as we want to run the **ERMES** solver:

```
VirtualLab.Parameters(
            Parameters_Master,
            Parameters_Var,
            RunMesh=False,
            RunSim=True,
            RunDA=True
            )

VirtualLab.Sim(
           RunPreAster=True,
           RunCoolant=False,
           RunERMES=True,
           RunAster=False,
           RunPostAster=True,
           ShowRes=True
           )
```

Launch **VirtualLab**.

Information generated by the **ERMES** solver is printed to the terminal. The results generated by **ERMES** are converted to a format compatible with **ParaVis** and saved to `PreAster/ERMES.rmed`. These are the results which are displayed in the GUI, assuming that the `kwarg` *ShowRes* is still set to `True`.

The results from **ERMES** show the whole domain, which includes the volume surrounding the sample and coil, which will obscure the view of them. In order to only visualise the sample and coil, these groups must be extracted. This is accomplished by selecting `Filters / Alphabetical / Extract Group` from the menu, then using the checkboxes in the properties window (usually on the bottom left side) to select `Coil` and `Sample` before clicking `Apply`, see Fig. 6.13.

It should then be possible to visualise any of the following results:

- Joule_heating
- Electric field (E) - real, imaginary and modulus
- Current Density (J) - real, imaginary and modulus

Joule_heating is the field which is used in **Code_Aster**.

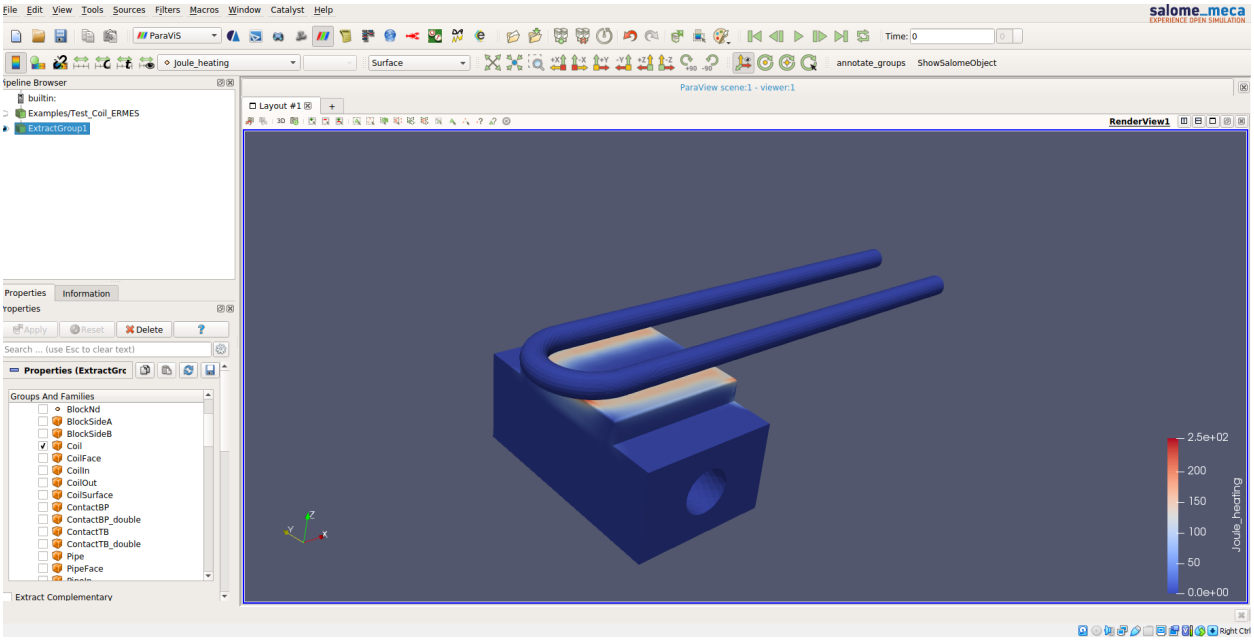


Fig. 6.13: Visualisation of Joule heating results as calculated by **ERMES**.

6.3.6 Task 3: Applying ERMES BC in Code_Aster

Next, a thermal simulation is performed by **Code_Aster** using the results from **ERMES** and the boiling curve. Because we're interested in the results once the sample reaches steady state there is no need to run a transient simulation. An additional benefit is that this will reduce the computation time substantially.

Action

You will also need to change the `kwarg` *RunAster* back to `True` in the *RunFile* to run the simulation. Also change *RunERMES* to `False` because the EM data has already been created.

```
VirtualLab.Sim(
            RunPreAster=True,
            RunCoolant=False,
            RunERMES=False,
            RunAster=True,
            RunPostAster=True,
            ShowRes=True
            )
```

Launch **VirtualLab**.

Both the **ERMES** and **Code_Aster** results are displayed in **ParaVis** with the suffix 'ERMES' and 'Thermal' respectively. By investigating the visualisation of the **Code_Aster** results you will observe that the temperature profile on the sample is very similar to the *Joule_heating* profile generated by **ERMES**.

6.3.7 Task 4: Coil change

The previous analysis will be run again using a different coil. This time an actual coil geometry used for testing in HIVE will be selected. This is referred to as 'HIVE'.

Action

In `TrainingParameters.py` you will need to change *Sim.Name* to 'Examples/HIVE_Coil' and change *CoilType* to 'HIVE':

```
Sim.Name = 'Examples/HIVE_Coil'
Sim.CoilType = 'HIVE'
```

As the entire simulation steps need to be performed again *RunCoolant* and *RunERMES* must be changed back to `True`:

```
VirtualLab.Sim(
            RunPreAster=True,
            RunCoolant=True,
            RunERMES=True,
            RunAster=True,
            RunPostAster=True,
            ShowRes=True
            )
```

Launch **VirtualLab**.

The **ERMES** and **CodeAster** results should both be opened in **ParaVis** to view. You should notice that the peak temperature in the component using this coil is higher than in the previous simulation even though they are positioned in the same location. This is because the winding of the coil creates a more powerful field to induce the heat generation in the component.

Note: You can open the results from the previous analysis alongside this by going to `File/Open ParaView File` and navigating to the directory 'Test_Coil' as shown in Fig. 6.14.

6.3.8 References

6.4 Image-Based Simulation

6.4.1 Introduction

Image-based simulation is a technique which enables more accurate geometries of components to be modelled. Imaging techniques, such as X-ray computed tomography (CT) scanning, enable the characterisation of internal features within a component, from which a more accurate mesh can be generated compared with an idealised CAD-based equivalent. These methods are able to capture features present due to manufacturing methods, such as asymmetry or micro-cracks, yielding simulations with increased accuracy. That is, image-based meshes allow simulations of components 'as manufacutred' rather than 'as designed'.

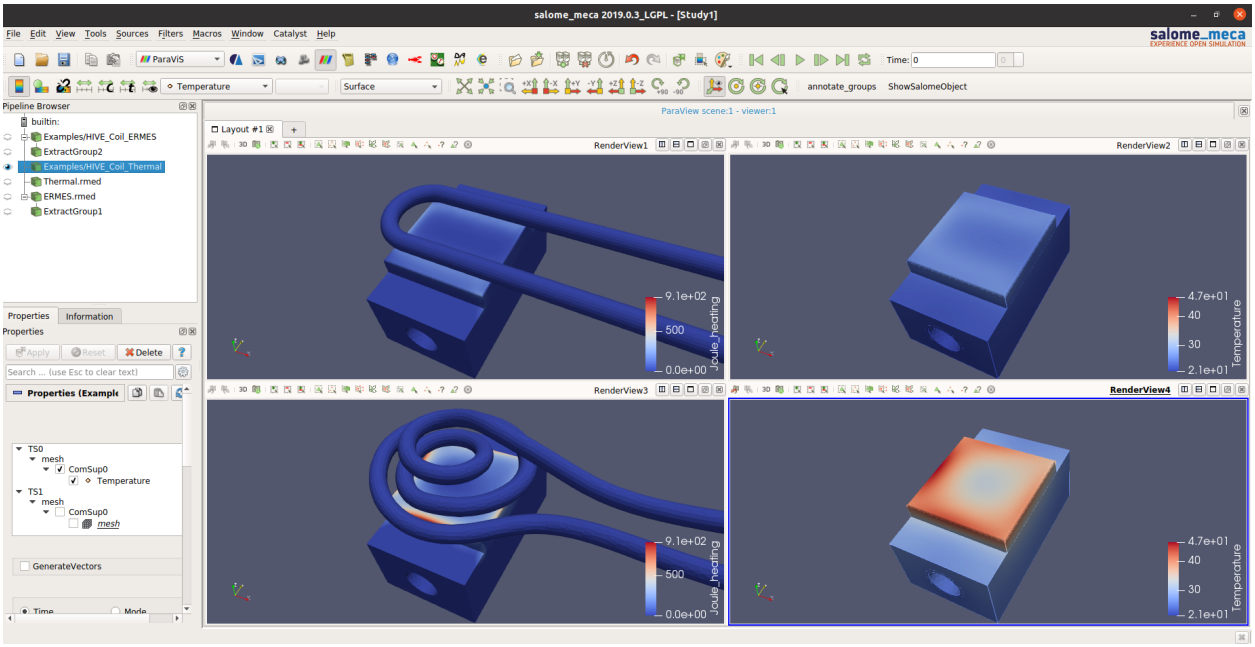


Fig. 6.14: Visualisation of simulation results from **ERMES** and **Code_Aster** for both coil types.

6.4.2 Sample & Simulation

In this example, a CT scan of a dog-bone sample is used in a tensile test. The image-based mesh used for this simulation can be downloaded here. See www.ibsim.co.uk for further information on the image-based simulation methodology.

6.4.3 Task 1

Action

The *RunFile* `RunTutorials.py` should be set up as follows to run this simulation:

```
Simulation='Tensile'
Project='Tutorials'
Parameters_Master='TrainingParameters_IBSim'
Parameters_Var=None

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbJobs=1
           )

VirtualLab.Parameters(
```

(continues on next page)

(continued from previous page)

```
            Parameters_Master,
            Parameters_Var,
            RunMesh=True,
            RunSim=True,
            RunDA=True
            )

VirtualLab.Mesh(
            ShowMesh=False,
            MeshCheck=None
            )

VirtualLab.Sim(
            RunPreAster=True,
            RunAster=True,
            RunPostAster=True,
            ShowRes=True
            )

VirtualLab.DA()
```

Ensure that the downloadedi mage-based mesh has been saved to the following location `Output/Tensile/Tutorials/Meshes/Tensile_IBSim.med`

Launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

Looking at `Input/Tensile/Tutorials/TrainingParameters_IBSim.py` you will notice *Sim* has the variable 'Displacement' but not 'Force', meaning only a controlled displacement simulation will be run.

From the results shown in **ParaViS** you should notice the asymmetric nature of the displacement, stress and strain profiles. These are as a result of the subtle imperfections in the non-idealised Tensile_IBSim mesh compared with an idealised CAD-based mesh.

6.5 Mesh Voxelisation

6.5.1 Introduction

3D polygon meshes are a common and memory efficient way to represent 3D objects for use in CAD software. However there exist other ways to represent objects in 3D space. One common method is known as voxelisation (commonly spelled voxelization). This involves representing the object as a grid of 3D pixels (voxels). The voxels can be assigned different values to represent a certain characteristic, for example a material type or temperature. The voxel grid can be visualised with the values assigned to a colour or greyscale. The voxels can be stored as a numerical array or as a series of 2D images (or a single virtual tiff stack in the case of **CAD2Vox**) with each image representing a slice along a particular axis.

VirtualLab integrates an internally developed python package called CAD2Vox that provides an interface for converting many different types of CAD mesh formats into a voxel-based representation in parallel. The calculations can be done either with threading on multi-core CPUs using OpenMP or on a Nvidia GPU using CUDA.

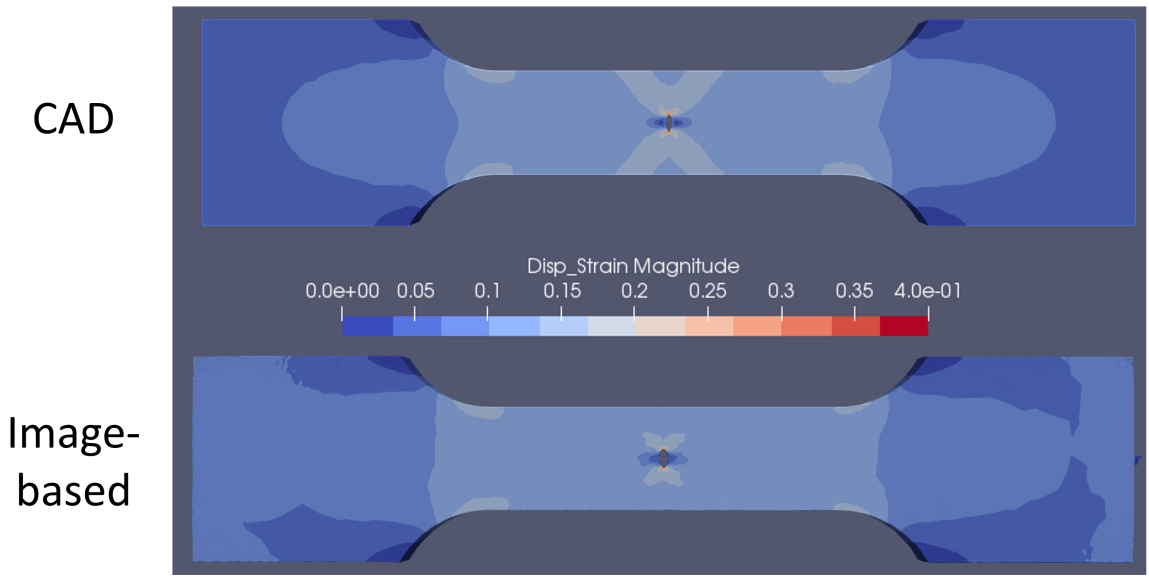


Fig. 6.15: Comparison of CAD-based (idealised) and image-based simulation results.

The **CAD2Vox** integration with **VirtualLab** allows us to run **CAD2Vox** as the final step in some analysis to generate a voxel representation of the CAD mesh that is generated by Salome. However, you can also use it standalone on any of the 30 or so mesh formats supported by meshio.

6.5.2 Prerequisites

The examples provided here are mostly self-contained. However, in order to understand this tutorial, at a minimum you will need to have completed the first tutorial to obtain a grounding in how **VirtualLab** is setup. Also, although not strictly necessary, we also recommend completing the third tutorial because we will be using the **Salome** mesh generated from the HIVE analysis as part of one of the examples. All the other tutorials (that is tutorials 2 and 4) are useful but not required if your only interest is the voxelisation features.

6.5.3 Example 1: Running in an existing analysis workflow

In this first example we will use the same analysis performed in Tutorial 1 using a dog-bone component in a tensile test.

Action

The *RunFile* `RunTutorials.py` should be set up as follows to run this simulation:

```
Simulation='Tensile'
Project='Tutorials'
Parameters_Master='TrainingParameters_Cad2Vox'
Parameters_Var=None

 VirtualLab=VLSetup(
            Simulation,
            Project
            )
```

(continues on next page)

(continued from previous page)

```
VirtualLab.Settings(
            Mode='Interactive',
            Launcher='Process',
            NbJobs=1
            )

VirtualLab.Parameters(
            Parameters_Master,
            Parameters_Var,
            RunMesh=True,
            RunSim=True,
            RunVoxelise=True,
            RunDA=True
            )

VirtualLab.Mesh(
            ShowMesh=False,
            MeshCheck=None
            )

VirtualLab.Sim(
            RunPreAster=True,
            RunAster=True,
            RunPostAster=True,
            ShowRes=True
            )

VirtualLab.DA()
VirtualLab.Voxelise()
```

Launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

The main change to note in the *Runfile* is the call to `VirtualLab.Voxelise()`. This is the function that initiates the voxelisation using the parameters defined in *Parameters_Master*`` and *Parameters_Var*. Additionally, within the Environment section, RunVoxelise is explicitly set to run with `True` in `VirtualLab.Parameters`. This isn't technically necessary because the inclusion of `VirtualLab.Voxelise()` in the methods section means it is `True` by default, but explicitly stating this is good practice.

Looking at the file `Input/Tensile/Tutorials/TrainingParameters_Cad2Vox.py` you will notice that the `Namespaces` `Mesh` and `Sim` are setup the same as in the previous tutorial. That is, they generate the CAD geometry and mesh using `Scripts/Experiments/Tensile/Mesh/DogBone.py` and then run two simulations, first force controlled then displacement controlled. Also, since `DA` is not defined in *Parameters_Master*, no data analysis will take place.

You will also notice the Parameters file has a new Namespace `Vox`. This contains the parameters used to control the voxelisation. The file is setup with some sensible default values.

The only values that are strictly required are:

- `Vox.Name`: Label for the **CAD2Vox** run(s), this can be anything you like. This doubles up as the name of the output file(s), therefore you probably want it to be something descriptive.

- `Vox.mesh`: The name of the mesh(es) you wish to voxelise. These are assumed to be in the simulation Meshes directory, in this case `Output/Tensile/Tutorials/Meshes/`. The code also assumes a file extension of `.med` if none is given.
- `Vox.gridsize`: Number of voxels in each dimension.

We have also included a few optional parameters:

- `Vox.cpu` : The default behaviour is to first check for CUDA and a compatible GPU and if they cannot be found it will fall back to CPU. This flag allows us to skip the check and just use the CPU.
- `Vox.use_tetra`: This tells **CAD2Vox** we are using a mesh with Tetrahedron data instead of the default Triangles.

There are also a number of options we have not used in this file. They are listed here for reference:

- `Vox.solid`: Auto fill interior volume when using triangle (surface) data.
- `Vox.unit_length`: You can use this instead of `Vox.gridsize` to define the length/width/height of a single cubic voxel in Mesh-space. **CAD2Vox** then automatically calculates the number of voxels in each dimension using the min and max of the mesh geometry. Hence you don't specify gridsize when using this option.
- `Vox.greyscale_file`: You can use this option to specify a custom name and path for the .csv file that contains materials and associated greyscale values. If the .csv file does not exist, the code will automatically generate a new file and populate it with values read from the mesh file. If this is not set the code defaults to the filename `greyscale_{Vox.Name}.csv` and assumes it's in the simulation output directory (again automatically generating the .csv file if it does not exist).
- `Vox.Num_Threads`: This sets the Number of OMP Threads to use for voxelisation (only needed for CPU). OpenMP by default automatically detects the number of CPUs on the system and uses the maximum it can. This setting allows you to change the number of threads if desired.
- `Vox.image_format`: This option allows you to select the image format for the final output. If it is omitted (or set to `None`) the output defaults to a tiff virtual stack. However, when this option is set the code outputs each slice in z as a separate image in any format supported by Pillow (see the PILLOW docs for the full list). Simply specify the format you require as a string, e.g., `Vox.image_format="png"`.
- `Vox.Orientation`: This option allows you to quickly change the Orientation of the generated output. The default is "XY" but this can be any of "XY, "YZ" or "XZ".
- `Vox.Output_Resolution` : This option allows you to change the size of the final output. By default a bounding box is drawn tightly around the mesh with the exact size calculated using either unit_length or Gridsize. This option allows you to crop or pad with 0's the final output images by specifying a list of 3 ints as [pix_X,pix_Y,number_of_images].
- `Vox.Nikon_file`: You can use this option to the read the voxel unit length from a Nikon xtect file. This can either be the name of the file (including the file extension), if it is in the Input directory. or the absolute path to the file (again including the file extension).

Advanced tip

All these parameters work in the same manner as with `Mesh` and `Sim`. Whereby using lists in *Parameters_Var* will initiate multiple runs with different parameters.

With this in mind `Vox.Gridsize` accepts the value 0 to skip it when using in conjunction with `Vox.unit_length` and, conversely, `Vox.unit_length` accepts [0.0,0.0,0.0] for the reverse effect (i.e., skip it when using `Vox.Gridsize`).

This is useful if, for example, you wish to run two different cases one after the other. The first with a gridsize of 500 and a second with a unit length of 5.0. In that case you could set `Vox.Gridsize=[[500,500,500],[0,0,0]]` and `Vox.unit_length=[[0.0,0.0,0.0],[5.0,5.0,5.0]]` inside *Parameters_Var* to achieve this.

The output from the voxelisation can now be found under `Output/Tensile/Tutorials/Voxel-Images/Notch1.tiff` this can be viewed with appropriate software such as ImageJ, see Fig. 6.16.

In this folder you will also find the file `greyscale_Notch1.csv`. This file contains in csv format all the materials that were read from the mesh file and the corresponding 8-bit greyscale values used in the output images. We will go into this file in detail with the next example because it's not really interesting in this case due to there only being one material, that is "Copper". Hence, the entire dog-bone is coloured white (that is, the max greyscale value of 255 for an 8-bit dataset). Please note that the first column is the region name as read from the mesh file by **CAD2Vox**, in this case there is only one region. Therfore, you should see `Un-Defined` in the first column. See the *next example* for further details.

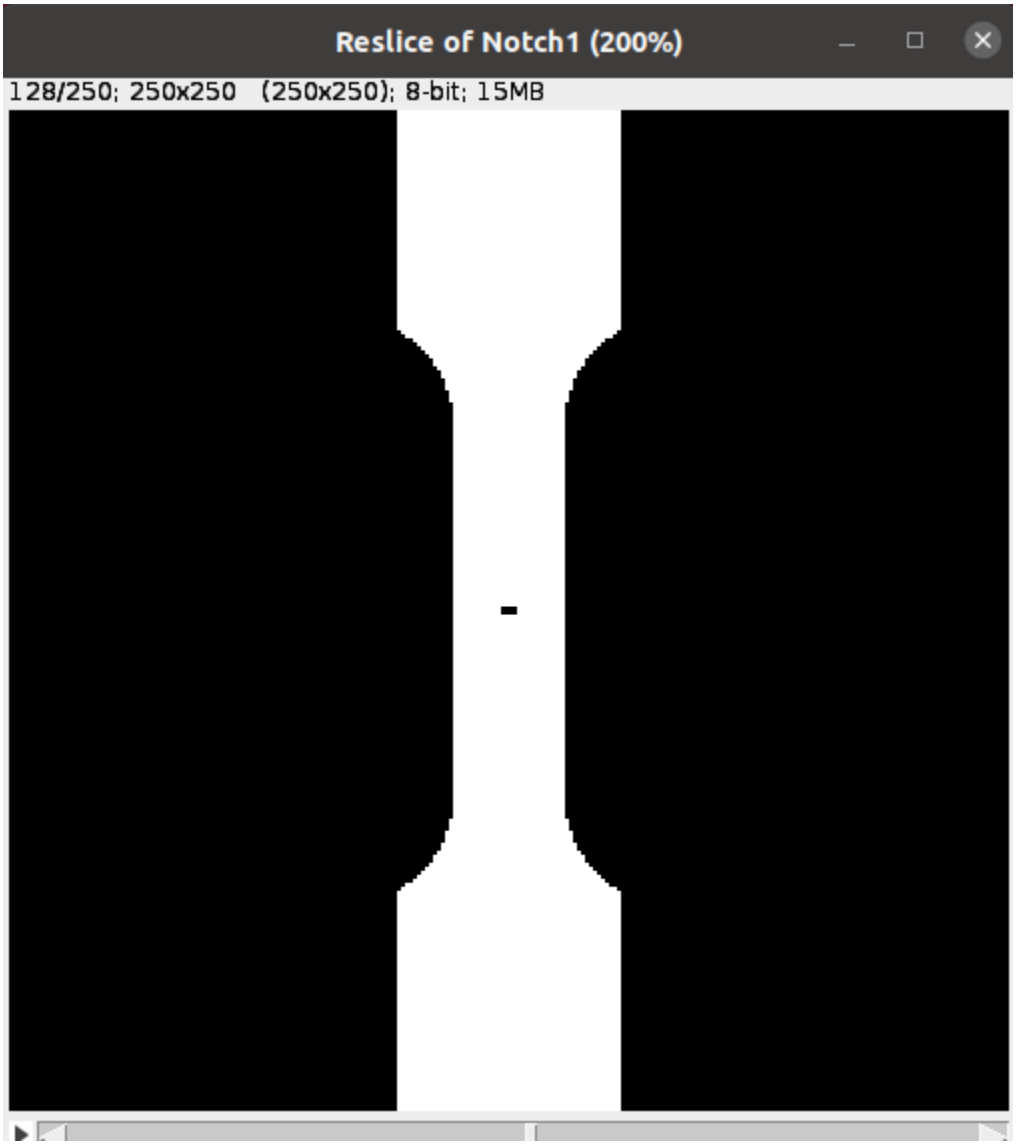


Fig. 6.16: Slice from the tiff stack generated by **CAD2Vox** as visualised by **ImageJ**.

Action

We encourage you to have play around with the various parameters set in the `Vox` Namespace.

Here are some Specific things you could do:

- Try Increasing the value `Vox.Gridsize` from 200 to 500 or even 1000. How does this effect the quality of the image and the run-time?
- Try swapping `Vox.Gridsize` with `Vox.unit_length`. What effect does this have?
- Try changing the format of the output to jpeg using `Vox.image_format`
- If you have CUDA installed and access to a GPU try using it to see how it effects the run time.

6.5.4 Example 2: Running CAD2Vox Standalone

CAD2Vox can be run on an existing mesh file separate from any other analysis within **VirtualLab**. This may be useful if, for example, you have previously performed some long running simulation and now wish to voxelise the CAD mesh without having to needlessly repeat the work in **Code_Aster**.

For this example we will voxelise the AMAZE mesh that was previously generated from the HIVE analysis in tutorial 3. If you have previously completed exercise 3 the mesh should be located in `Output/HIVE/Tutorials/Meshes/AMAZE_Sample.med`. If you have not completed tutorial 3 you can either do so, or you can run the following command:

```
VirtualLab -f RunFiles/Tutorials/Mesh_Voxelisation/Task2_Pre-setup.py
```

This performs the meshing and a bare bones simulation (in non-interactive mode) to generate the necessary output files (which are used as inputs for this tutorial).

Action

The *RunFile* `RunTutorials.py` should be set up as follows to just perform the voxelisation:

```
Simulation='HIVE'
Project='Tutorials'
Parameters_Master='TrainingParameters_Cad2Vox'
Parameters_Var=None

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbThreads=1
           )

VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=False,
           RunSim=False,
           RunDA=False
           )

    VirtualLab.Voxelise()
```

Launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

In this example you can see that we have turned off **Salome** and **Code_Aster** by setting `RunMesh=False`, `RunSim=False`, and `RunDA=False`. Therefore, only the voxelisation method will now take place.

Once again the file `Input/HIVE/Tutorials/TrainingParameters_Cad2Vox.py` is setup with some sensible default values using the `Vox` Namespace. The output from the voxelisation can be found in `Output/HIVE/Tutorials/Voxel-Images/AMAZE_Sample.tiff`.

Unlike the previous example this mesh has 3 regions representing 2 different materials, tungsten and copper (see `Sim.Materials` in `training_parameters.py`. In this case the regions are labelled as: `Block Sample`, `Pipe Sample`, and `Sample Tile`. These have been automatically read in from the mesh by **CAD2Vox** and each region has different greyscale values applied to make them visually distinct from one another.

The greyscale values used for each region can be seen in the file `Output/HIVE/Tutorials/Voxel-Images/greyscale_AMAZE.csv`. This file contains a simple csv table with 3 columns of data separated by commas. First is the region name, as read from the mesh file by **CAD2Vox**, second is the region index assigned by **Salome**, and the third is the greyscale value used in the output.

When first generated the greyscale values are evenly spread from 1 to 255 across all regions found, see Fig. 6.17. These can be changed in this file to whatever values you wish and will be read in on subsequent runs. You can also change the region names if desired. However, we do not recommend changing the region index as this is used internally by **CAD2Vox** to generate the voxel image slices and may produce unexpected results.

Unfortunately, **Salome** does not use the most descriptive names for each region of the mesh (it just uses the keys from `Sim.Materials`). Also, the mesh file may contain objects that **Salome** uses internally (e.g., planes for calculating force/displacment etc.) with no easy way of distinguishing them automatically from the material regions. Therefore, you may need to play around with the greyscale values of a small number of regions to work out what the labels refer to. You can then rename them to something more appropriate and set the greyscale for any region you don't want to see in the final output to 0.

Tip: If you wish to change where the greyscale file is located you can use the previously mentioned parameter `Vox.greyscale_file` to set a custom path, remembering to include the .csv extension. Also if you mess up the file and want to regenerate the greyscale file simply delete the `greyscale_AMAZE.csv` file (or move it to another location) and re-run **CAD2Vox**.

Tip: If you want to use a custom directory to store input meshes for standalone use, you can define `Vox.mesh` to be a string that is the absolute path to the mesh file you wish to use. However, you will need to ensure you include the `.med` file extension. On Windows, absolute paths usually start with `C:\` (although, depending on your exact system, they can be any other drive letter) on MacOS and Linux they always start with `/`.

Action

As mentioned previously, the labels for the mesh regions are not the most useful. Therefore, here are some specific things you could try to rectify this:

- rename each region in the greyscale file to better describe what it represents (e.g. changing "Pipe sample" to "Copper Pipe").
- Set the greyscale values so the regions are distinct from one another.

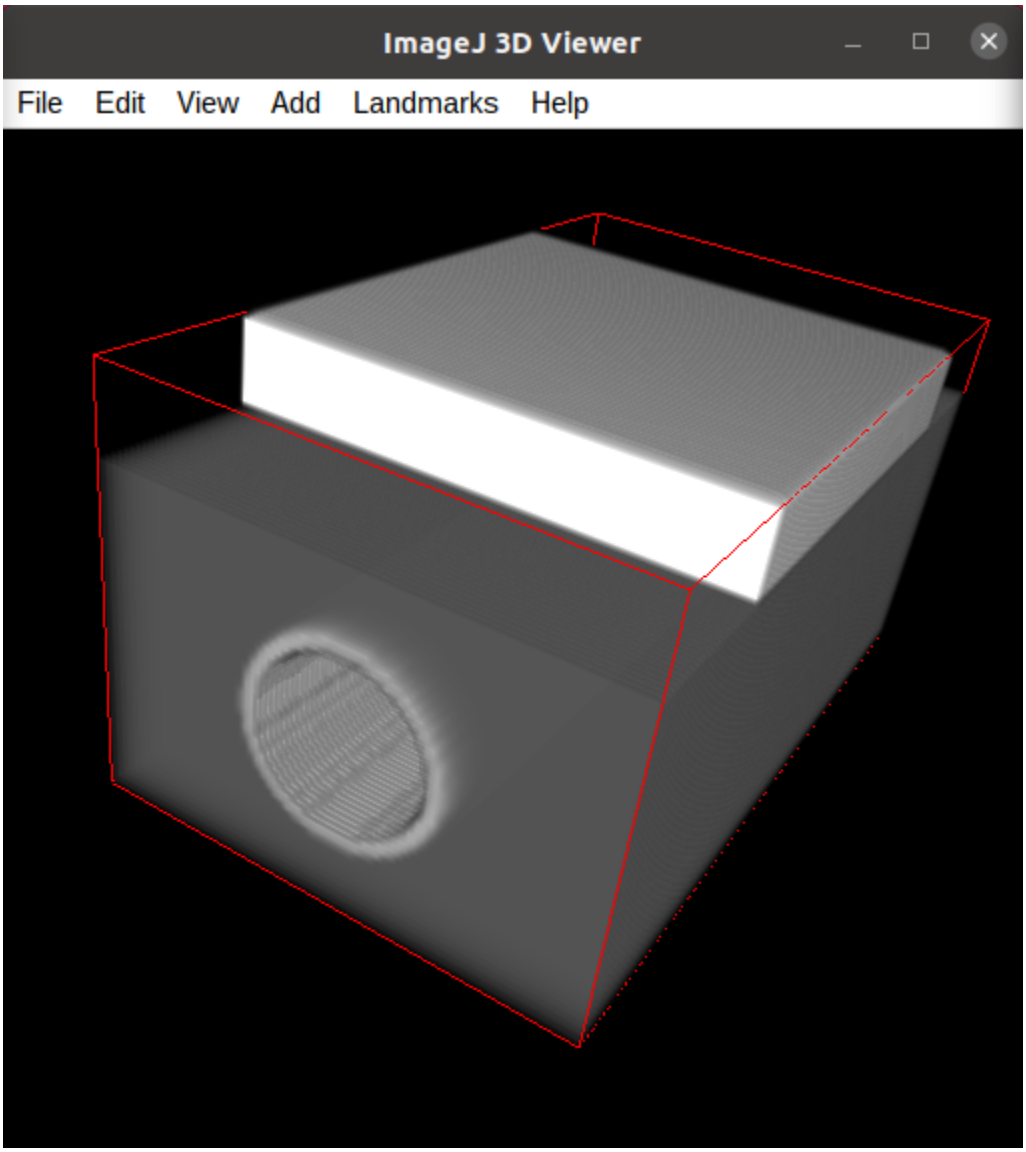


Fig. 6.17: 3D visualisation of the voxelised grid generated by **CAD2Vox** as visualised by **ImageJ**.

6.5.5 Example 3: Using non Salome med mesh files

This example involves using mesh formats other than Salome med. **CAD2Vox** itself actually uses the python package meshio. to read in mesh data. This package officially supports more than 30 common mesh formats. Therefore, if you have your mesh geometry in another format there is a good chance **CAD2Vox** will work 'out-of-the-box'.

There are however, 3 caveats to bear in mind:

1. We have not tested every possible format. We know that `.med`, `.stl`, `.ply`, and `.obj` all work as expected. You are welcome to try other formats as they should work however, your results may vary.
2. **CAD2Vox** can only work on meshes containing Triangles or Tetrahedrons. No other cell shapes are currently supported.
3. Greyscale values from material data are only officially supported with `.med` since `.obj`, `.ply` and `.stl` meshes don't contain material data. As such, for other mesh formats the greyscale is just set to white (255) for the entire mesh. You can change this value in `greyscale_Welsh-Dragon.csv` where you will find the "region" listed as "Undefined".

With these in mind, using a different mesh format through **VirtualLab** is as simple as setting `Vox.mesh` to a string containing the name of the file you wish to use including the file extension. You can then place the mesh in the same default directory as you would for a `.med` mesh. Or, as discussed earlier, you can use the absolute path to the file, again including the extension.

For our example we will use the Welsh Dragon Model which was released by Bangor university, UK, for Eurographics 2011. The model can be downloaded from here. This file should be placed in `Output/Examples/Dragon/Meshes` .

Action

The *RunFile* `RunTutorials.py` should be set up as follows to perform the voxelisation:

```
Simulation='Examples'
Project='Dragon'
Parameters_Master='TrainingParameters_Dragon'
Parameters_Var=None

VirtualLab=VLSetup(
           Simulation,
           Project
           )

VirtualLab.Settings(
           Mode='Interactive',
           Launcher='Process',
           NbThreads=1
           )

VirtualLab.Parameters(
           Parameters_Master,
           Parameters_Var,
           RunMesh=False,
           RunSim=False,
           RunDA=False
           )

VirtualLab.Voxelise()
```

We can then once again launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

The output is located in `Output/Examples/Dragon/Voxel-Images/Welsh-Dragon.Tiff`. You may notice that, since this mesh only contains triangle data, the resulting voxel image only contains coloured voxels on the surface of the model, see Fig. 6.18. You will also notice that, much like the dog bone in *example 1*, the model surface defaults to white (greyscale value of 255). This is because `.stl` files contain no information on materials. Therefore, as such, the entire mesh is set to a single greyscale value. Once again you can change this value in `greyscale_Welsh-Dragon.csv` if desired.

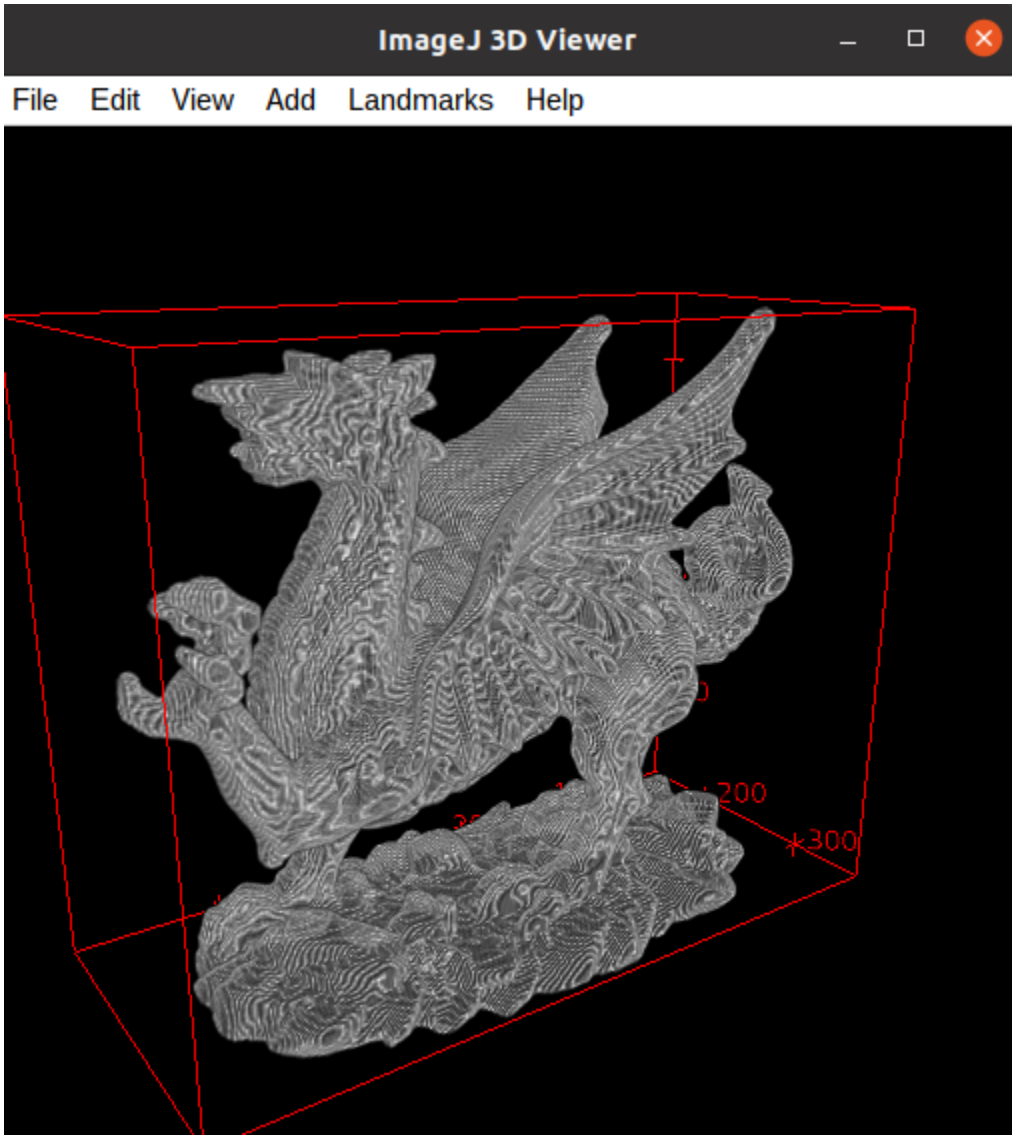


Fig. 6.18: 3D visualisation of the voxelised grid generated by **CAD2Vox** as visualised by **ImageJ**.

Auto-filling surface meshes

Because this final example uses a triangle mesh, one final thing you can try is changing the option `Vox.solid` to `True`.

This will use a different algorithm to auto-fill the interior volume. This works well on this particular mesh. However, if you wish to use this on your own surface meshes you will need to be aware of a few caveats:

1. The algorithm used is not robust. Therefore, depending on the geometry, it may work well but can sometimes leave holes in the mesh.
2. The algorithm used is also much slower than the normal surface algorithm.
3. In the current version of **CAD2Vox** materials are not implemented when using solid. This means that the voxels in the model will always have a greyscale value of 255. The code will also not generate a greyscale csv file and will simply ignore any that already exist.

6.6 Machine learning (HIVE)

6.6.1 Introduction

Machine learning (ML) is the science of getting computers to learn from data and make predictions without being explicitly programmed to do so. ML algorithms enable computers to self-learn by finding patterns and extracting features from data, which can then be used to make predictions on new data. Broadly speaking ML algorithms are broken down in to three categories; supervised learning, unsupervised learning and reinforcement learning.

ML is a widely used tool which we interact with on a daily basis. Examples include suggested web searches, face detection and recognition and ChatGPT language model, to name but a few.

The use of ML in science and engineering is also becoming more prevalent. It has a wide variety of applications in the field, such as; identifying tumours from images generated by scans, genome sequencing for drought resistant plants, processing huge quantities of data generated by experimental sensors. It is also commonly used to create surrogate models (often referred to as meta-models or emulators), which are often orders of magnitude faster to evaluate compared with simulations. This enables tasks such as parameter optimisation, sensitvity analysis or what-if analysis to be performed in a fraction of the time.

Surrogate models are an example of supervised learning algorithms, where the goal is to learn the relationship between the inputs and outputs of a simulation. The outputs of these surrogates can either be single value (SV) which are key properties of the simulation results, e.g. maximum stress, or full field (FF) which is the entire results one would expect from a simulation, e.g. temperature at each node of a mesh, for example.

These tutorials show how surrogate models can be used to greatly improve the insight gained for the HIVE experimental facility. In these tutorials the use of both SV and FF surrogate models are presented.

6.6.2 Data collection

To train surrogate models first simulation data from across the parameter space of interest must be collected. For the HIVE experiment there are 7 key parameters which effect the resulting temperature of the component; the displacement of the coil relative to the sample in the x,y and z direction, the rotation of the component relative to the pipe, the temperature and flow rate of the coolant, and the current in the coil. The range of values for each parameter is given in the below table, where the first 3 parameters are distances measured in metres, the rotation is measured in degrees, temperature in Celcius, coolant velocity in m/s and the current is measured in Amps.

Parameter	Range
Displacement in x	[-0.005,0.005]
Displacement in y	[-0.015,0.015]
Displacement in z	[0.003,0.006]
Rotation	[-5,5]
Coolant temperature	[30,80]
Coolant velocity	[0.65784,7.23626]
Current	[200,2000]

For this work 1000 data points are required, with 700 used for training the model and 300 used to test the performance of the model on unseen data. These are referred to as training and test datasets, with the ratio of 7:3 known as the train:test split. The RunFile used to perform these 1000 simulations and extract the necessary data can be found at `RunFiles/Tutorials/ML/HIVE_Example/DataCollect.py`.

Note: Performing these 1000 simualtions will be time consuming without sufficient computing resources. The data required for each task will be downloaded (if not already available), therefore this file can be used as a template to see how data can be collected.

6.6.3 Example 1: Power vs variation

The positioning of the coil relative to the component has a huge impact in terms of the thermal power delivered to the component. While moving the coil closer to the component (reducing the displacement in the z direction) will increase the quantitfy of thermal power, it also leads to a less uniform, more varying, heating profile on the coil adjacent surface. In a fusion device the heat loads are locally uniform, therefore a highly non-unfirom heating profile is not representative of the real conditions a component will be subjected to. It is therefore desirable to identify a coil positioning which maximises the power delivered to the component while minimising the variation of the heating profile on the coil adjacent surface.

In this example only the first 4 parameters effect the coil positioning and therefore the results generated by the **ERMES** EM solver, so it is only these parameters which one must consider. Therefore the models considered in this example will have 4 inputs, and 2 outputs; the power delivered to the component and the level of variation over the surface.

The RunFile used to perform this analysis can be found at `RunFiles/Tutorials/ML/HIVE_Example/PowerVariation.py`. At the top of the file are flags to dictate how the analysis will be performed.

```
CoilType = 'Pancake'
ModelType = 'MLP'
CreateModel = True
PVAnalysis = False
```

The *CoilType* means the coil which has been used in the simulations to generate the data. Next the *ModelType* states the type of supervised learning algorithm which will be used. There are two options available; a multi-layered perceptrion (MLP), which is a vanilla neural network, or Gaussian process regression (GPR). *CreateModel* is a flag to dictate whether or not to train a new model, while the *PVAnalysis* flag is to deicde whether or not to perform the power-variation analysis.

After VirtualLab has been initiated using the experiment 'HIVE' and project name 'ML_analysis', the DataFile is defined, which is the file where the data required to train the model is stored

```
DataFile = '{}_coil/PowerVariation.hdf'.format(CoilType)
```

If this file doesn't already exist it is downloaded from Zenodo (DOI: 10.5281/zenodo.8300663)

```
if not VirtualLab.InProject(DataFile):
    DataFileFull = "{}/{}".format(VirtualLab.GetProjectDir(),DataFile)
    print("Data doesn't exist, so downloading.")
    r = requests.get('https://zenodo.org/record/8300663/files/PowerVariation.hdf')
    os.makedirs(os.path.dirname(DataFileFull),exist_ok=True)
    with open(DataFileFull,'wb') as f:
        f.write(r.content)
```

To generate ML models **VirtualLab's** ML method is used. All models generated using the 'ML' method are stored in a sub-directory 'ML' in the project directory.

To begin with 3 MLP models with different architectures will be created to assess their performance. The master parameters for this are

```
ML.File = ('NN_Models','MLP_hdf5')
ML.TrainData = [DataFile, 'Features', [['Power'],['Variation']],{'group':'Train'}]
ML.ValidationData = [DataFile, 'Features', [['Power'],['Variation']],{'group':'Test'}]
ML.TrainingParameters = {'Epochs':1000,'lr':0.05}
ML.Seed = 100
```

File specifies that the analysis will be performed using the 'MLP_hdf5' routine in the file 'NN_Models' found in `Scripts/Common/VLRoutines`. This directory contains routines which are used by a variety of different experiments. The _hdf5 in the orutine name is there to specify that the model expects the data to be in a hdf5 file. *TrainData* specifies where the data which is used to train model can be found. The first argument is the name of the file where data is stored, the second argument is the name of the dataset which contains the values for the inputs, while the third is the name of the datasets which contain the values for the outputs of the model. In this example the values for the 4 input parameters are stored in the dataset 'Features', while the outputs of the model are the two values 'Power' and 'Variation'.

The fourth argument is an optional dictionary where additional information ca be provided. Here 'group' specifies the name of the group within the hdf5 file where these datasets can be found.

Note: Instead of using the group argument the entire path to the dataset could have been specified, e.g. 'Train/Features' for the second argument.

ValidationData has the same form as the TrainData but is taken from the group of data called 'Test'. This data is not used to train the model, but is monitored during training to ensure the model isn't overfitting the training data.

TrainingParameters is a dictionary of information which is used during the training of a model. 'Epochs' are the number of times the training data is iterated over, while 'lr' is the learning rate at which the weights of the model are updated.

The attribute *Seed* specifies the seed value to use to initiate any random sequences. Since the weights in an MLP are randomly generated this ensures that the model will always have the same set of initial weights to ensure reproducability.

Next are the parameters assigned to ParametersVar, which are the different architectures and names used for each model.

```
Architectures = [[32,32],[16,32,16],[8,16,8,4]]
for architecture in Architectures:
    ML.ModelParameters.append({'Architecture':architecture})
    arch_str = '_'.join(map(str,architecture))
    ML.Name.append("PV/{}/MLP/{}".format(CoilType,arch_str))
```

The first model will have two hidden layers with 32 nodes in each, the second model will have 3 hidden layers, with 16, 32 and 16 nodes respectively, and finally the third model will have 4 layers of sizes 8,16,8,4. These architectures are defined in the *ModelParameters* dictionary. For example the first of these three models will be saved in the directory `ML/PV/Pancake/MLP/32_32` in the project directory, assuming that the CoilType is 'Pancake'.

Once the models have been generated and saved their performance is compared against one another using the DA method.

```
DA.Name = "Analysis/{}/PowerVariation/MLP_Compare".format(CoilType) # results will be␣
↪saved to same directory as before
DA.File = ['PowerVariation','MLP_compare']
DA.MLModels = var_parameters.ML.Name # use the models defined earlier
DA.TestData = [DataFile, 'Features', [['Power'],['Variation']],{'group':'Test'}] #␣
↪unseen data to analyse performance
```

This uses the 'MLP_compare' routine found in the file `Scripts/Experiments/HIVE/DA/PowerVariation.py` to create a plot comparing the accuracy of the three models on the training and test data. The models to compare are defined using *MLModels*, which are simply the names of the models defined in ParametersVar.

Action

Ensure that *ModelType* is set to 'MLP' at the top of the RunFile and that *CreateModel* is `True` and *PV-Analysis* is `False`.

Launch **VirtualLab** with

```
VirtualLab -f RunFiles/Tutorials/ML/HIVE_Example/PowerVariation.py
```

You should see three models being generated and saved to the directories `ML/PV/Pancake/MLP/32_32`, `ML/PV/Pancake/MLP/16_32_16` and `ML/PV/Pancake/MLP/8_16_8_4` respectively. Along with this a plot comparing the normalised root mean square error (nRMSE) for the three models on the test an train data is created and can be found at `Analysis/Pancake/PowerVariation/MLP_Compare/Comparison.png`, which should look like Fig. 6.19

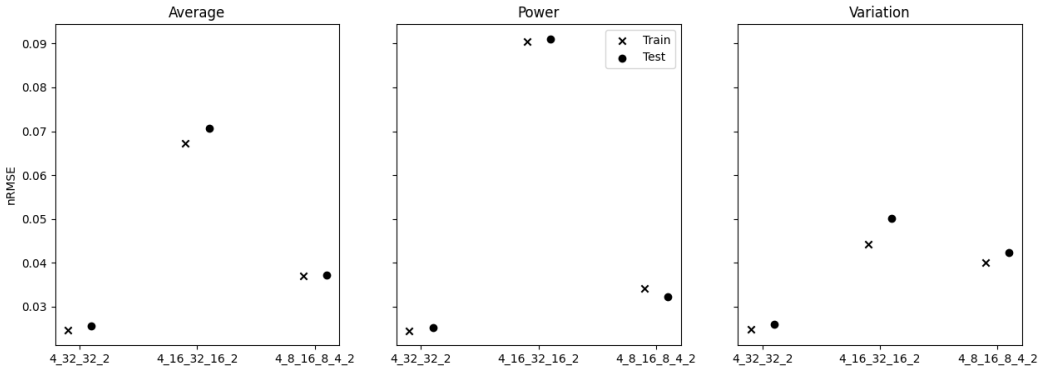


Fig. 6.19: Comparison of nRMSE of three different MLP architectures for predicting power & variation on test and train datasets.

Next the performance of three GPR models will be assessed. The parameters for this are similar to those for the MLP case, however in the *ModelParameters* dictionary this time it is the kernel of the GPR model which is varied.

```
ML = Namespace(Name = [],ModelParameters=[])
for kernel in GPR_kernels:
    ML.ModelParameters.append({'kernel':kernel})
    ML.Name.append("PV/{}/GPR/{}".format(CoilType,kernel))
```

These three models will be saved under the name of their kernel in the directory `ML/PV/Pancake/GPR`. As GPR models must invert matricies to make predictions these models will likely take longer to train compared with the MLP models.

Action

Change *ModelType* to 'GPR' at the top of the RunFile.

Launch **VirtualLab**

You should see three models being generated and saved to the directories `ML/PV/Pancake/GPR/RBF`, `ML/PV/Pancake/GPR/Matern_1.5` and `ML/PV/Pancake/GPR/Matern_2.5`. Along with this a plot comparing the performance of the three models will be created and can be found at `Analysis/Pancake/PowerVariation/GPR_Compare/Comparison.png`, which should look like Fig. 6.20.

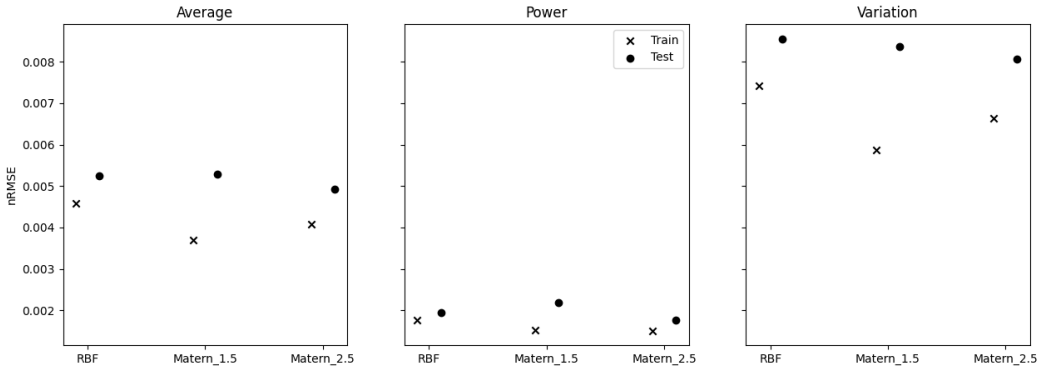


Fig. 6.20: Comparison of nRMSE of three different GPR kernels for predicting power & variation on test and train datasets.

You should notice that the nRMSE for the three GPR models are much lower than the MLP models, indicating that GPR is able to more accurately predict the power and variation for a given coil configuration.

Next the best model is used to create an envelope of the power a component can have delivered against the level of variation in the heating profile. As the best performing model the GPR model with the Matern_2.5 kernel is chosen, which is specified by the *MLModel* attribute of the DA method. This analysis is performed using the 'Insight_GPR' function in the `PowerVariation.py` file.

```
DA.Name = "Analysis/{}/PowerVariation/GPR_Analysis".format(CoilType)
DA.File = ['PowerVariation','Insight_GPR']
DA.MLModel = "PV/{}/GPR/Matern_2.5".format(CoilType)
```

Action

Keep *ModelType* as 'GPR' at the top of the RunFile and change *CreateModel* is `False` and *PVAnalysis* to `True`.

Launch **VirtualLab**

In the directory `Analysis/Pancake/PowerVariation/GPR_Analysis` a plot named `Envelope.png` is created, which will look like Fig. 6.21. This plot demonstrates that, for a given power delivered to the component there is a big difference in the variation of the heating profile.

This method enables HIVE's operators to identify configurations for the coil which result in the least amount of variation in the heating profile, thus better replicating the in-service conditions a component is subjected to.

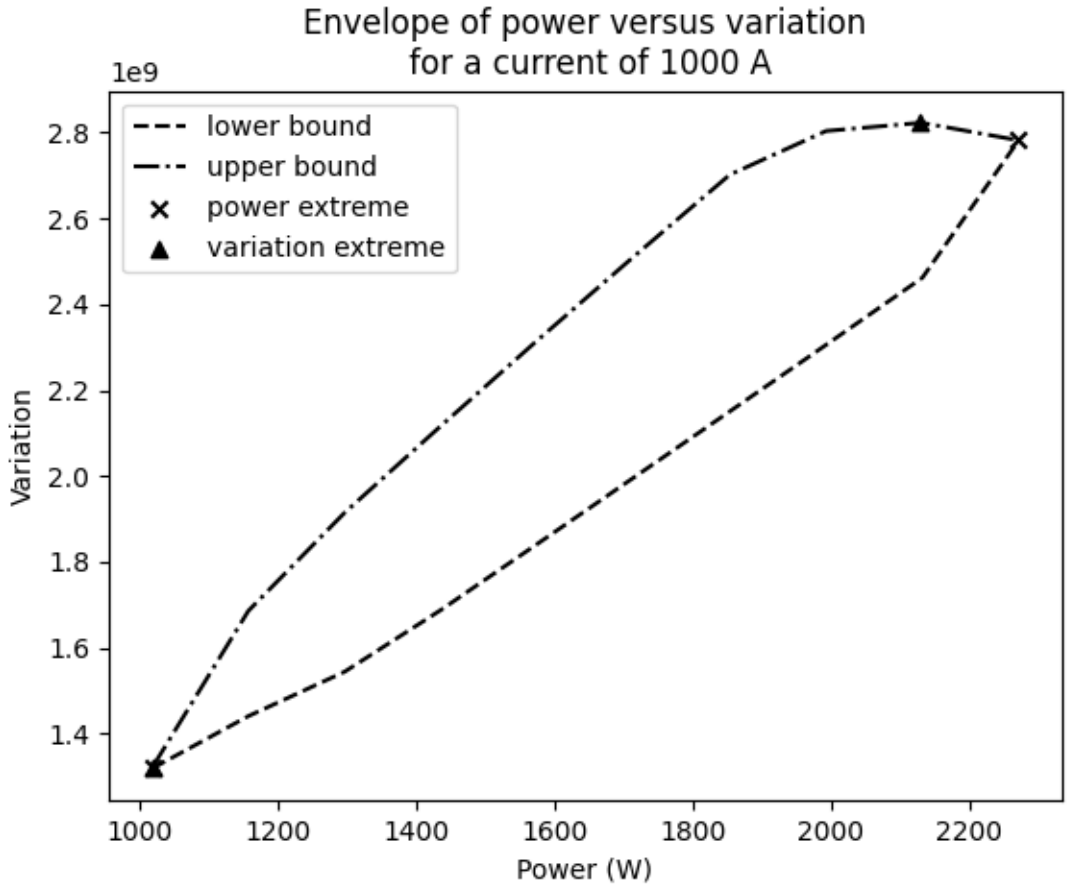


Fig. 6.21: Envelope of the power delivered to the component versus the variation score of the heating profile using the Matern_2.5 GPR model.

6.6.4 Example 2: Heating profile prediction

Often the operators of HIVE will want to be able to visualise the heating profile which a component is subjected to on the coil adjacent surface. Although the previous example showed a method for identifying desirable coil configurations, these would need to be used as inputs to a FEA simulation to first generate data for creating a visualisation of the heating profile. This example shows how it is possible to use ML models to predict and visualise the temperature on a 2D surface.

The RunFile used to perform this analysis can be found at `RunFiles/Tutorials/ML/HIVE_Example/HeatingProfile.py`.

At the top of the file are flags to dictate how the analysis will be performed and should look like this:

```
CoilType='Pancake'
PCA_Analysis = False
ModelType = 'GPR' # this can be GPR or MLP
CreateModel = True
CreateImages = True
```

The coil adjacent surface consists of 10,093 finite element nodes, the values for each of which we would like to predict. Creating a ML model which directly predicts that many outputs is at best impractical, and often unfeasible. This large number of outputs is compressed using the princial component analysis (PCA), which projects high dimensional data on to k-lower dimensional sub spaces.

PCA is a lossy compression algorithm, meaning that compressing the data and then reconstructing it will not return the original data. Fig. 6.22 shows the error between the original data and the reconstructed data against the number of principal components used to compress the data for the train and test data.

Note: PCA is able to perfectly reconstruct the train data as the principal components are optimal with respect to this data.

This plot also highlights the number of princial components needed to ensure that 99% (10) and 99.9% (41) of the variance in the data is retained. Generally, ensuring that a certain amount of variance in the data is retained is the most popular method by which to choose the number of principal components to use.

For this example, 11 principal components will be used because there is only a very small improvement in the reconstruction loss for a x4 increase in the number of principal components.

Note: If you'd like to generate this plot for yourself, make sure the PCA_Analysis at the top of the RunFile is set to `True`

Below are the parameters which are used to generate the GPR ML model:

```
ML.Name = 'HeatProfile/{}/GPR'.format(CoilType)
ML.File = ('GPR_Models','GPR_PCA_hdf5')
ML.TrainingParameters = {'Epochs':1000,'lr':0.05}
ML.TrainData = [DataFile, 'Features', 'SurfaceJH',{'group':'Train'}]
ML.ModelParameters = {'kernel':'Matern_2.5','min_noise':1e-8,'noise_init':1e-6}
ML.Metric = {'threshold':0.99}
```

The *File* attribute is similar to that from example 1, however this time we use a routine which will specifically compresses down the output using PCA. The *TrainData* attribute is also similar, with a dataset known as 'SurfaceJH' used for the output. The number of principal components to use is specified using *Metric*, where threshold will ensure at least 0.99 of the variance is retained. *ModelParameters* is again used to define the kernel used, which in this case is

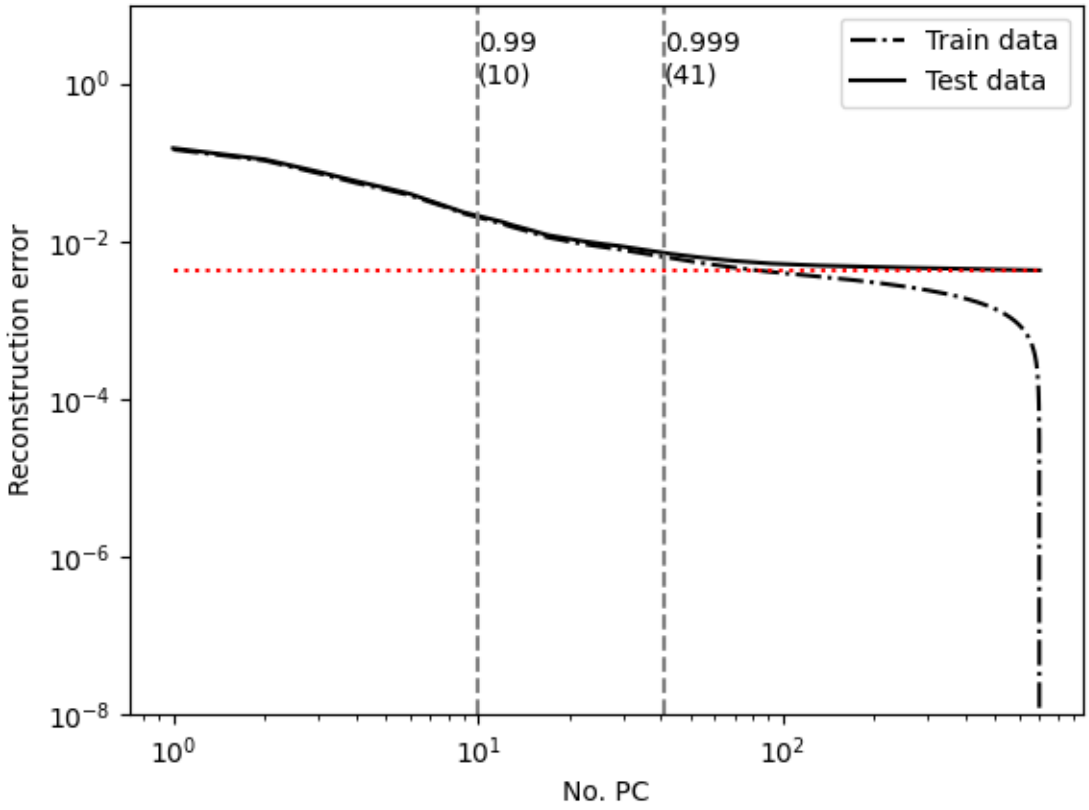


Fig. 6.22: Plot of reconstruction error using principal component analysis for Joule heating field on coil adjacent surface.

'Matern_2.5'. Along with this additional parameters relating to the noise of the model are set. This reduces the minimum bound of the noise parameter from 1E-3 set by GPyTorch to 1E-8, along with initialising its value at a smaller value. Again, *TrainingParameters* specifies the parameters used to train the model.

Next, the model will be used to generate images of heating profiles and compare them with simulations (on the test dataset). The simulations to compare are references using *DA.Index* in the 'CreateImage' section of the script. This is currently set to [1], meaning that a comparison for simulation number 1 will be performed.

Note: These images are generated using **ParaViS**. If you are using a virtual machine the GUI will need to be opened for the creation of images. This can be achieved by ensuring that *GUI* is set to `True` at the top of the file.

Action

Ensure that *ModelType* is 'GPR' at the top of the RunFile and that *CreateModel* and *CreateImages* are set to `True`.

Launch **VirtualLab** with

```
VirtualLab -f RunFiles/Tutorials/ML/HIVE_Example/HeatingProfile.py
```

Note: Generating the model may take 10 minutes or so, so feel free to grab yourself a coffee.

You should first notice that a model named 'HeatProfile/Pancake/GPR' is being generated. Following this the analysis is performed, with the resulting images saved to `Analysis/Pancake/HeatingProfile/GPR`. Here you will find the 'ground truth' Joule heating profile generated by **ERMES** (`Ex1_Simulation.png`) as shown in Fig. 6.23 along with that predicted by the GPR model (`Ex1_ML.png`) as shown in Fig. 6.24. You also have the absolute error between the two (`Ex1_Error.png`).

These plots show that there is sufficiently accurate agreement between the simulation and GPR model for predicting the Joule heating profile on the coil adjacent surface.

Action

Create images for other examples by changing *DA.Index* to any number(s) between 0 and 299 (since there are 300 simulations in the test dataset). For example

```
DA.Index = [4,11,32]
```

will generate comparison images for simulation 4,11 and 32.

You dont need to generate a new model, so ensure that CreateModel* is set to `False`.

This example shows how it would be possible for the operators of HIVE to visualise the heating profile generated by the positioning of the coil in a fraction of the time compared with a simulation. This enables more rapid decision making with regards to setting up the experiment.

Note: The above analysis has been performed using GPR models, however MLP models are also available. Feel free to change *ModelType* to 'MLP' and follow the same steps as the above.

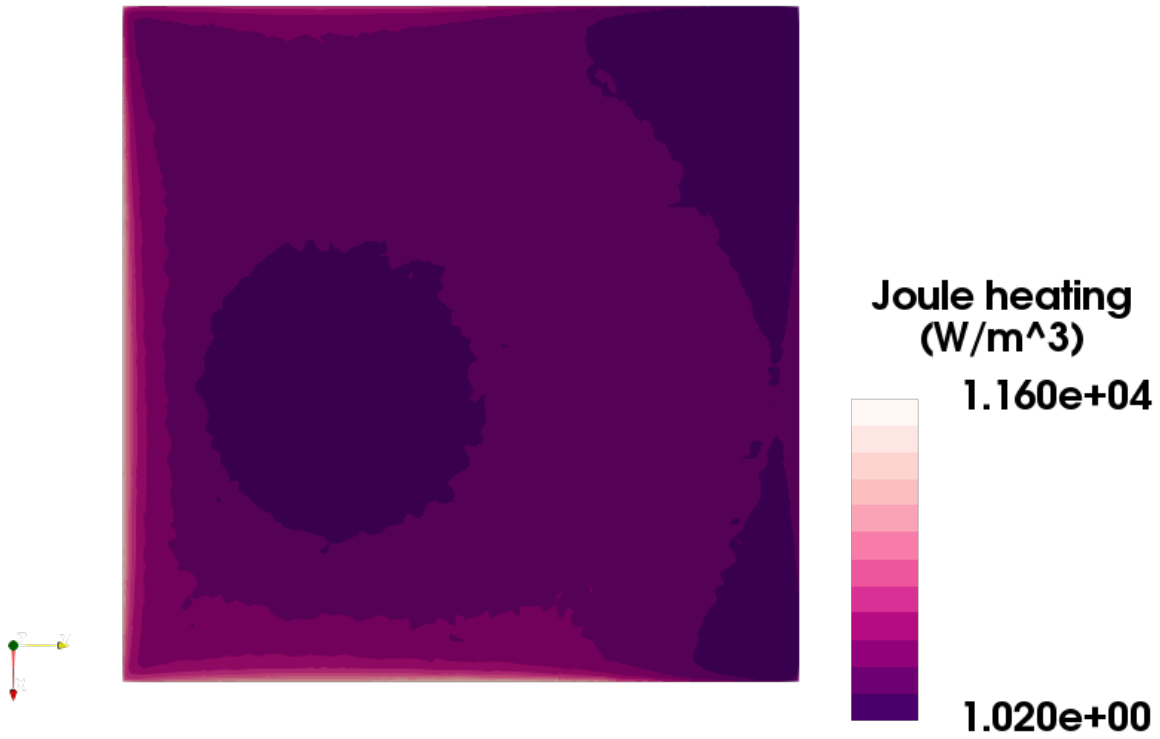


Fig. 6.23: Joule heating profile on coil adjacent surface from simulation (example 1)

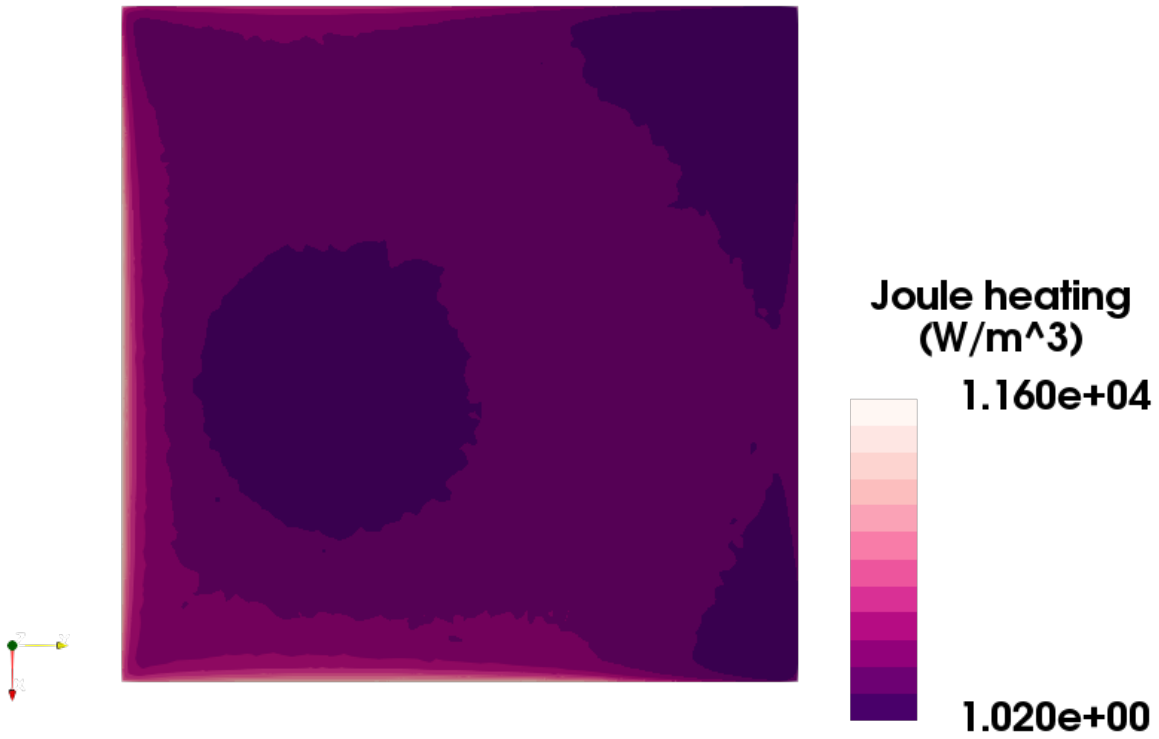


Fig. 6.24: Joule heating profile on coil adjacent surface from GPR model (example 1)

6.6.5 Example 3: Inverse solutions (temperature)

This example demonstrates how 3D surrogate models can be used to solve a variety of different inverse problem posed by HIVE. The work here builds on the previous example, showing how a 3D surrogate model of the temperature field can be generated. This will then be used to identify the experimental parameters which will deliver the maximum temperature to the component, along with those that deliver a certain desired temperature.

The RunFile used to perform this analysis can be found at `RunFiles/Tutorials/ML/HIVE_Example/InverseSolution_T.py`. At the top of the file are flags to dictate how the analysis will be performed and should look like this:

```
CoilType='Pancake'
PCA_Analysis = False
ModelType = 'GPR' # this can be GPR or MLP
CreateModel = True
InverseAnalysis = True
```

Fig. 6.25 shows the reconstruction error versus the number of principal components used to compress the data. The first thing to note is that much smaller errors are possible with this dataset compared with that in example 2. If 20 princial components are used then the reconstruction error is less than 1E-3 for both the test and train data, which shows that the compression is appropriate for this application.

Clearly more principal components could be used, however increasing to 200 would reduce the reconsturction error to around 1E-4, which is a large computational increase for only a small improvement in the ovrall accuracy. As a result, 20 principal components will be used for this analysis.

Note: If you'd like to generate this plot for yourself, make sure the PCA_Analysis at the top of the RunFile is set to `True`. This, however, may take a little while.

The parameters for generating the GPR model are similar to those in example 2. However, notice that this time *ML.Metric* specifies the number of principal components to use, instead of the variance threshold:

```
ML = Namespace()
ML.Name = 'Temperature/{}/GPR'.format(CoilType)
ML.File = ('GPR_Models','GPR_PCA_hdf5')
ML.TrainingParameters = {'Epochs':1000,'lr':0.05}
ML.TrainData = [DataFile, 'Features', 'Temperature',{'group':'Train'}]
ML.ModelParameters = {'kernel':'Matern_2.5','min_noise':1e-8,'noise_init':1e-6}
ML.Metric = {'nb_components':20}
```

Following this you have the parameters to perform analyses with the model. The key parameters here are *Index*, which indicates the index for which to generate comparison images like in the previous example, and *DesiredTemp*, which is the maximum temperature we would like the component to reach for us to identify the experimental parameters. These are currently:

```
DA.Index = [2]
DA.DesiredTemp = 600
```

Action

Ensure that *ModelType* is 'GPR' at the top of the RunFile and that *CreateModel* and *InverseAnalysis* are set to `True`.

Launch **VirtualLab** with

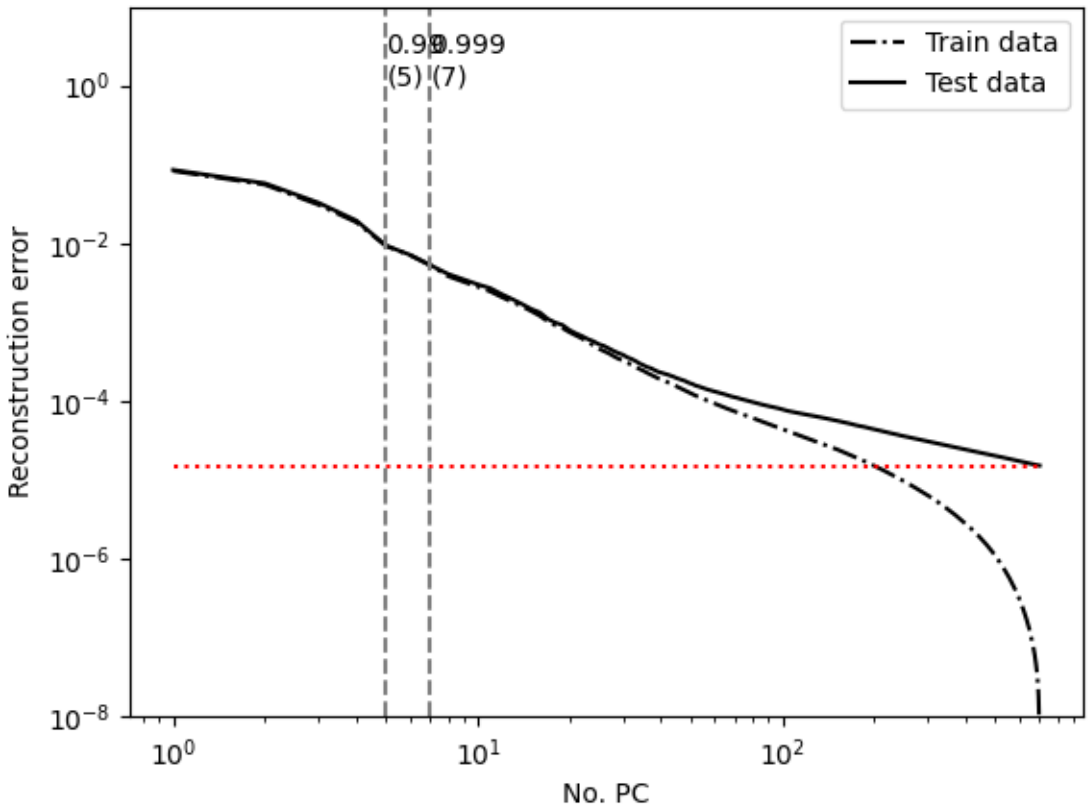


Fig. 6.25: Plot of reconstruction error using principal component analysis for temperature field

```
VirtualLab -f RunFiles/Tutorials/ML/HIVE_Example/InverseSolution_T.py
```

Note: This example uses **ParaViS** to generate images. If you are using a virtual machine the GUI will need to be opened for the creation of images. This can be achieved by ensuring that *GUI* is set to `True` at the top of the RunFile.

Note: Generating the model may take a little while, so feel free to grab yourself a coffee.

Fistly you will see the loss of the model reduce as the model parameters are updated using the training data, like in the previous examples. This model is saved to `Temperature/Pancake/GPR` in the `ML` directory in the project directory.

Following this the analysis with the model will take place. Printed to the terminal you should see the parameter combination which will deliver the maximum temperature to the component within the defined parameter space. This should look like the following:

```
Parameter combination which will deliver a maximum temperature of 1333.90 C:

-5.00e-03, -1.50e-02, 3.00e-03, -5.00e+00, 7.99e+01, 4.17e+00, 2.00e+03
```

Many of these values are intuitive. The third value, displacement in the z direction, is the minmum value possible (coil is as close to the component as possible), with the fourth value - the rotation - pushing the coil even closer to the component. The fifth value is the coolant temperature, which is at the maximum value of 80 °C, while the seventh value is the current, which is also at its maximum value of 2000 A. An image of the temperature field at the maximum temperature can be found at `MaxTemperature.png` in the results directory `Analysis/Pancake/InverseSolution_T/GPR`, and should look like Fig. 6.26.

While the above problem is somewhat trivial, often the goal of a HIVE experiment is to reach a certain maximum temperature within a component, or deliver a certain temperature to a specific part of the component. This type of problem is much less intuitive due to the combination of a high number of experimental parameters. There are also, usually, a number of combination of parameters which will deliver the desired result. The next part of the output provides 5 combinations of experimental parameters which will deliver a maximum temperature to the component specified by *DesiredTemp*, which in this case is 600 °C. These should look like:

```
2.02e-04, -7.33e-03, 4.36e-03, -3.30e+00, 4.46e+01, 5.58e+00, 1.63e+03
-4.43e-03, -1.81e-04, 5.04e-03, 2.36e+00, 7.19e+01, 4.84e+00, 1.76e+03
-3.32e-03, -9.34e-03, 3.82e-03, -3.30e+00, 4.35e+01, 3.78e+00, 1.45e+03
2.82e-03, 7.08e-03, 5.33e-03, 8.09e-01, 6.76e+01, 3.17e+00, 1.78e+03
-2.57e-03, 6.57e-03, 3.56e-03, -1.67e+00, 5.28e+01, 3.43e+00, 1.48e+03
```

Images for each of the 4 temperature field are saved to the result directory, highlighting the multiple different temperature profiles which will deliver a maximum temperature of 600 °C.

Alongside these images you will find a comparison for the example specified by *Da.Index* between the temperature profile generated by the GPR model and by the simulation. In this case, this was example number 2. The temperature profile from the simulation and GPR model are shown in Fig. 6.27. and Fig. 6.28. respectively.

This example shows how a 3D GPR surrogate model can solve one of the most prevalent inverse problems posed by HIVE, providing not only the experimental parameters but also images of their resulting temperature field.

Action

Change *ModelType* to 'MLP' at the top of the file and re run the analysis.

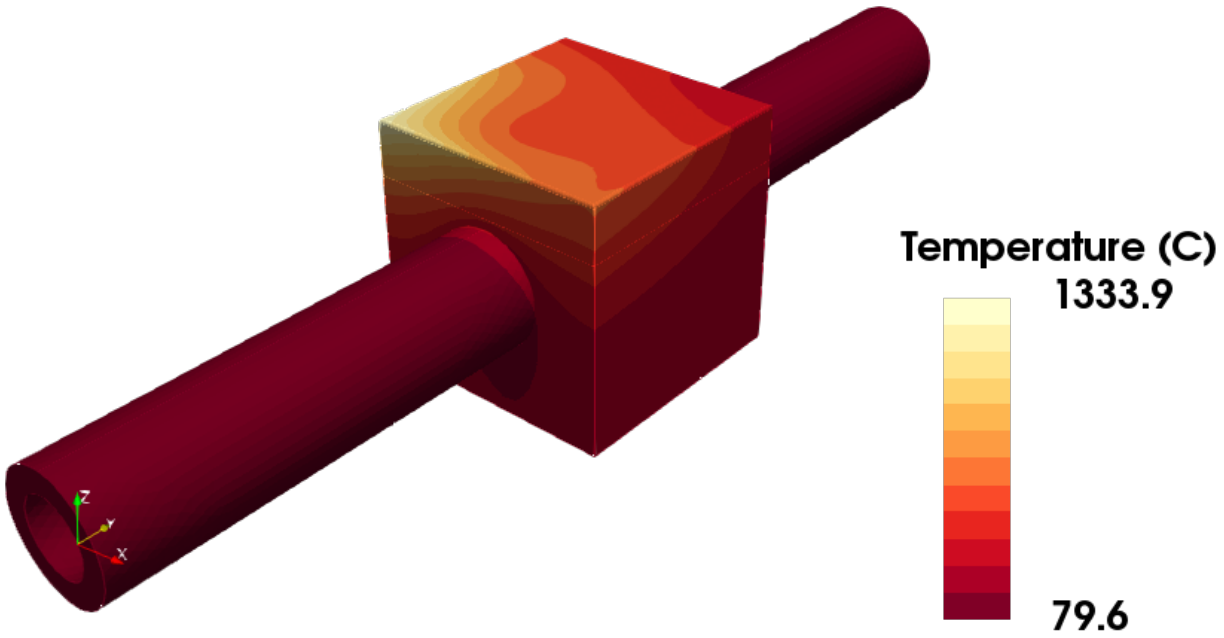


Fig. 6.26: Temperature profile which delivers the maximum temperature within a defined parameter space.

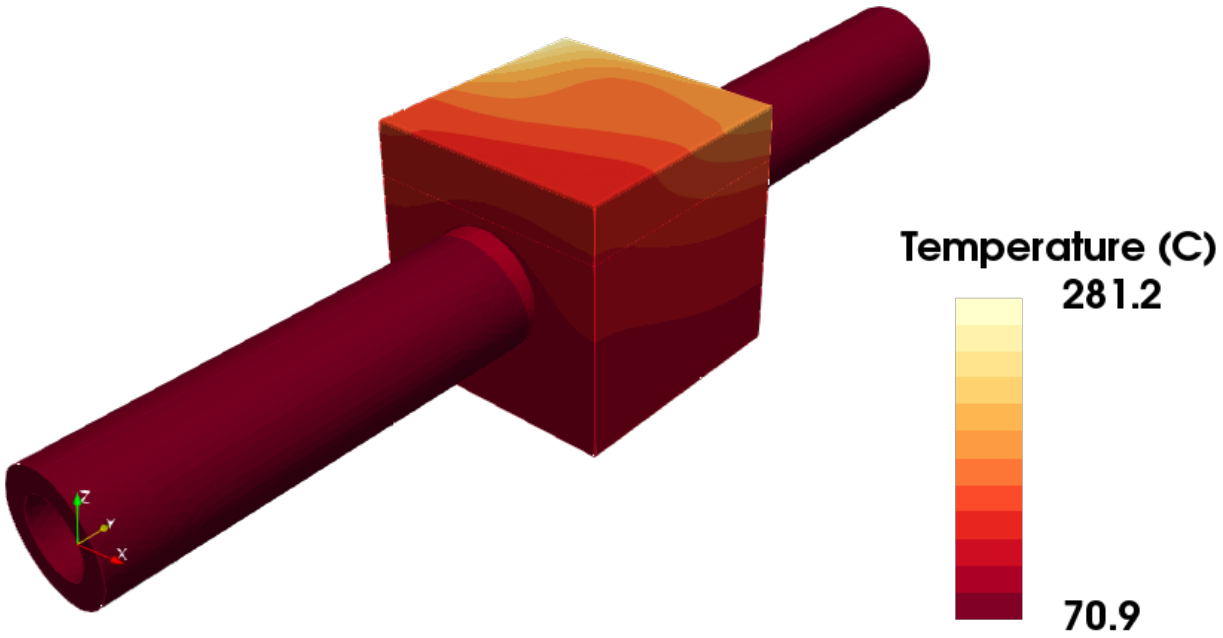


Fig. 6.27: Temperature profile from simulation (example 2)

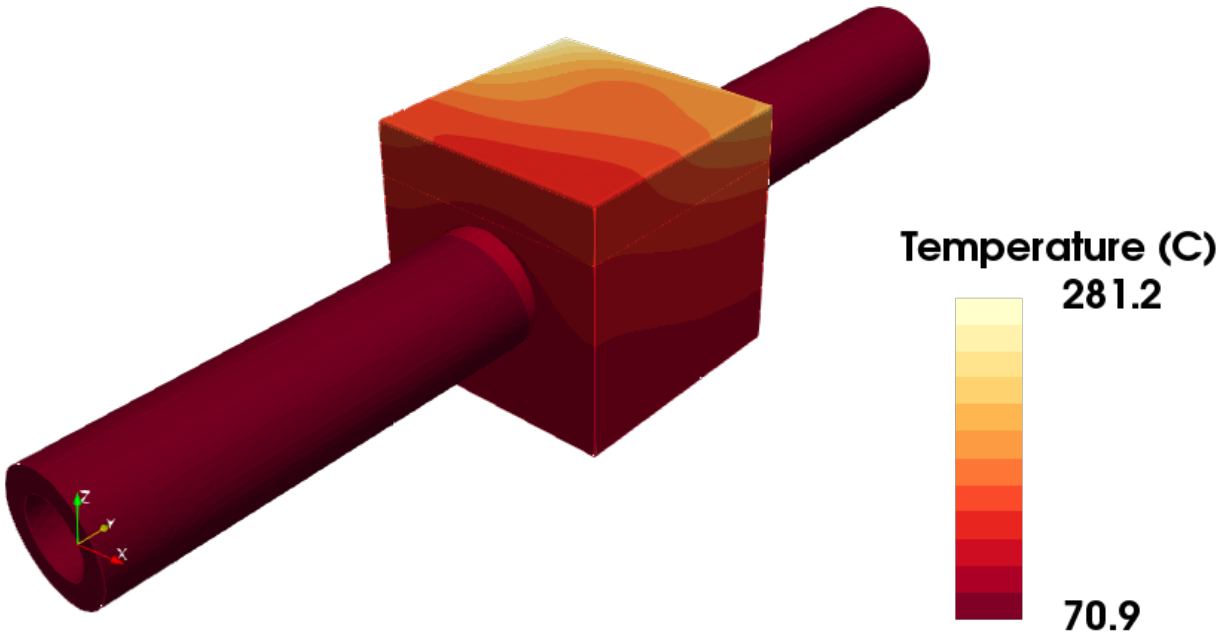


Fig. 6.28: Temperature profile using ML model (example 2)

You should notice that while an MLP trains faster than a GPR model it is less accurate.

6.6.6 Example 4: Inverse solutions (Von Mises)

In this example a surrogate model of the Von Mises stress field is used in conjunction with the temperature field surrogate generated in the previous example to identify more complex inverse solutions.

The RunFile used to perform this analysis can be found at `RunFiles/Tutorials/ML/HIVE_Example/InverseSolution_VM.py`. At the top of the file are flags to dictate how the analysis will be performed and should look like this:

```
CoilType='Pancake'
PCA_Analysis = False
ModelType = 'GPR' # this can be GPR or MLP
CreateModel = True
InverseAnalysis = True
```

Fig. 6.29 shows the reconstruction error versus the number of principal components used to compress the Von Mises stress nodal data. Notice that the reconstruction error is higher for the Von Mises stress compared with the temperature data from the previous example. To achieve a reconstruction error of 1E-3 with this data around 100 principal component would be required, which is relatively large. Instead 20 principal components will be used, which will still ensure that more than 99.9% of the variance is retained.

Note: If you'd like to generate this plot for yourself, make sure the PCA_Analysis at the top of the RunFile is set to `True`. This, however, may take a little while.

The parameters for generating the GPR model are identical to those in the previous example, with the only difference being that the name of the dataset used for the model output is now 'VonMises':

```
ML.Name = 'VonMises/GPR'
ML.File = ('GPR_Models','GPR_PCA_hdf5')
ML.TrainingParameters = {'Epochs':1000,'lr':0.05}
ML.TrainData = [DataFile, 'Features', 'VonMises',{'group':'Train'}]
ML.ModelParameters = {'kernel':'Matern_2.5','min_noise':1e-8,'noise_init':1e-6}
ML.Metric = {'nb_components':20}
```

Following this you have the parameters to perform analysis with the model. Notice that in this example both the temperature and Von Mises ML models are used:

```
DA.Name = 'Analysis/{}/InverseSolution_VM/GPR'.format(CoilType)
DA.File = ('InverseSolution','AnalysisVM_GPR')
DA.MLModel_T = 'Temperature/{}/GPR'.format(CoilType)
DA.MLModel_VM = 'VonMises/{}/GPR'.format(CoilType)
```

The other parameters used in this analysis are the same as the previous example:

```
DA.Index = [2]
DA.DesiredTemp = 600
```

Index is the index of the test data which will be used to compare the output of the model with the 'ground truth' simulation, as we did in the previous example.

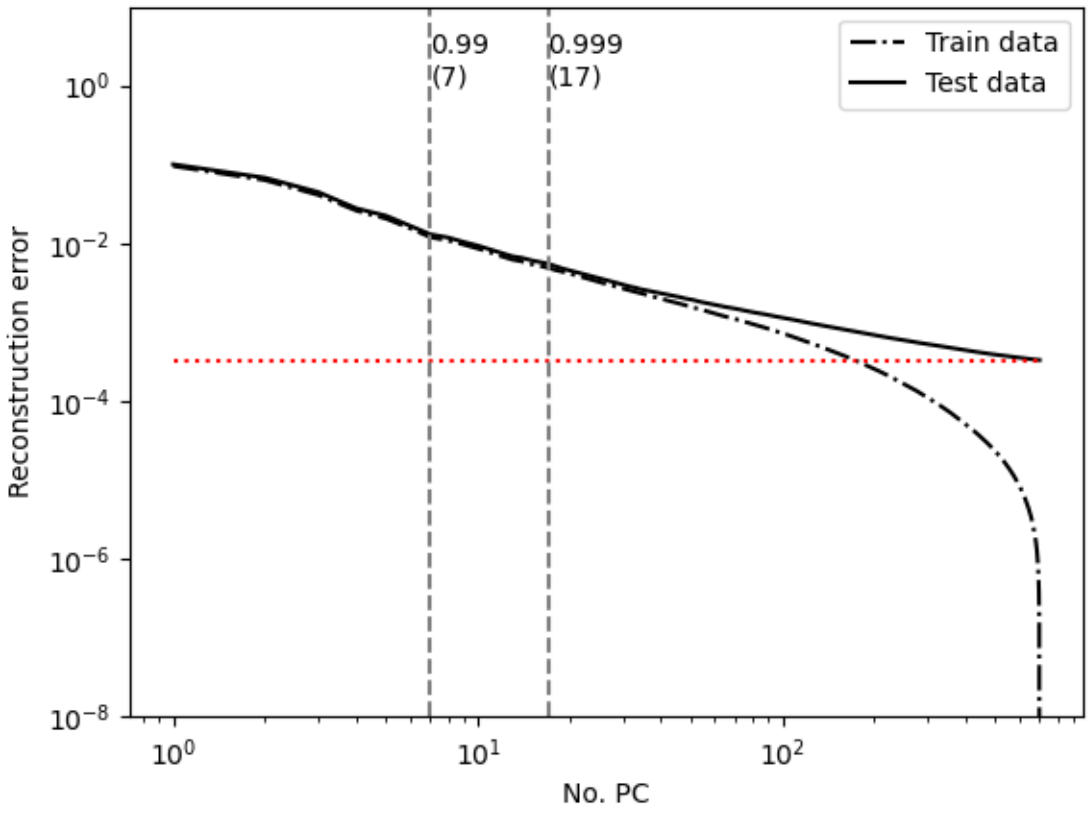


Fig. 6.29: Plot of reconstruction error using principal component analysis for temperature field

Action

Ensure that *ModelType* is 'GPR' at the top of the RunFile and that *CreateModel* and *InverseAnalysis* are set to `True`.

Launch **VirtualLab** with

```
VirtualLab -f RunFiles/Tutorials/ML/HIVE_Example/InverseSolution_VM.py
```

Note: This example uses **ParaViS** to generate images. If you are using a virtual machine the GUI will need to be opened for the creation of images. This can be achieved by ensuring that *GUI* is set to `True` at the top of the RunFile.

Note: Generating the model may take a little while, so feel free to grab yourself a coffee.

The inverse analysis performed first is to identify the experimental parameters which will provide the maximum amount of Von Mises stress in the component. You should notice an output like this:

```
Parameter combination which will deliver a maximum Von Mises stress of 965.04 MPa:

4.12e-03, -1.09e-02, 3.00e-03, -5.00e+00, 7.99e+01, 6.58e-01, 2.00e+03
```

Many of these are as we'd expect, with the coil displacement in the z direction at 3.00e-03, it's minimum value, along with the coolant temperature at its maximum value (80 °C) and the current also at the maximum (2000 A). This combination of parameters will result in a Von Mises stress of 965 MPa. An image of the Von Mises stress field using these parameters can be found at `Analysis/Pancake/InverseSolution_VM/GPR/MaxVonMises.png`.

DesiredTemp is again the maximum temperature we want the component to reach, however as we have the von Mises model we would like to go a step further. The previous example showed a variety of different temperature profiles where the maximum temperature of 600 °C is delivered, each of which will result in a different stress fields in the component. Therefore, it is desirable to identify the experimental parameters which will maximise the Von Mises stress while ensuring that 600 °C is delivered to the component. The output for this should look like this:

```
Parameter combination which delivers 600.00 C and maximises the Von Mises stress,␣
↪delivering 586.28 MPa:

4.56e-03, 7.27e-03, 5.51e-03, -5.00e+00, 3.00e+01, 7.22e+00, 1.92e+03
```

Note: Using two models for the optimisation may be slightly time-consuming.

An image of the temperature field and Von Mises stress field using these parameters can be found at `T600_T.png`. and `T600_VM.png` in `Analysis/Pancake/InverseSolution_VM/GPR`.

Alongside these you will find `Ex2_Simulation.png` `Ex2_ML.png` and `Ex2_Error.png` which show a comparison of the model output with the simulation for example 2 (that which was specified using *DA.Index*).

Note: You can perform the same analysis again using an MLP model if you'd like.

6.6.7 Example 5: Thermocouple optimisation

One of the instruments used to collects data from an experiment on HIVE is by using thermocouples. Thermocouples are probes which are joined to the surface of a component prior to an experiment and provide pointwise temperature data. Unfortunately this data does not provide a huge amount of understanding about the component's behaviour, especially at locations the thermocouples can't measure, e.g. the inside of the component. Knowledge of the full temperature field throughout the component would greatly improve the understanding of the component and its suitability for a fusion device.

In this example, the temperature surrogate models generated in example 3 are used to predict what the temperature field is throughout the component by using simulated thermocouple data. Using examples from the test dataset, temperature at thermocouple locations are extracted and it is assumed that this is the only information we have.

Following this, the sensitivity of the placement of the thermocouples is presented, along with a method of optimising their location.

The RunFile used to perform this analysis can be found at `RunFiles/Tutorials/ML/HIVE_Example/Thermocouple.py`. At the top of the file are flags to dictate how the analysis will be performed and should look like this:

```
CoilType='Pancake'
ModelType = 'MLP' # this can be GPR or MLP
EstimateField = True
Sensitivity = False
Optimise = False
```

Notice that *ModelType* in this example is 'MLP', which is chosen because it's evaluation is substantially faster compared with GPR, which is useful for the optimisation of the thermocouple locations.

Note: The MLP model for the temperature field should have been created in example 3. This will need to be completed before the analysis of this example can take place.

To estimate the field from the thermocouple, firstly the placement of the thermocouples is required. This is specified using the *ThermocoupleConfig* attribute

```
DA.ThermocoupleConfig = [['TileSideA',0.5,0.5],
                         ['TileFront',0.5,0.5],
                         ['TileSideB',0.5,0.5],
                         ['TileBack',0.5,0.5],
                         ['BlockFront',0.5,0.5],
                         ['BlockBack',0.5,0.5],
                         ['BlockBottom',0.5,0.5]]
```

Here each list represents a thermocouple, with the first value being the surface which the thermocouple is attached to, with the next 2 the positioning on the surface (scaled to [0,1] range). This configuration is for 7 thermocouples, with each placed at the centre of the respective surface, see Fig. 6.30 - 6.32.

These are the 7 thermocouples which will be used, with the temperature data extracted from example 7 of the test dataset (again specified using *Index*).

Action

Ensure that *ModelType* is 'MLP' at the top of the RunFile and that *EstimateField* is set to `True`, while *Sensitivity* and *Optimise* are both `False`.

Launch **VirtualLab** with

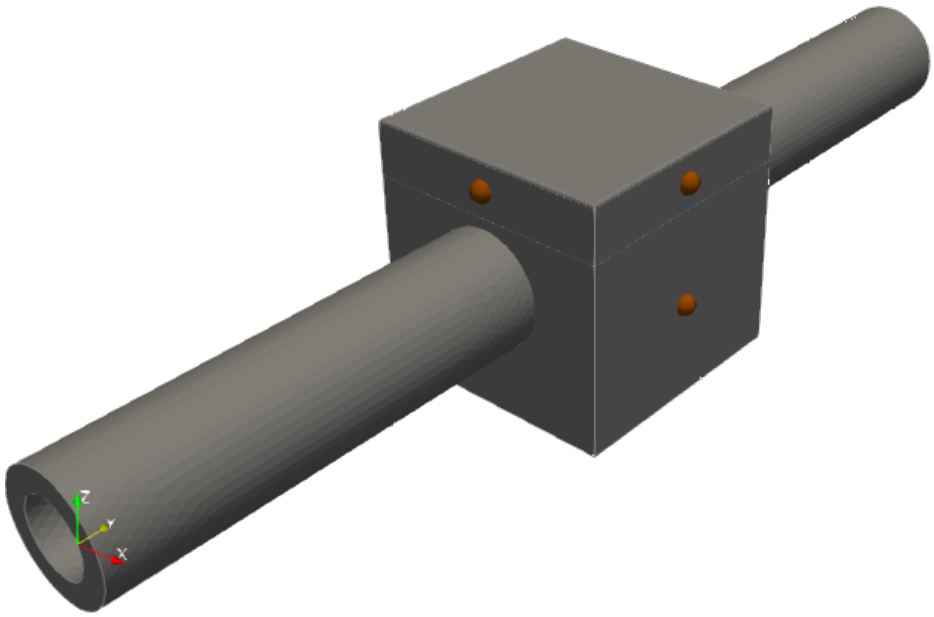


Fig. 6.30: Thermocouples at centre of surfaces (viewpoint 1)

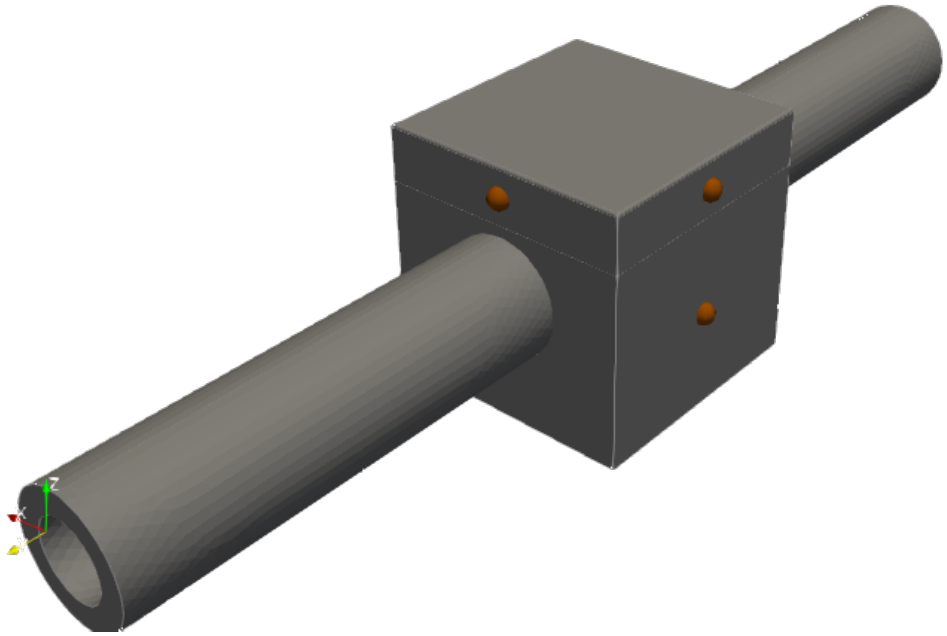


Fig. 6.31: Thermocouples at centre of surfaces (viewpoint 2)

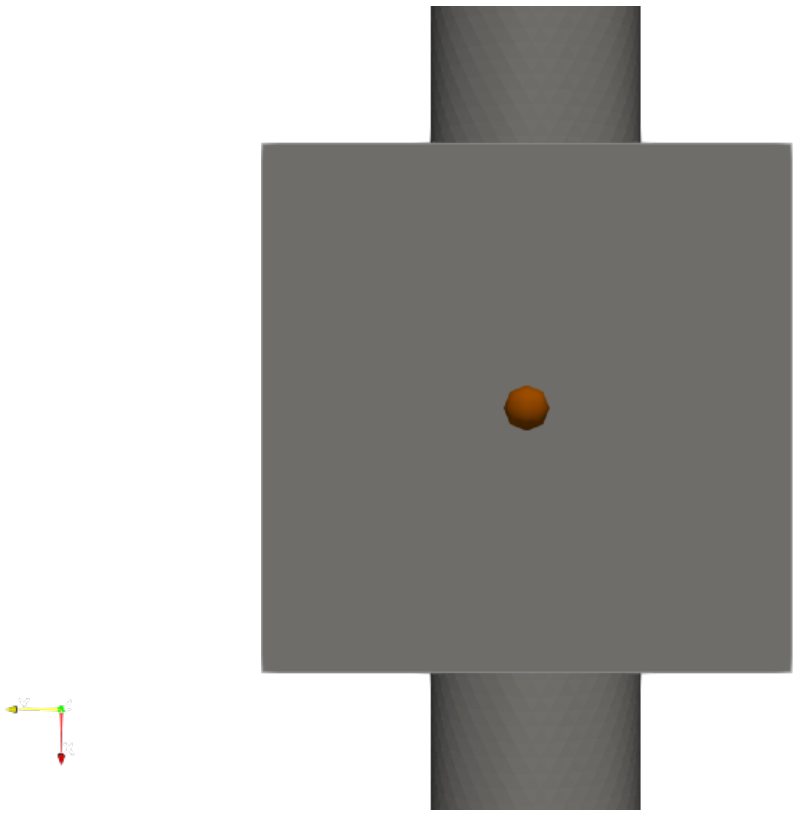


Fig. 6.32: Thermocouples at centre of surfaces (viewpoint 3)

```
VirtualLab -f RunFiles/Tutorials/ML/HIVE_Example/Thermocouple.py
```

In the directory `Analysis/Pancake/Thermocouple/MLP/EstimateField` you will find `Ex7_Simulation.png` which shows the temperature field predicted by the simulation, while `Ex7_ML.png` shows the temperature field estimated by the surrogate model using the temperature at the 7 thermocouple locations. An error plot is also provided in `Ex7_Error.png`, highlighting good agreement between the two. This shows that it is possible to estimate a full temperature field using only 7 surface temperature points.

Adding thermocouples to components is a time-consuming task, therefore it is desirable to use as few of them as possible. Where the thermocouples are placed has a big impact on whether or not the original temperature field can be reconstructed.

The next task will look at 5 random configuraitions of 4 thermocouples to see how many temperature fields fit to them. These are decided using the *NbConfig* and *NbThermocouple* attributes

```
DA.CandidateSurfaces = ['TileSideA','TileSideB','TileFront','TileBack','BlockFront',
↪'BlockBack','BlockBottom']
DA.NbThermocouples = 4
DA.NbConfig = 5
```

The *CandidateSurfaces* attribute is simply the different surfaces where thermocouples can be placed.

Action

Change *EstimateField* is set to `False` and *Sensitivity* to `True`.

Launch **VirtualLab**

In `Analysis/Pancake/Thermocouple/MLP/Sensitivity` you will find `PlacementSensitivity.png`, which is also shown in Fig. 6.33.

This plot provides a score for each configuration. This score is the number of temperature fields which fit the temperature data provided, averaged over 5 test cases. A score of 1 represents a perfect score, since this means that in all 5 cases only a single temperature field fit to the data.

Configuration 3 provides a very low score, showing that this would be a good choice compared with the others. A visualisation of the thermocouple placements which gave this score can be found in `TC_configs/Config_3`. Visualisation of the other, less impressive, configurations can also be found in the `TC_configs` directory.

The above raises the question regarding an optimal number and configuration of thermocouples. The next task will look at identifying the optimal configuration of thermocouples. Here, we use the gradient-free genetic algorithm to find an optima, see here for more details.

Similar to above, here we define the *CandidateSurfaces* and *NbThermocouples*, but we also define *GeneticAlgorithm*

```
DA.GeneticAlgorithm = {'NbGen':5,'NbPop':20,'NbExample':5,'seed':100}
```

GeneticAlgorithm is a dictionary containing information for running the genetic algorithm optimisation. 'NbGen' is the maximum number of generations which the optimisation will run for, while 'NbPop' is the population size. 'NbExample' is the number of testcases to average the score over, while 'seed' seeds the initial population for reproducability. Additional parameters can also be passed to the algorithm, see routine 'Optimise_MLP' in `Scripts/Experiments/HIVE/DA/Thermocouple.py`.

Action

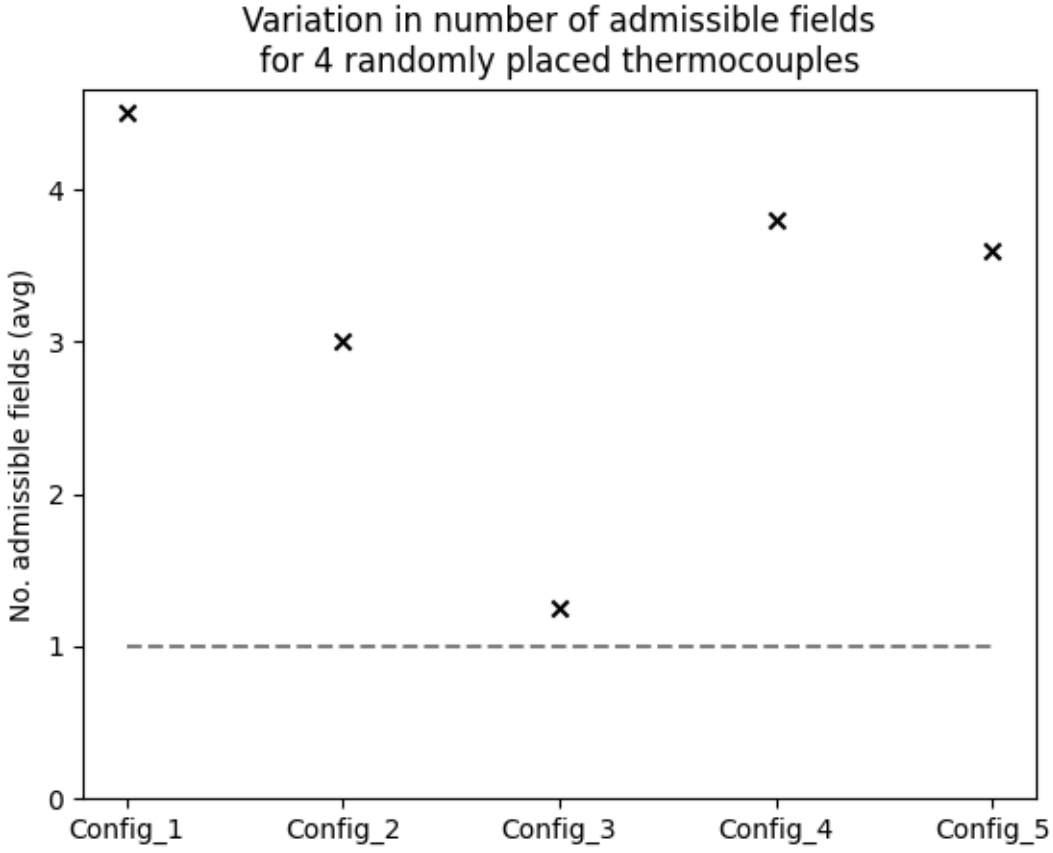


Fig. 6.33: Thermocouples at centre of surfaces (viewpoint 3)

Change *Sensitivity* is set to `False` and *Optimise* to `True`.

Launch **VirtualLab**

In the terminal information relating to the genetic algorithm will be printed, mainly the current best score and configuration found.

You should find that with 4 thermocouples it is possible to identify a configuration which will deliver a perfect score of 1. The placement of these thermocouples can be seen in `Analysis/Pancake/Thermocouple/MLP/Optimise_4/OptimalConfig`.

Action

Perform the same analysis again but with NbThermocouple = 3.

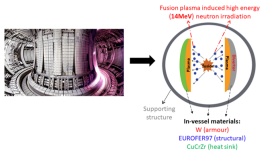
Launch **VirtualLab**

You should notice that when only 3 thermocouples are used the lowest score is well above the optimal score of 1, meaning that 4 thermocouples are required to accurately predict the temperature field throughput the component.

6.7 Irradiation Damage

6.7.1 Introduction

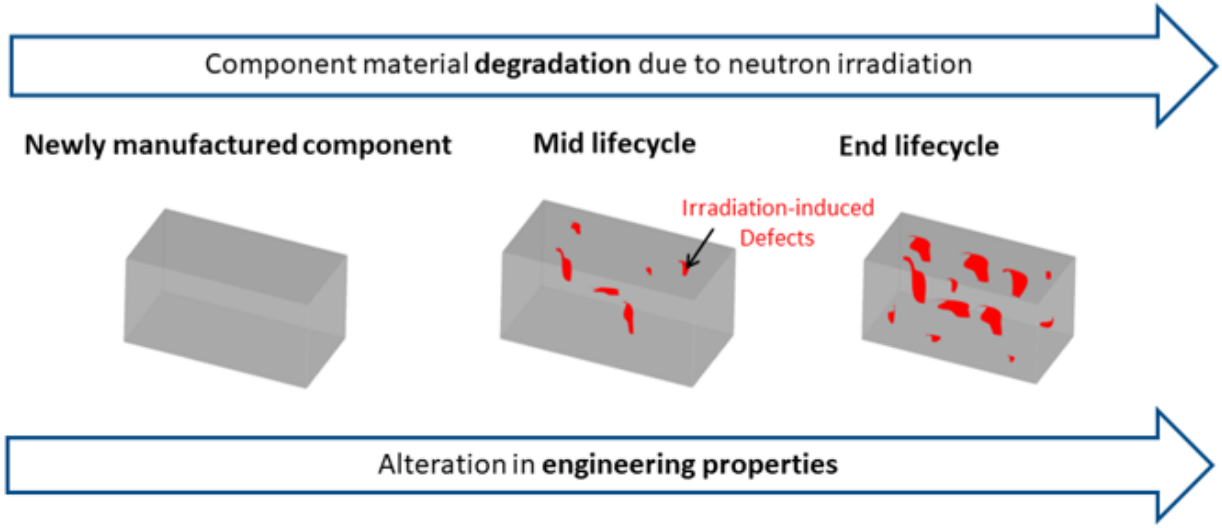
The hostile irradiation conditions in a tokamak, a fusion energy device, induce changes in the engineering properties of the materials which consequently lead to a degradation of performance for in-vessel components during their lifecycles. Material properties change at different rates as a function of the irradiation dose received and other factors such as temperature. The kind of dose received at a point within the component depends on the attenuation path between the source and location (i.e., distance and the materials between the source and point), temperature (which depends on the geometry, loading conditions and efficiency of the part as a whole), neutron flux, neutron energy, and material cross-sections. This problem is highly non-linear and crosses multiple length-scales making the prediction of how the performance of a part will evolve over its lifecycle extremely challenging.



The methodology used implements a multi-scale numerical model to analyse the influence of neutron irradiation-induced defects on the mechanical behaviour of in-vessel components. The developed workflow integrates open-source software (developed by others) for Monte-Carlo based neutronics, dislocation dynamics (DD) and finite element analysis (FEA) in such a way that gives the flexibility as a general solver to investigate current and novel tokamak components exposed to various irradiation doses and temperature conditions. This work has the potential to transform engineering design for fusion energy by being able to assess a design's performance across its whole lifecycle.

The workflow is employed in VirtualLab in the form of a multi-scale hierarchical computational predictive capability to link the neutron irradiation-induced micro/nano scale defects response to the mechanical behaviour of tokamak components as shown in Fig. 6.34.

The letters (A)-(H) in the Fig. 6.34 denotes various stages/components of the workflow incorporated VirtualLab. The neutron heating values are calculated in the Monte-Carlo based neutronics simulations from OpenMC (B) which are implemented as thermal loads for finite element simulation (FEA) in Code_Aster (C). The damage energy obtained from



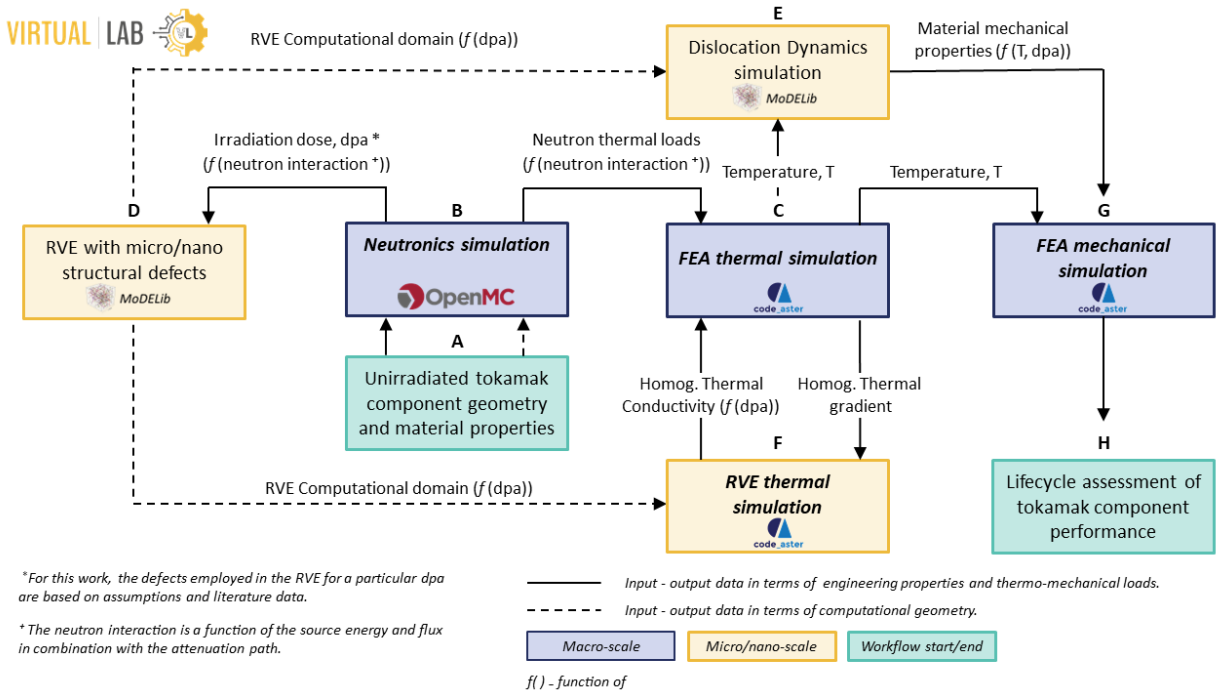


Fig. 6.34: Schematic diagram of the workflow for the developed platform.

the OpenMC simulation is used to calculate the displacements per atom (dpa) which in turn facilitates the implementation of dose dependent irradiation-induced defects as input for Dislocation dynamics (DD) (D, E). In fact, OpenMC (B) is employed within the open-source software package Paramak and Code_Aster is within the Salome-Meca. For the mechanical FEA simulation (G), the Von-Mises plasticity model with isotropic hardening is implemented. The yield strength and stress-strain data for the plasticity model is obtained from the DD simulation performed in the software 'Mechanics of Defect Evolution library' (MoDELib) (D, E) on an uniaxially loaded 'Representative Volume Element' (RVE) for a given fusion relevant material with neutron irradiation-induced defects. These packages are in turn linked together within VirtualLab.

A multi-scale homogenisation technique is implemented to calculate the effective thermal conductivity due to irradiation-induced defects in the RVE of the fusion relevant component. In the multi-scale homogenisation method, the RVEs with the defect densities generated from MoDELib (D) are assigned at the Gauss points of the FEA mesh. The temperature gradients at the gauss points from the FEA thermal simulation (C) are imposed as the boundary condition for the RVE with irradiation-induced defects and RVE thermal simulations (F) are performed using FEA. The resultant effective/homogenised thermal conductivity is employed for macro-scale FEA thermal simulation (C). The thermal fields from FEA thermal simulation (C) and RVE with irradiation-induced defect densities (D) based on temperature and dpa values are employed for the DD simulation (E) to obtain the yield strength.

The results from the FEA thermal simulation and mechanical simulation are subjected to lifecycle assessement (H) using plastic flow localisation rule.

6.7.2 Sample

To demonstrate the developed platform's potential, a case study has been carried out for a tungsten armour monoblock from the divertor region. Due to the parametric nature of the platform, the dimensions can easily be amended to facilitate investigations of alternative designs. The monoblock consists of tungsten (W) armour as the outer component, copper (Cu) as an interlayer and an inner CuCrZr cooling channel as shown in the Fig. 6.35.

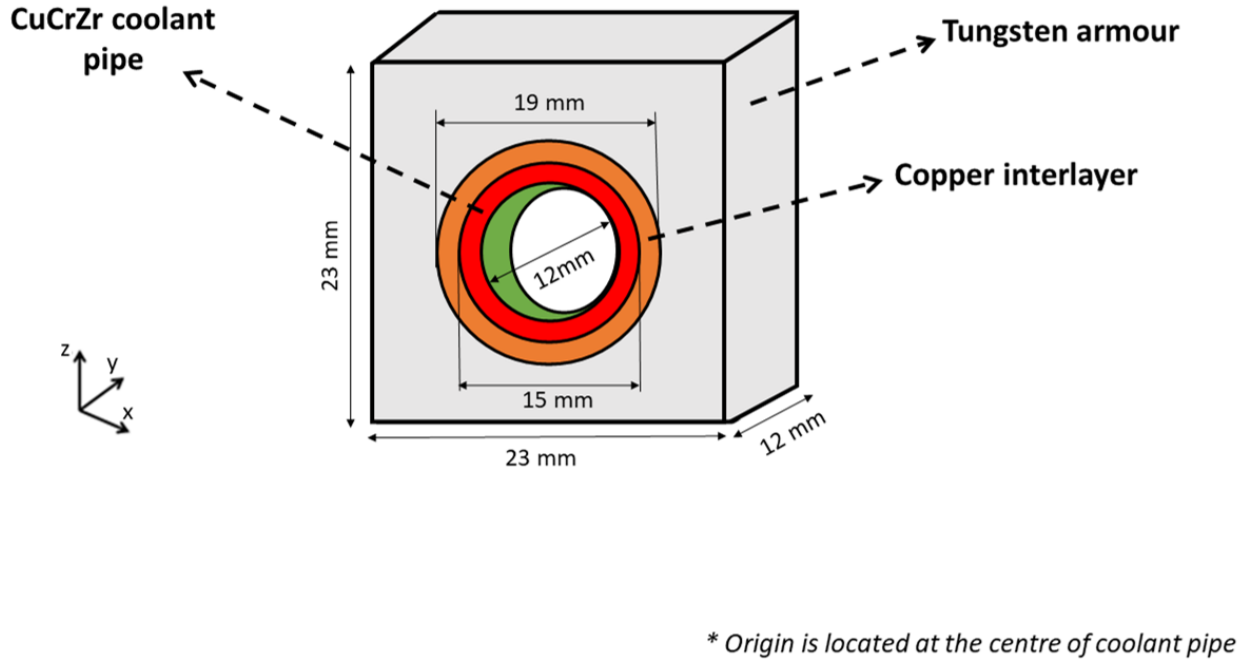


Fig. 6.35: Schematic representation of Tungsten monoblock.

`Mesh` contains all the variables required by **SALOME** to create the CAD geometry and subsequently generate its mesh.

```
Mesh.Name = 'mono'
Mesh.File = 'monoblock'
```

Mesh.File defines the script used by **SALOME** to generate the mesh, which in this case is `Scripts/Experiments/Irradiation/Mesh/monoblock.py`.

Once the mesh is generated it will be saved to the sub-directory `Meshes` of the project directory as a `MED` file under the user specified name set in *Mesh.Name*. In this instance the mesh will be saved to `Output/Irradiation/Tutorials/Meshes/mono.med`.

The attributes of `Mesh` used to create the sample geometry in `monoblock.py` are:

```
# Geometric Parameters
# Origin is located at the centre of the CuCrZr coolant pipe
Mesh = Namespace()
Mesh.pipe_protrusion = [.05] # length of pipe between monoblocks
Mesh.Warmour_height_lower=[1.15] # Lower tungsten armour height from the origin
Mesh.Warmour_height_upper=[1.15]# Upper tungsten armour height from the origin
Mesh.Warmour_width=[2.3] # Width of tungsten monoblock
Mesh.Warmour_thickness=[1.2] # Thickness of tungsten monoblock
Mesh.copper_interlayer_thickness=[.2]# Copper interlayer thickness
Mesh.pipe_radius=[.6] # Radius of CuCrZr coolant pipe
Mesh.pipe_thickness=[.15] # Thickness of CuCrZr coolant pipe
Mesh.mesh_size=[6] # Size of Mesh
Mesh.prot_mesh=[1] # Size of Mesh for length of pipe between monoblocks
Mesh.arm_ext=[0] # total monoblock height = Warmour_height_lower + Warmour_height_lower␣
↪+ arm_ext
Mesh.seg_diag=[4] # size of mesh at the diagonal line between copper interlayer and␣
↪tungsten armour
```

The attributes of `Mesh` used to create the CAD geometry and its mesh are stored in `monoblock.py` alongside the `MED` file in the `Meshes` directory.

The generated mesh is shown in the Fig. 6.36.

6.7.3 Neutronics simulation (B)

OpenMC (B) is employed which implements Monte-Carlo code to model the neutron transport, heating, and PKAs in fusion conditions (14 MeV). The nuclear heating values generated from the reactions are computed using nuclear data processing code, NJOY, implemented within OpenMC package. To calculate the dpa across the tokamak components, the damage energy per source particle is obtained based on the Material Table (MT) = 444 within the HEATR module of NJOY in OpenMC.

The ENDFB-7.1 nuclear data from the NNDC OpenMC distribution is employed for the neutronics calculation (B). The geometry creation and neutronics simulation are performed in the Paramak software package. The monoblock CAD geometry is created using CadQuery and is converted to OpenMC neutronics model by means of DAGMC . To perform the simulation in OpenMC, the cross-section and mass density of the materials in the monoblock are required. The cross-sections of the materials are obtained from the ENDFB-7.1 nuclear data. The simulation is performed for 500,000 particles per batch and a total of 50 batches is irradiated on the monoblock from an isotropic fusion energy source with 14 MeV monoenergetic neutrons. The scored neutron heating and damage energy (MT = 444) values are tallied onto the OpenMC mesh of the monoblock. Since the tallied results of neutron heating are in electron Volts (eV), it is multiplied by the source strength of 1 GW fusion DT plasma and divided by the volume of the corresponding cells to obtain the neutron heating values in terms of W·m-3. From the tallied damage energy results, the dpa across the monoblock is calculated based on the threshold energy of the material with some assumptions on recombination factor.

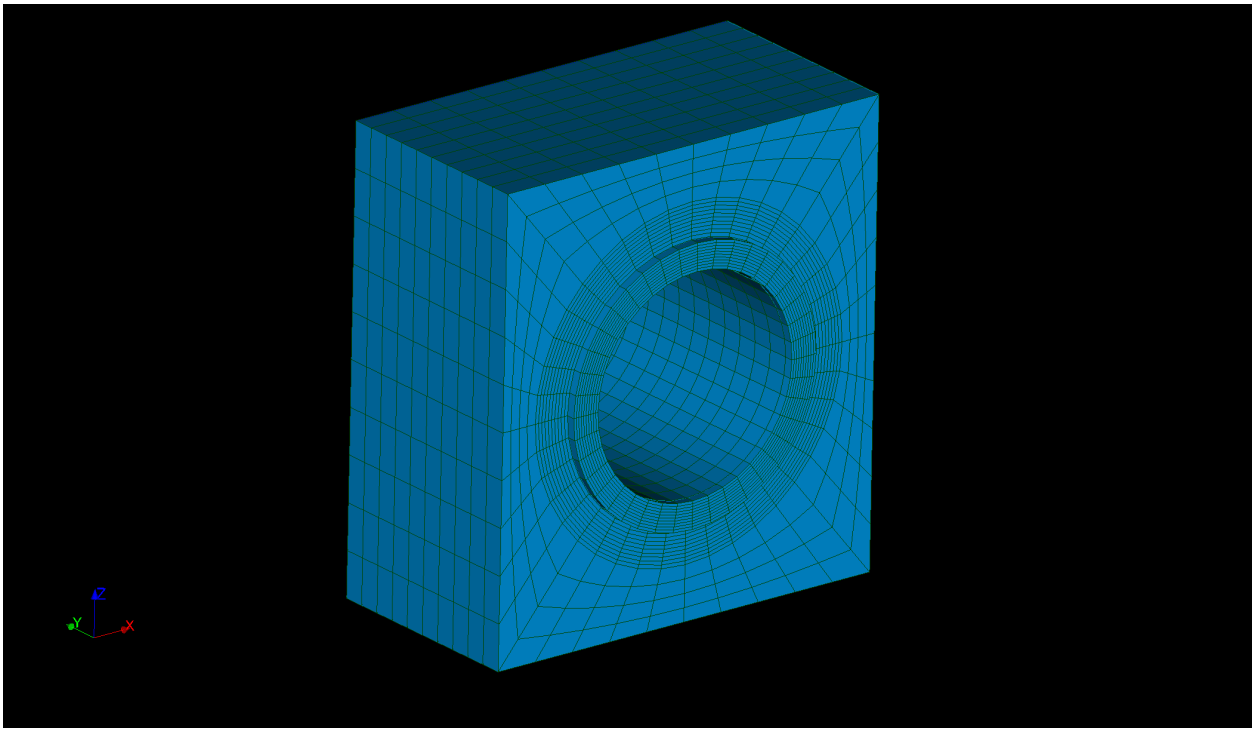


Fig. 6.36: Computational mesh of Tungsten monoblock.

The dpa calculated in the study is an approximation which is calculated in terms of atom-based estimate of material exposure to neutron irradiation in fusion relevant conditions.

`Paramak` contains all the variables required by **Paramak** software package to create the CAD geometry:

```
Paramak = Namespace()
Paramak.Name = ['irradiated_day1000']
```

neutronics_cad located in `Scripts/Experiments/Irradiation/Paramak/neutronics_cad.py` defines the script used by **Paramak** to generate the cad geometry for neutronics simulation.

Once the cad is generated, the output file 'dagmc.h5m' will be saved to the sub-directory `Output/Irradiation/Tutorials/'irradiated_day1000/dagmc.h5m` in *Paramak.Name*.

The attributes of `Paramak` used to create the sample cad geometry are:

```
# Geometric Parameters for neutronics simulation
Paramak.Warmour_height_lower=[1.15,]
Paramak.Warmour_height_upper=[1.15]
Paramak.Warmour_width=[2.3]
Paramak.Warmour_thickness=[1.2]
Paramak.copper_interlayer_radius=[.95]
Paramak.copper_interlayer_thickness=[.2]
Paramak.pipe_radius=[.6]
Paramak.pipe_thickness=[.15]
Paramak.dagmc=['dagmc.h5m']
Paramak.pipe_length=[1.2]
Paramak.pipe_protrusion=[.05]
```

`Openmc` contains all the variables required by **Openmc** software package to perfrom neutronics simulation:

```
Openmc = Namespace()
Openmc.Name = ['irradiated_day1000']
```

neutronics_simulation located in `Scripts/Experiments/Irradiation/Openmc/neutronics_simulation.py` defines the script used by **Openmc** to generate the cad geometry for neutronics simulation.

Once the simulation is completed, the output file 'damage_energy_openmc_mesh.vtk' and 'heating_openmc_mesh.vtk' will be saved to the sub-directory `Output/Irradiation/Tutorials/irradiated_day1000` in *Openmc.Name*.

The attributes of `Openmc` used to perform neutronics simulation are:

```
Openmc.Warmour_height_lower=[1.15] # Lower height of tungsten block from origin
Openmc.Warmour_height_upper=[1.15] # Upper height of tungsten block from origin
Openmc.Warmour_width=[2.3] # Width of tungsten monoblock
Openmc.Warmour_thickness=[1.2] # Thickness of tungsten monoblock
Openmc.pipe_protrusion=[.05] # Length of cucrzr coolant pipe between monoblocks
Openmc.source_location=[9.5] # Neutron source location
Openmc.thickness=[25] # Mesh size along monoblock thickness
Openmc.height=[50] # Mesh size along monoblock height
Openmc.width=[50] # Mesh size along monoblock width
Openmc.damage_energy_output=['damage_energy_openmc_mesh.vtk']
Openmc.heat_output=['heating_openmc_mesh.vtk']
Openmc.dagmc=['dagmc.h5m']
```

The tallied neutron heating values and damage energy values of monoblock are stored in the output file 'damage_energy_openmc_mesh.vtk' and 'heating_openmc_mesh.vtk' will be saved to the sub-directory `Output/Irradiation/Tutorials/irradiated_day1000` in *Openmc.Name*. However, these values are generated for cell values of the mesh. In order to convert cell values to node values, paraview is implemented.

`paraview` contains all the variables required by **paraview** software package to convert cell values to node values in the output file generated by Openmc simulation using script `Scripts/Experiments/Irradiation/Mesh/neutronics_post.py`

```
paraview = Namespace()
paraview.Name = ['irradiated_day1000']
paraview.File=['neutronics_post']
```

Two files are generated: 'heating_openmc_mesh_pv.vtk' for neutron heating and 'damage_openmc_mesh_pv.vtk' for damage energy across the monoblock as shown in which will be saved to the sub-directory `Output/Irradiation/Tutorials/'irradiated_day1000/ in *Openmc.Name* which as depicted :numref:`Fig. %s <heating>` and `Fig. %s`

The tallied neutron heating values are converted to finite element mesh by means of Code_Aster script:file:*Scripts/Experiments/Irradiation/Sim/neutronics_heating.comm*

The attributes of `Sim` used for the conversion of tallied neutron heating values are converted to finite element mesh are:

```
Sim = Namespace()
Sim.Name=['irradiated_day1000']
Sim.AsterFile = ['neutron_heating']
Sim.Mesh = ['mono']
Sim.width_mesh=[50] # mesh size across width of tungsten monoblock
Sim.height_mesh=[50] # mesh size across height of tungsten monoblock
Sim.thic_mesh=[25] # mesh size across thickness of tungsten monoblock
```

(continues on next page)

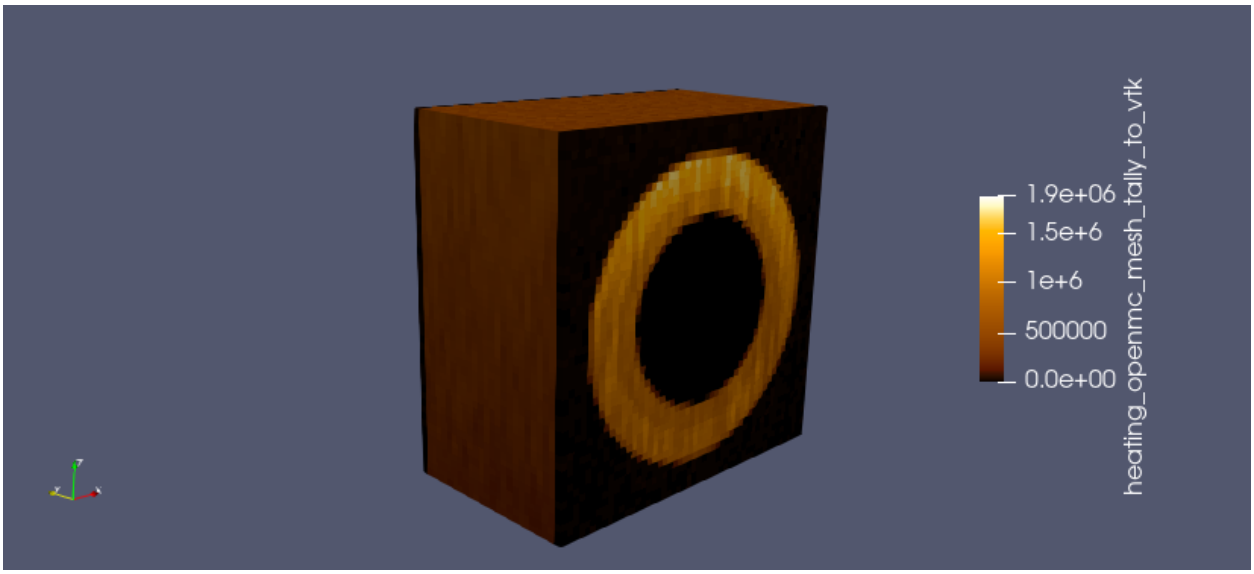


Fig. 6.37: Neutron heating across Tungsten monoblock.

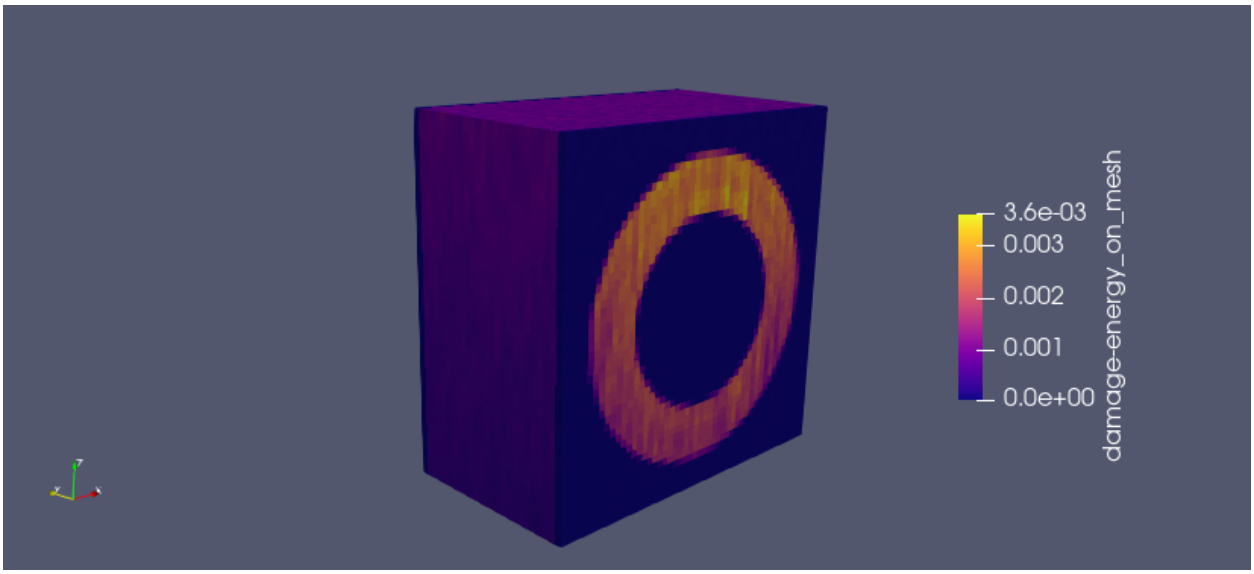


Fig. 6.38: Damage energy across Tungsten monoblock.

(continued from previous page)

```
Sim.Pipe = [{'Type':'smooth tube', 'Diameter':0.012, 'Length':0.012}]
Sim.Coolant =[{'Temperature':100, 'Pressure':3.3, 'Velocity':10}]
```

The tallied damage energy values are converted to finite element mesh by means of Code_Aster script:file:*Scripts/Experiments/Irradiation/Sim/damage.comm*

The attributes of `Sim` used for the conversion of tallied damage energy values are converted to finite element mesh are:

```
Sim = Namespace()
Sim.Name=['irradiated_day1000']
Sim.AsterFile = ['damage']
Sim.Mesh = ['mono']
Sim.width_mesh=[50] # mesh size across width of tungsten monoblock
Sim.height_mesh=[50] # mesh size across height of tungsten monoblock
Sim.thic_mesh=[25] # mesh size across thickness of tungsten monoblock
Sim.Pipe = [{'Type':'smooth tube', 'Diameter':0.012, 'Length':0.012}]
Sim.Coolant =[{'Temperature':100, 'Pressure':3.3, 'Velocity':10}]
```

The damage energy across the monoblock obtained from the neutronics simulation is employed to calculate the displacement per atom (dpa) at the various stages of the operation as a function of days in fusion energy conditions. The script employed for converting damage energy to dpa is script:file:*Scripts/Experiments/Irradiation/DPA/dpa_calc.py*

The attributes of `DPA` used for the conversion of damage energy to dpa are:

```
DPA= Namespace()
DPA.Name=['irradiated_day1000']
DPA.Cluster_tu=[15] # Number of clusters for tungsten
DPA.Cluster_cu=[10] # Number of clusters for copper
DPA.Cluster_cucrzr=[10] # Number of clusters for cucrzr
DPA.fusion_power=[1.5e5] # fusion power in Watts
DPA.days=[0] # Number of days
DPA.File=[('dpa_calc','dpa_calculation')] # python code for converting damage energy to␣
↪dpa
DPA.Warmour_height_lower=[1.15]  # lower height of monoblock from origin
DPA.Warmour_height_upper=[1.15] # Upper height of monoblock from origin
DPA.Warmour_width=[2.3] # Width of monoblock
DPA.Warmour_thickness=[1.2] # Thickness of monoblock
DPA.width_mesh=[50] # Mesh size along the width used from neutronics simulation
DPA.height_mesh=[50] # Mesh size along the height used from neutronics simulation
DPA.thic_mesh=[25] # Mesh size along the thickness used from neutronics simulation
```

The dpa calculated serves as input for Dislocation dynamics simulation to calcuate the yield strength as a function of dpa and irradiation temperature.

The dpa values are mapped into FEA mesh such that the yield strength (f(dpa, temp)) and thermal conductivity (f(dpa, temp)) calculated from Dislocation dynamics simulation and homogenisation technique, respectively, are allocated to assigned dpa and temperature fields across the monoblock during the FEA simulation.

The script employed for mapping dpa values into FEA mesh is script:file:*Scripts/Experiments/Irradiation/Sim/dpa_post.comm*

The attributes of `Sim` used for the conversion of dpa to FEA mesh are:

```
# Inputs for plotting the dpa distribution across the monoblock

Sim = Namespace()
```

(continues on next page)

(continued from previous page)

```
Sim.Name=['irradiated_day1000']
Sim.AsterFile = ['dpa_post']
Sim.Mesh =['mono']
```

The dpa distribution across monoblock is depicted in Fig. 6.39

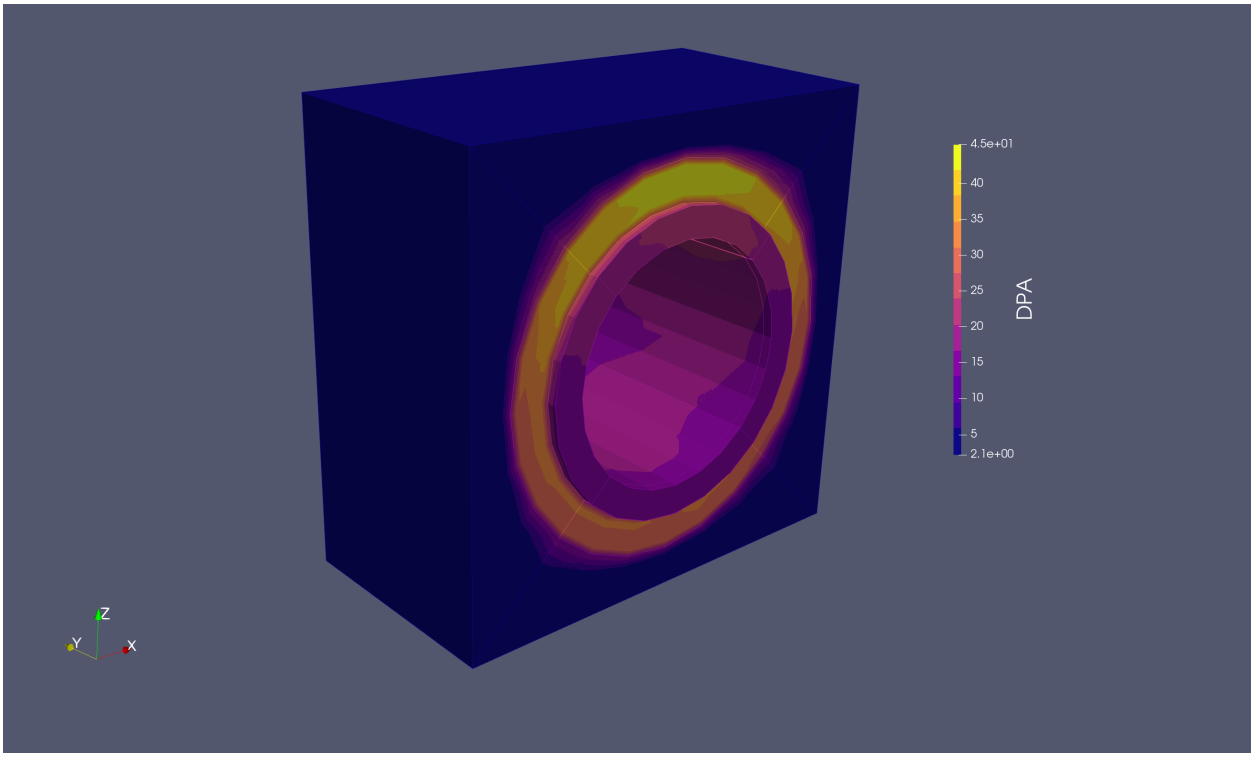


Fig. 6.39: DPA distribution across Tungsten monoblock.

6.7.4 Dislocation Dynamics simulation (D,E)

In neutron irradiated fusion relevant materials, it has been corroborated that there is an elevation in yield strength with respect to the pristine state. This is mainly because of the dislocation at the crystallographic slip planes interact with the irradiation-induced defects (dislocation loops, precipitates, voids, stacking fault tetrahedra) to cause irradiation-induced hardening. In fact, the dislocations are termed as plastic deformation carriers which interact with defects causing annihilation and rearrangement of dislocations resulting in the overall change in the microstructure with respect to the primary state of microstructure in pristine state. Dislocation Dynamics (DD) models are employed to analyse the irradiation-induced defect-dislocation interaction and understand the irradiation-induced hardening mechanism. Engineering properties such as yield strength can be calculated from DD models to design and conduct experiments on macro-scale component. In this current platform, DD model, MoDELib , is incorporated to understand the evolution of irradiation-induced microstructure through dislocation line and irradiation-induced interaction. MoDELib is developed based on phenomenological mobility law. DD simulations are carried in a Representative Volume Element (RVE) of the fusion reactor component materials containing irradiation-induced defect is loaded with uniaxial force in terms of strain rate at a specific irradiation temperature to calculate yield strength. The irradiation-induced defect information for RVE is represented in the form of density and geometric dimensions which are mainly obtained from experimental analysis and ab intio calculations .

In MoDELib, DD models are employed for fusion relevant materials such as iron (Fe), tungsten (W) and copper (Cu). Fig. 6.40 shows the RVE with irradiation-induced defects with log normal probability distribution which serves as the

computational domain for DD model. The density and size distribution of the irradiation-induced defects for DD model are identified based on the dpa values and thermal fields obtained from neutronics simulation (B) and FEA thermal simulation (C), respectively, which are obtained from literatures based on experimental and computational studies .

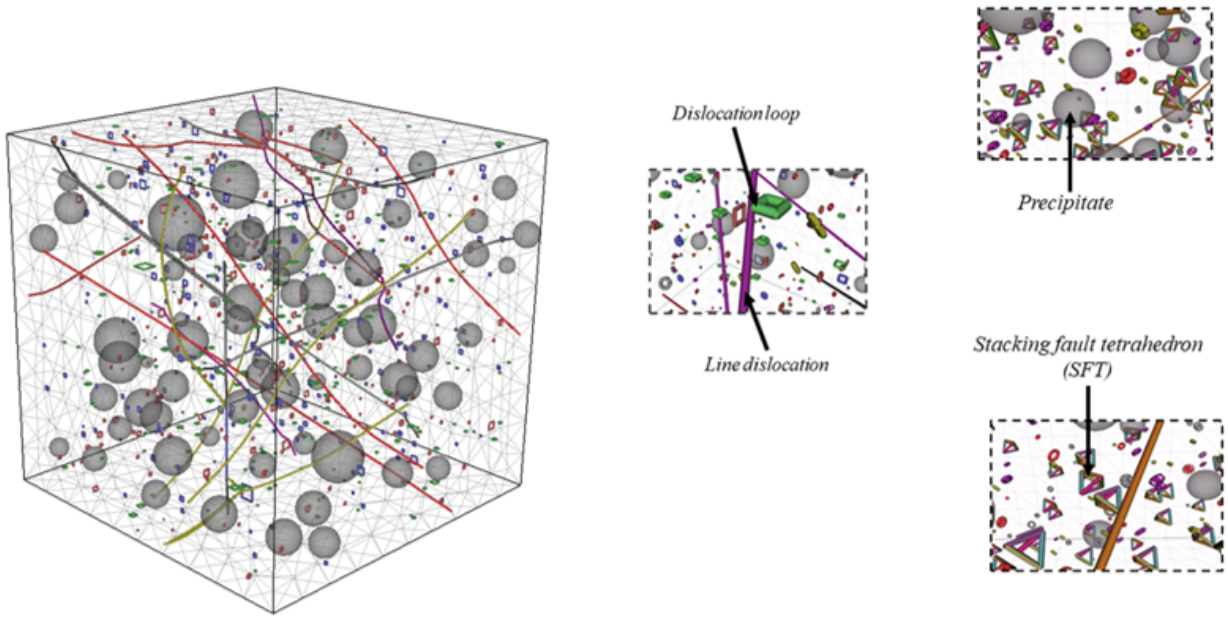


Fig. 6.40: RVE of tokamak component with irradiation-induced micro/nano structural defects.

`modelib` contains all the variables required by **modelib** software package to perfrom Dislocation Dynamics simulation:

```
modelib = Namespace()
modelib.Name = ['microstructure']
```

The Dislocation Dynamics simulation is peformed by means of python script:file:*Scripts/Experiments/DislDy/modelib/DDD.py*

The attributes of `modelib` used for the Dislocation Dynamics simulation are:

```
modelib = Namespace()
modelib.Name = 'microstructure'
modelib.File='DDD' #python file for performing DD simulation
modelib.dislocationline = 2e14 # density of dislocation line
modelib.dislocationloop = 1e22 # density of dislocation loop
modelib.prec=1e21 # density of precipitate
modelib.b=.1 # transformation strain for precipitate
modelib.dim=1 # scaling parameter of cubic RVE
modelib.temp=300 # Temperature
modelib.strainrate=1e-11 # uniaxial strain rate load on RVE
```

The execution of python script 'DDD.py` generates folders and files in:file:*Scripts/Output/DislDy/Tutorials/microstructure*

The folders and files generated are shown in Fig. 6.41

The results are stored in 'F' and 'evl' folders.

The defect density parameters are stored in 'inputfiles'.

After the Dislocation Dynamics simulation is completed, the strain-strain curve is plotted using python script:file:*Scripts/Experiments/DislDy/DPA/mechanical_load_results.py*

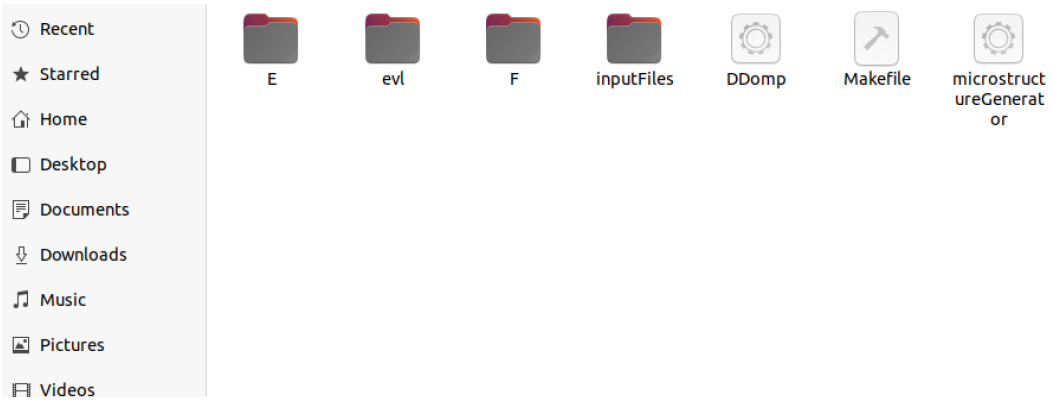


Fig. 6.41: Folders and files generated for Dislocation Dynamics simulation.

In order to calculate yield strength from the stress-strain data from Dislocation Dynamics simulation, the attributes of `DPA` are used:

```
DPA = Namespace()
DPA.Name = 'microstructure'
DPA.File=('mechanical_load_results','dpa_calculation')
```

The execution of python script ‘mechanical_load_results.py` calculates yield strengh and strain-strain plot in:file:*Scripts/Output/DislDy/Tutorials/microstructure*

The stress-strain plot generated from Dislocation Dynamics simulation are shown in Fig. 6.42

6.7.5 RVE thermal simulation (F)

Due to neutron irradiation, the defects produced induce change in the engineering properties of in-vessel components of tokamak reactor. In particular, the thermal fields of in-vessel components change at different stages of its lifecycle during operation due to irraidation induced change in thermal conductivity. In order to analyse the thermal conductivity of neutron irradiation materials, experimental measurements and computational models are employed. Due to the modularity of VirtualLab, various computional models can be employed to calculate thermo-physical property of fusion relevant components. In this current module, multi-scale homogenisation technique is employed to calculate effective thermal conductivity of irradiation-induced fusion relevant materials. The homogenisation method accommodates only three-dimensional defect types such as void, precipitates like Rhenium (Re) and Osmium (Os) which are produced in irradiated tungsten material.

In multi-scale homogenisation technique, RVE with irradiation-induced defects is assigned at the Gauss integration points of the macro-scale FEA thermal simulation (C). The thermal gradients at the Gauss integration point from the FEA thermal simulation (C) are used as the temperature boundary condition on the surface of the RVE containing irradiation-induced defects (F, Figure 1). The resultant homogenized heat flux and thermal conductivity obtained from RVE thermal simulation (F) are then transferred to the Gauss integration point of the macro-scale component for FEA thermal simulation (C).

In VirtualLab, ‘RVE’ module implements the RVE thermal simulation (F) for tungsten material which employs the temperature and thermal gradient from FEA thermal simulation (C) of tungsten armour as the linear thermal boundary conditions on the surface of RVE. The RVE is modelled with spherical inclusions which represents the irradation-induced precipitates such as Rhenium (Re) and Osmium (Os) in tungsten material. The computational domain of tungsten RVE with precipitates as spherical inclusions are generated from MoDELib.

As the first step, the generation of RVE with precipitates is carried out by means of ‘modelib’ module for the computational domain of RVE thermal simulation (F).

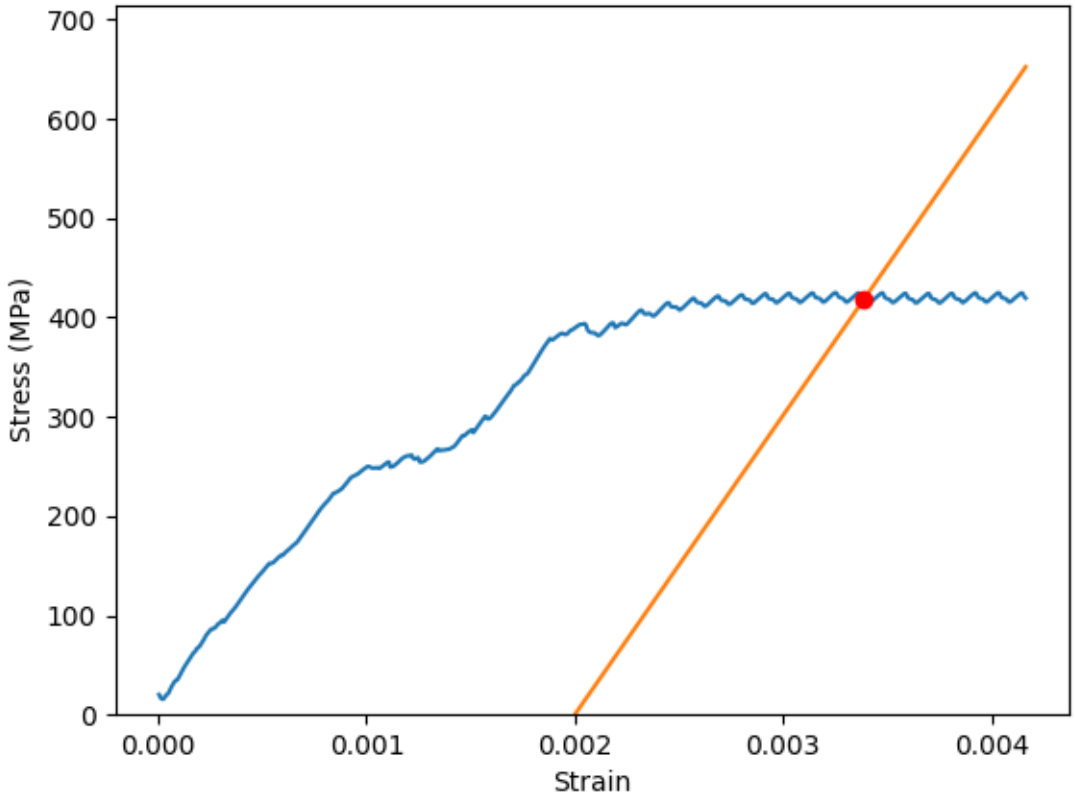


Fig. 6.42: Stress-strain generated from Dislocation Dynamics simulation (red denotes yield strength value).

`modelib` contains all the variables required by **modelib** software package to generate RVE with precipitates:

```
modelib = Namespace()
modelib.Name = ['microstructure0']
```

The generation of RVE with defects is peformed by means of python script:file:*Scripts/Experiments/RVE/modelib/DDD.py*

The attributes of `modelib` used are:

```
modelib.dislocationline = [0]
modelib.dislocationloop = [0]
modelib.prec=[1e21]
modelib.b=[.01]
modelib.dim=[15]
modelib.temp=[500]
modelib.strainrate=[1e-11]
```

The execution of python script 'DDD.py` generates folders and files in:file:*Scripts/Output/RVE/Tutorials/microstructure0*

In the 'E' folder, the coordinates of the precipitates are provided.

These coordinates and RVE box from the 'E' folder are used as the input for generating mesh of the RVE with precipitates for RVE thermal simulation (F).

The attributes of `DPA` used for the extracting the RVE with precipitate coordinates from 'E' folder using python script:file:*Scripts/Experiments/RVE/DPA/mesh.py*

```
DPA= Namespace()
DPA.Name = ['microstructure0']
DPA.File=['mesh']
```

'Rhenium.txt' and 'Osmium.txt' files are generated from the 'DPA' method.

`Mesh` contains all the variables required by **SALOME** to generate mesh for RVE with precipitate geometry.

```
Mesh.Name = 'RVE'
Mesh.File = 'RVE'
```

Mesh.File defines the script used by **SALOME** to generate the mesh, which in this case is `Scripts/Experiments/RVE/Mesh/RVE.py`.

Once the mesh is generated it will be saved to the sub-directory `Meshes` of the project directory as a `MED` file under the user specified name set in *Mesh.Name*. In this instance the mesh will be saved to `Output/Irradiation/Tutorials/Meshes/RVE.med`.

The attributes of `Mesh` used to create the sample geometry in `RVE.py` are:

```
Mesh = Namespace()
Mesh.Name = ['RVE']
Mesh.File = ['RVE']
dpa=[1]
e=len(dpa)
name=[]
for i in range(0,e):
    name.append('{}/RVE/Tutorials/'+ 'microstructure'+str(i)+'/Rhenium.txt')

name1=[]
for i in range(0,e):
```

(continues on next page)

(continued from previous page)

```
    name1.append(name[i].format(VLconfig.OutputDir))

Mesh.rve=name1

nameos=[]
for i in range(0,e):
   nameos.append('{}/RVE/Tutorials/'+ 'microstructure'+str(i)+'/Osmium.txt')

nameos1=[]
for i in range(0,e):
    nameos1.append(name[i].format(VLconfig.OutputDir))

Mesh.rveos=nameos1
```

The RVE mesh generated from *SALOME are shown in Fig. 6.43

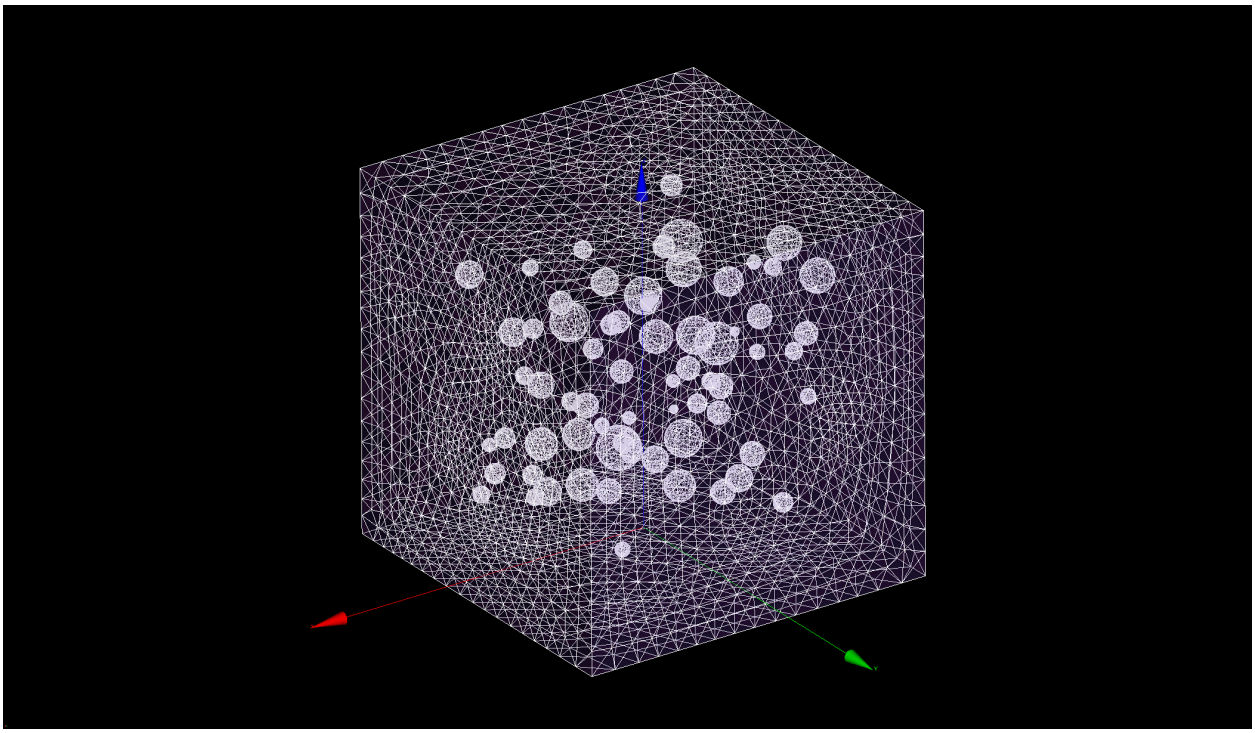


Fig. 6.43: RVE mesh with irradiation-induced defects.

The next step is to generate perfrom RVE thermal simulation by means of Code_Aster script:file:*Scripts/Experiments/RVE/Sim/RVE.comm*

The attributes of `Sim` used for the RVE thermal simulation (F) are:

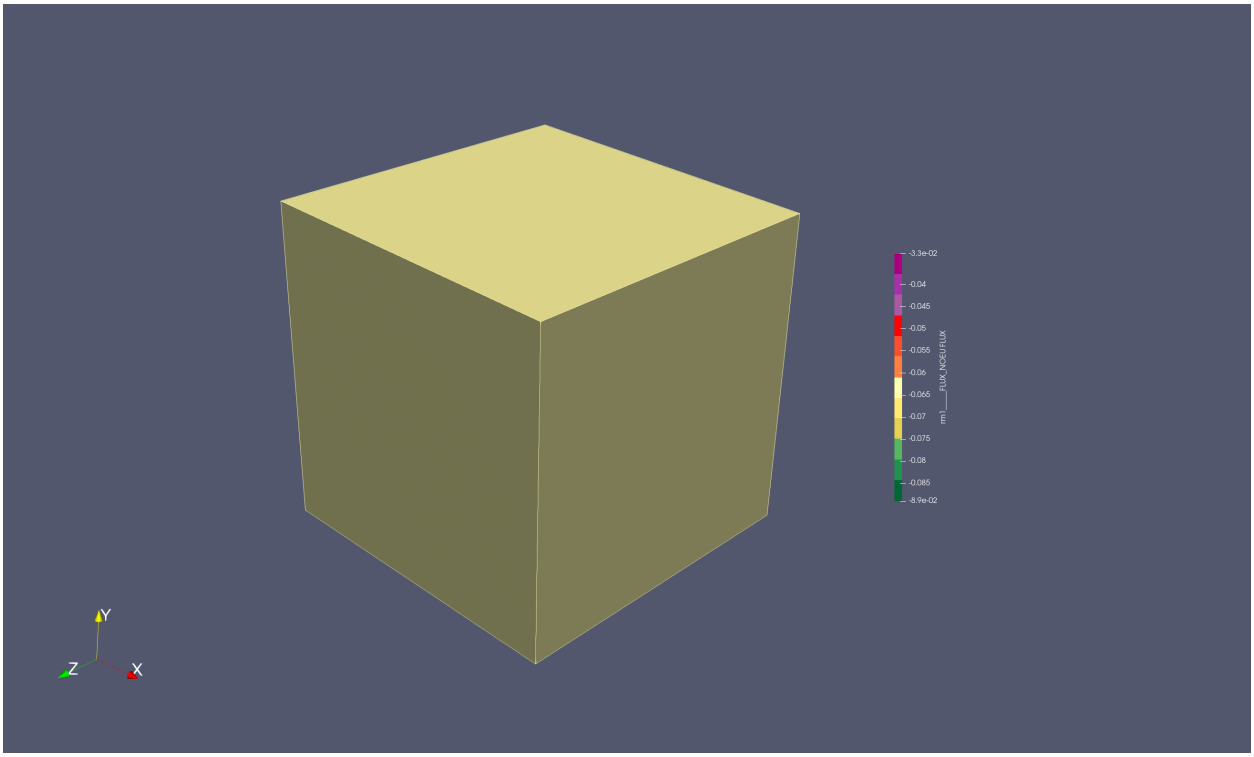
```
Sim = Namespace()
dpa=[1]
e=len(dpa)
name=[]
for i in range(0,e):
    name.append('microstructure'+str(i))
```

(continues on next page)

(continued from previous page)

```
Sim.Name = name
Sim.AsterFile = ['RVE']
Sim.Mesh = ['RVE']
Sim.dpa=[1] # dpa value
Sim.temp_gradientx=[.38] # Thermal gradient in 'x' direction from FEA thermal simulation
Sim.temp_gradienty=[.38] # Thermal gradient in 'y' direction from FEA thermal simulation
Sim.temp_gradientz=[.38] # Thermal gradient in 'z' direction from FEA thermal simulation
Sim.temp=[200] # Temperature at Gauss integration point from FEA thermal simulation
Sim.condTungsten=[.17] # Thermal conductivity of tungsten
Sim.condRhenium=[.039] # Thermal conductivity of rhenium
Sim.condOsmium=[.075] # Thermal conductivity of osmium
```

The heat flux of RVE from *Code_Aster* simulation as are shown in `Fig. %s`



6.7.6 FEA thermal simulation (C)

FEA thermal simulations (C) are performed for tungsten monoblock by imposing the following boundary conditions:

- Plasma heat load of 10 MW·m-2 is assigned at the top surface of monoblock.
- Neutron heating values from neutronics simulation (B) are imposed as volumetric heat source across the tungsten monoblock.
- Temperature dependant heat flux is derived based on the 1D modelling approach which is like that one employed for ITER cooling system.

The thermal properties such as thermal expansion coefficient, thermal conductivity and specific heat are obtained from literature. In particular, the thermal conductivity (f(dpa, temperature)) of tungsten armour are obtained from the

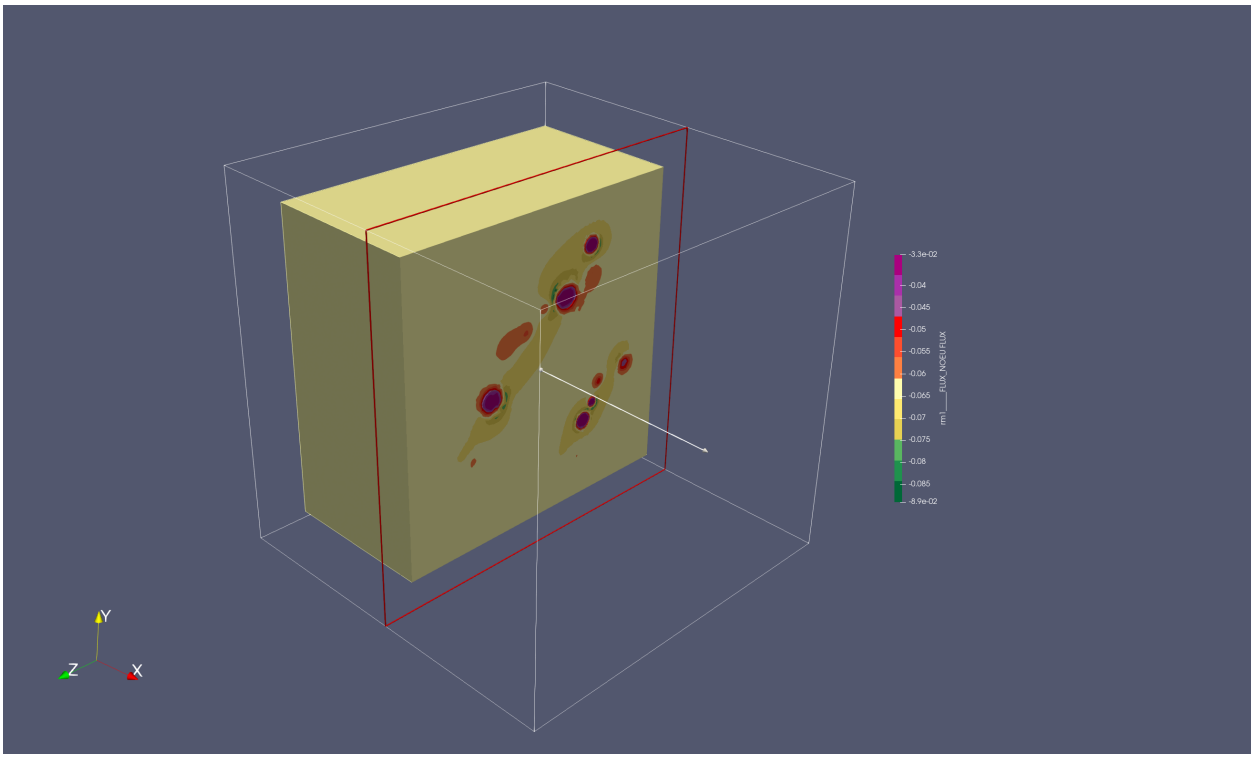


Fig. 6.44: Heat flux of RVE with irradiation-induced defects.

effective thermal conductivity derived from the RVE thermal simulations (F) modelled with irradiation-induced defects. While, the thermal conductivity (f(dpa, temperature)) of copper interlayer and CuCrZr coolant pipe are obtained both from literature and based on some assumptions.

The thermal boundary conditions imposed across the monoblock is depicted in Fig. 6.45.

The FEA thermal simulation is performed by means of Code_Aster script:file:*Scripts/Experiments/Irradiation/Sim/thermal.comm*

The attributes of `Sim` used for FEA thermal simulation are:

```
Sim = Namespace()
dpa=[0]
Sim.Name = ['irradiated_day1000']

# HTC between coolant and pipe (need Coolant and Pipe properties)
Sim.Pipe = [{'Type':'smooth tube', 'Diameter':0.012, 'Length':0.012}]
Sim.AsterFile =['thermal']
Sim.Mesh =['mono']
Sim.dpa=dpa
Sim.Coolant =[{'Temperature':150, 'Pressure':5, 'Velocity':10}]
```

FEA thermal simulation produces the following rmed files in `Output/Irradiation/Tutorials/irradiated_day1000/Aster/`

- thermal.rmed (thermal distribution across monoblock)
- thermacond.rmed (thermal conductivity distribution as a function of dpa and temperature across monoblock)
- yieldstrength.rmed (yield strength distribution as a function of dpa and temperature across monoblock)

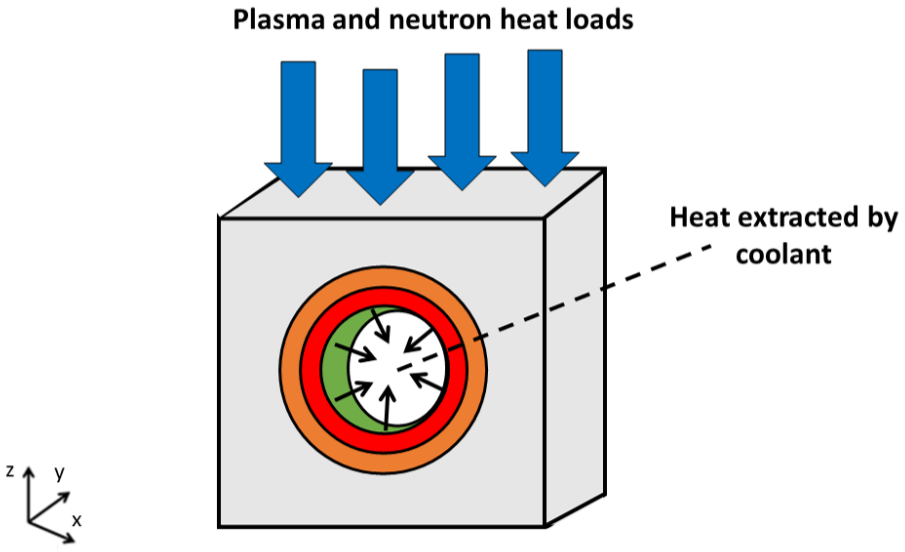


Fig. 6.45: Thermal boundary conditions of Tungsten monoblock.

- yieldstrength_cucrzr.rmed (yield strength distribution across CuCrZr pipe as a function of dpa and temperature across monoblock for lifecycle assessment (H) which will be discussed in the next sections)

The results from FEA thermal simulation and dpa distribution is depicted in Fig. 6.46.

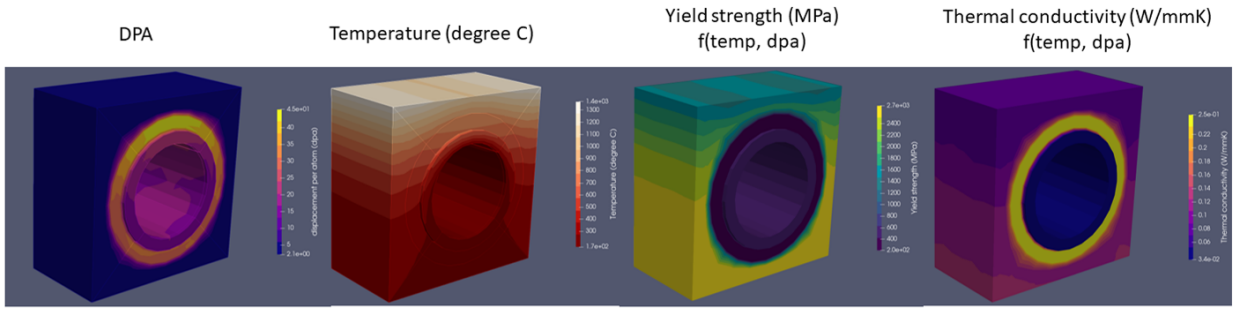


Fig. 6.46: DPA, thermal fields,yield strength and thermal conductivity distribution across Tungsten monoblock.

6.7.7 FEA Mechanical simulation (G)

FEA mechanical simulations (G) are performed for tungsten monoblock by imposing the following boundary conditions:

- Thermal stress obtained from FEA thermal simulation (C).
- Coolant pressure.
- Node constraints based on 3-2-1 method.

Elasto-plastic model is employed to perform FEA mechanical simulation. The yield strength for tungsten armour and copper interlayer is obtained from Dislocation Dynamics simulation (D,E). While the other mechanical properties are taken from the literature.

The mechanical boundary conditions imposed across the monoblock is depicted in Fig. 6.47.

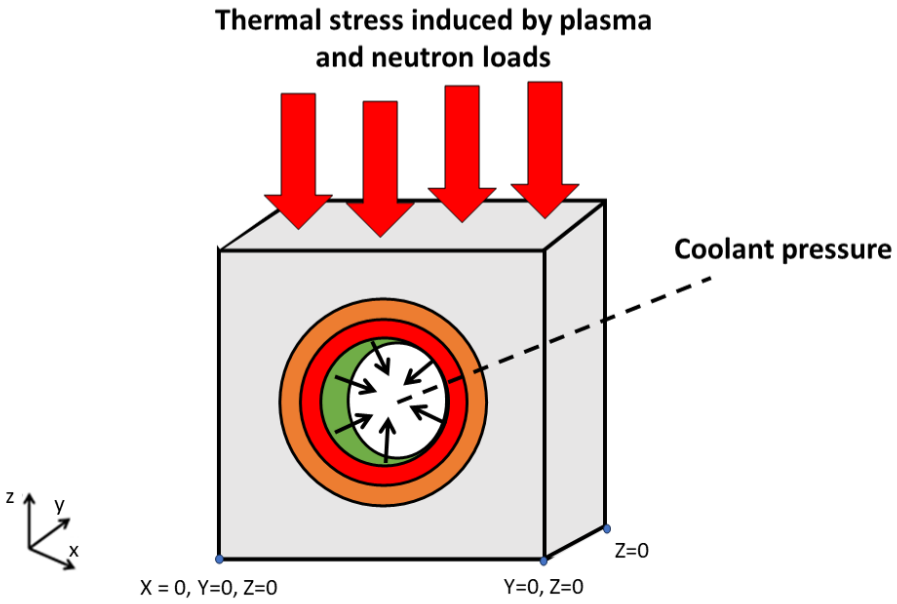


Fig. 6.47: Mechanical boundary conditions of Tungsten monoblock.

The FEA mechanical simulation is performed by means of Code_Aster script:file:*Scripts/Experiments/Irradiation/Sim/mechanical.comm*

The attributes of `Sim` used for FEA thermal simulation are:

```
Sim = Namespace()
dpa=[0]
Sim.Name = ['irradiated_day1000']

# HTC between coolant and pipe (need Coolant and Pipe properties)
Sim.Pipe = [{'Type':'smooth tube', 'Diameter':0.012, 'Length':0.012}]
Sim.AsterFile =['mechanical']
Sim.Mesh =['mono']
Sim.dpa=dpa
Sim.Coolant =[{'Temperature':150, 'Pressure':5, 'Velocity':10}]
```

FEA mechanical simulation produces the following rmed files in `Output/Irradiation/Tutorials/irradiated_day1000/Aster/`

- mechanical.rmed
- vmis.rmed (results for cucrzr pipe for lifecycle assessement (H))

The results from FEA mechanical and thermal simulation alongwith dpa distribution is depicted in Fig. 6.48.

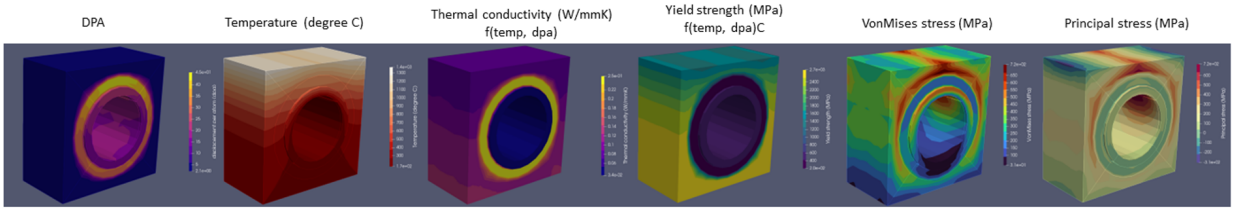


Fig. 6.48: DPA, thermal fields,thermal conductivity,yield strength,VonMises stress, Principal stress distribution across Tungsten monoblock.

6.7.8 Lifecycle assessment (H)

The in-vessel components in a fusion reactor are subjected to testing which must adhere to certain criteria in order to withstand extreme hostile conditions. The evaluation is based on the "Structural Design Criteria for In-vessel Components" (SDC-IC) design code for understanding the stress limits and analyse failure mechanism of the reactor components. In order to perform design code lifecycle assessment, the numerical results from the FEA thermal and mechanical simulation results are employed for elastic analysis procedure (EAP). In this platform, plastic flow localisation rule (SDC-IC 3121.2) is implemented to study the lifecycle assessment of in-vessel structural component, CuCrZr pipe, in both unirradiated and irradiated state. In the plastic flow localisation rule, the total stress (PL) (mechanical) + (QL) (secondary)) of the in-vessel component is tested against the yield stress (Se) which indicates the ductility limits of the material based on the equation:

PL+ QL Se (T,dpa)

T is the temperature. The total stress felt by a component comprises of both the primary (mechanical), PL, and secondary (thermal) stresses, QL, that are applied. If the total stress of the material exceeds its ductility, ductile failure occurs. This can be quantified by strength usage and reserve factor, Rf , this being the ratio of Se and the total stress applied. A value of Rf <1 indicates that ductile failure is likely to occur. The strength usage is calculated for three conditions:

- Maximum temperature
- Minimum temperature
- Maximum stress

As the first step, the 'RMED' files vmis.rmed and yieldstrength_cucrzr.rmed in `Output/Irradiation/Tutorials/irradiated_day1000/Aster/' are converted to .vtm format by means of python script:file:`Scripts/Experiments/Irradiation/DA/lifecycle_post_vtm`

The attributes of `DA` used for conversion are:

```
DA= Namespace()
DA.Name = ['irradiated_day1000']
DA.File=['lifecycle_post_vtm']
```

As the second step, lifecycle assessement method is employed using plastic flow localisation rule implementing the python script:file:*Scripts/Experiments/Irradiation/lifecycle/lifecycle_post*

The attributes of `lifecycle` used for performing lifecycle assessment are:

```
lifecycle= Namespace()
lifecycle.Name = ['irradiated_day1000']
lifecycle.File=['lifecycle_post']
```

As the third step, the results are plotted using python script:file:*Scripts/Experiments/Irradiation/DA/lifecycle_assess*

The attributes of DA used for results of lifecycle assessment are

```
DA = Namespace()
DA.Name = ['irradiated_day1000']
DA.File=['lifecycle_assess']
```

A plot 'lifecycle.png' is generated in Output/Irradiation/Tutorials/irradiated_day1000/Aster/ as depicted in Fig. 6.49.

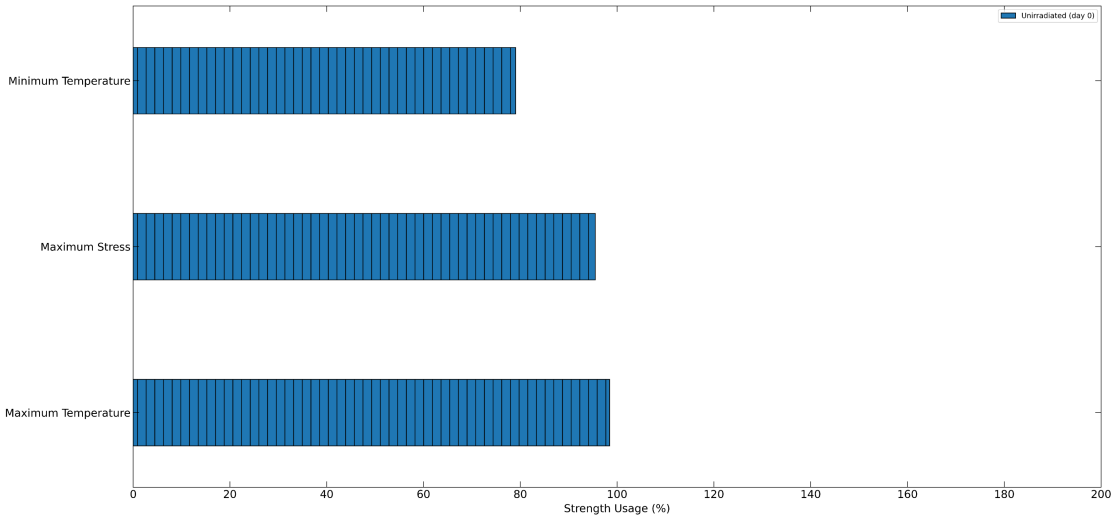


Fig. 6.49: Lifecycle assessment of CuCrZr pipe

6.8 Simulated X-ray imaging with GVXR

6.8.1 Introduction

X-ray imaging is a common to method used to to perform detailed analyses of the internal structure of an object in non-destructive way. VirtualLab allows users to create realistic simulations of X-Ray images using the software package GVXR. GVXR is a C++ x-ray simulation library developed by Frank Vidal and Iwan Mitchel at Bangor University.

It uses ray-tracing in OpenGL to track the path and attenuation of X-ray beams through a polygon mesh as they travel from an X-Ray source to a detector.

This tutorial will not cover the specifics of how to use GVXR, however training material on this in the form of jupiter notebooks can be found here.

The goal here instead is to show how GVXR can be run as a method inside a container within the VirtualLab workflow. As such we will cover similar examples to the training material but not the detailed theory behind them.

6.8.2 Prerequisites

The examples provided here are mostly self-contained. However, in order to understand this tutorial, at a minimum you will need to have completed the first tutorial to obtain a grounding in how **VirtualLab** is setup.

Also, although not strictly necessary, we also recommend completing the third tutorial because we will be using the **Salome** mesh generated from the HIVE analysis as part of one of the examples. All the previous tutorials (that is tutorials 2, 4 and 5) are useful but not required if your only interest is the X-Ray imaging features.

We also recommend you have at least some understanding of how to use GVXR as a standalone package and have looked through the GVXR training material as we will be working through very similar examples.

6.8.3 Example 1: First X-ray Simulations with GVXR

In this first example we will demonstrate how to simulate a single X-Ray image. We will start with a simple monochromatic point source and an object made from a single element.

Action

The *RunFile* `RunTutorials.py` should be set up as follows to run this simulation:

```
Simulation='Examples'
Project='Dragon'
Parameters_Master='TrainingParameters_GVXR-Draig'
Parameters_Var=None


#=============================================================================
# Environment

#=============================================================================

VirtualLab=VLSetup(
        Simulation,
        Project
        )

VirtualLab.Settings(
        Mode='Interactive',
        Launcher='Process',
        NbJobs=1
        )

VirtualLab.Parameters(
        Parameters_Master,
        Parameters_Var,
        RunCT_Scan=True,
        RunCT_Recon=False
        )

VirtualLab.CT_Scan()
```

A copy of this run file can also be found in `RunFiles/Tutorials/X-ray_imaging/Task1_Run.py`.

The mesh we be using For this example is the Welsh Dragon Model which was released by Bangor university, UK, for Eurographics 2011. The mesh, which can be found here, can be downloaded and placed in the directory `Output/Examples/Dragon/Meshes` by running the following command

```
VirtualLab -f RunFiles/Tutorials/X-ray_imaging/Task1_mesh.py
```

Action

Once you have the mesh downloaded you can launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

Because we have set `Mode='Interactive'` in `VirtualLab.Settings` you should see a 3D visualization of the dragon model in the path of the X-Ray beam casting a shadow onto the X-Ray detector behind, see Fig. 6.50.

You can use the mouse to zoom and rotate the scene to get a better view. Once finished you can close the window or type `q` on the keyboard.

Tip

To prevent this visualization from appearing in future runs simply set Mode to `'Headless'` or `'Terminal'`.

The X-ray image itself can be found in `Output/GVXR/Tutorials/GVXR_Images/Dragon/Dragon_1.tiff`, and should look like Fig. 6.51.

Looking though the *RunFile* The main thing to note is the call to `VirtualLab.CT_Scan()`. This is the function that initiates X-ray imaging using the parameters defined in *Parameters_Master* and *Parameters_Var*. Additionally, RunCT_Scan is explicitly set to `True` in `VirtualLab.Parameters`.

This isn't technically necessary because the inclusion of `VirtualLab.CT_Scan()` in the methods section means it is `True` by default, but explicitly stating this is good practice.

The parameters file we used is `Input/Examples/Dragon/TrainingParameters_GVXR-Draig.py` you will notice this file has a new Namespace `GVXR`. This contains the parameters used to setup and control the X-Ray Imaging. The file is setup with some sensible default values.

The GVXR Namespace contains a number of options many of which we will cover in later examples. For the curious a full list of these can be found in the appendix.

For ease of discussion of this first example we will break the required parameters down into four sections:

1. X-ray Beam parameters
2. Detector Parameters
3. Sample Parameters
4. Misc. Parameters

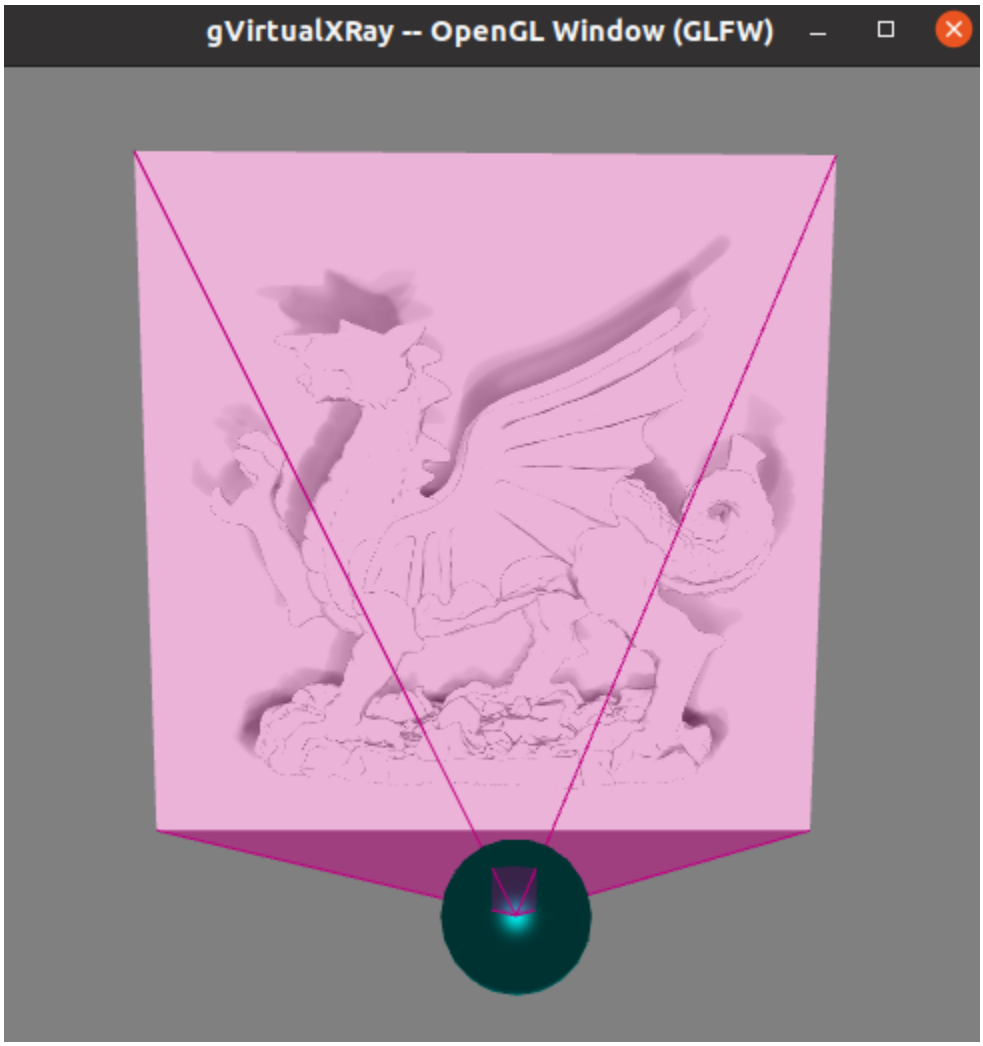


Fig. 6.50: Visualization of X-Ray imaging for Dragon model

Fig. 6.51: X-Ray Image of Dragon model.

Setting up the Beam:

Our first group of parameters concern the properties of the X-Ray Beam (source) GVXR needs to know 3 basic properties to define a source.

1. The position of the source
2. The beam shape
3. The beam energy (spectrum)

To set the position we use `GVXR.Beam_PosX`, `GVXR.Beam_PosY` and `GVXR.Beam_PosZ` the default units are mm. However, you can easily change this to essentially any metric units by setting `GVXR.Beam_Pos_units` to the appropriate string ("mm","cm","m" etc ...)[1]_.

For the beam shape we use `GVXR.Beam_Type`. GVXR allows for two choices:

- Cone beam: `GVXR.Beam_Type = 'point'`
- Parallel beam (e.g. synchrotron): `GVXR.Beam_Type = 'parallel'`

Finally we need to set the beam spectrum. Out of the box GVXR supports Monochromatic and PolyChromatic sources. You can also use the package xpecgen to generate more realistic/complex spectra, such as those from xray tubes. This will be covered in a later session. For now we will stick with a simple Monochromatic source.

This can be set with `GVXR.Energy`, this should be floating point (decimal) number, default units are MeV. The Intensity (taken as number of photons) is set with `GVXR.Intensity` this should be an integer (whole number). You can also optionally use `GVXR.energy_units` with a string to denote the energy units. This can be any of "eV", "keV" or "MeV" (take care with capitalization).

Tip

Setting up a simple monochromatic source can be easily done by passing in a list of numbers for energy and intensity. For example `GVXR.Energy = [50,100,150]` and `GVXR.Intensity = [500,1000,200]` will specify an X-ray source with 500, 1000, and 200 photons of 50,100 and 150 Mev respectively.

Action

Try changing the Beam energy from its current value of 0.08 Mev to 200 keV and observe what happens to the resulting image. you may also wish to try changing the beam from a cone beam to a parallel one.

Setting up the Detector:

Setting up the detector we need to specify its position, shape and physical size.

Similar to the beam to set the position we use `GVXR.Detect_PosX`, `GVXR.Detect_PosY` and `GVXR.Detect_PosZ` again the default units are mm. However, you can easily change this to essentially any metric units by setting `GVXR.Detect_Pos_units` to the appropriate string ("mm","cm","m" etc ...)[1]_.

For the number of pixels in each direction we use `GVXR.Pix_X` and `GVXR.Pix_Y`. Note: somewhat confusingly, up for the detector (i.e. Y) is along the Z axis in GVXR.

For the detector size we define the spacing between pixes with `GVXR.Spacing_X` and `GVXR.Spacing_Y` again the default units are mm but this can be changed with `GVXR.Spacing_units`.

Setting up the Sample:

Next we need to set the properties of the Sample in this case our dragon model

For our sample we need specify four things:

1. A 3D model of the object
2. What the Sample is made from
3. It's position
4. It's size
5. It's orientation

First we need to specify the name of mesh file used. This is done with `GVXR.mesh` This can be any mesh format supported by the python package meshio. You only need to specify the filename including file extension.

To set the position, much like the X-Ray beam we use `GVXR.Model_PosX`, `GVXR.Model_PosY` and `GVXR.Model_PosZ` in this case these define the center of the cad mesh in 3D space.

However unlike the beam position these are optional and if they are not given the mesh we be centered on the scene at the origin (that is [0,0,0]).

For units you have two parameters:

- `GVXR.Model_Pos_units` for the position
- `GVXR.Model_Mesh_units` for the mesh itself

The default units are mm. However, once again you can easily change this to essentially any metric units by using the appropriate string ("mm","cm","m" etc ...).

For scaling the mesh we have the optional values `GVXR.Model_ScaleX`, `GVXR.Model_ScaleY` and `GVXR.Model_ScaleZ`. These allow you to set a decimal scale factor in each dimension to reduce of increase the size of the model as needed. e.g. `GVXR.Model_ScaleX=1.5` will scale the model size by 1.5 times in the X direction.

We can also optionally set the initial rotation with `GVXR.rotation`. This is set as a list of 3 floating point numbers to specify the rotation in degrees about the X,Y and Z axes. The default is [0,0,0] (i.e. no rotation). This is useful if the model is not correctly aligned initially.

A note about Rotation

If you have used GVXR previously you will know that rotation can be a pain to deal with because of how OpenGL defines rotations (heres a link to good article for those interested souls). Sufficed to say I personally find rotations very quickly become unintuitive especially when dealing with multiple rotations and translations in sequence.

As such in VirtualLab rotations (both initial rotation and for CT scans) are defined in the simplest way I can think off. They are clockwise, centered on the mesh, are fixed to the scene (world) axes and are performed in the order X then Y then Z. (i.e. a `GVXR.rotation=[26.0,0,-15.3]` will perform a sequence of 2 rotations first 26 degrees clockwise about the X axis, then 15.3 degrees anti-clockwise about the Z axis).

If that makes no sense to you don't worry to much about it to much. If you are worried just leave it at the default [0,0,0] or play with the numbers until it looks right. Hopefully its intuitive enough.

Finally we need to set the material of the sample. For this we use three parameters:

- `GVXR.Material_list` a list of materials used.
- `GVXR.Amounts` a list of of relative amounts for each material, only used with mixtures.
- `GVXR.Density` a list of the densities in g/cm^3 for each material.

These are all lists of values to define the properties for each material used.

To actually define materials we use `GVXR.Material_list`. Each item in the list defines the material. In our case for the sake of simplicity we only have one mesh so we only need one value.

Using multiple materials

The current example uses a single mesh made from a single material. The step up to multiple materials however, is slightly more complicated. We will be covering a multi-material example in the example 3.

However, due to limited development time/resources. In the current version of VirtualLab the use of multiple materials is only supported by using mesh regions in salome .med mesh files. We do hope to add multi-materials for all mesh formats via the use of multiple meshes in the near future. However, for now this is a known limitation of the current version.

In GVXR materials are split into three types: elements, mixtures (alloys) and Compounds. To define an element we supply the English name, symbol or atomic number. So for a single mesh made from Chromium we can use any of `GVXR.Material_list = ['Chromium']`, `GVXR.Material_list = ['Cr']`, or `GVXR.Material_list = [24]`.

For a mixture we define a list of the elements in the mixture as atomic numbers (Note: names/symbols are not yet supported). You will also need to define the relative amounts of each using `GVXR.Amounts` with decimal values between 0.0 and 1.0 representing percentages from 0 to 100%. So for example a mixture of 25% Titanium and 75% Aluminum would be defined as: `GVXR.Material_list = [[22,13]]` and `GVXR.Amounts = [[0.25,0.75]]`

Compounds are defined as strings that represent the chemical formulae e.g. water would be `'H2O'` whilst Aluminum Oxide would be `'Al2O3'`. So for example a sample made from Silicon carbide would be defined as: `GVXR.Material_list = ['SiC']`.

For **both Compounds and Mixtures** you also will need to define the density for each material used, in g/cm^3. So for our previous example of Silicon carbide we can simply look up the density as 3.16 g/cm^3 thus we can use `GVXR.Density=[3.16]`

The density for the mixture of Titanium and Aluminum is more complex as there is no standard value so we need to approximate it. According to the royal society of chemistry Ti has a density of 4.506 g/cm^3 whilst Al is 2.70 g/cm^3. Thus for for our mixture using Vegard's law we get a approximate density of

$$\rho_{Ti_{0.25}Al_{0.75}} \approx \rho_{Ti} * 0.25 + \rho_{Al} * 0.75 = (0.25 * 4.506) + (0.75 * 2.70) = 3.152 g/cm^3$$

Thus `GVXR.Density=[3.152]`

Task

The default material for this example is Aluminum. Try changing this to something much more dense like tungsten (hint the chemical symbol for tungsten is W whilst its atomic mass is 74) and observe what the effect is on the resulting image. You could also try changing the sample to Aluminum oxide (which for reference has a density of 3.987 g/cm^3).

6.8.4 Example 2: Defining scans using a Nikon .xect files.

Many CT scanners use the Nikon .xect format to define scan parameters. These are just specially formatted text files ending in the .xect file extension. VirtualLab can read in parameters from these files.

To use these files you need to use `GVXR.Nikon_file` which sets the name of the nikon file you wish to use. This can either be in the Input directory or the absolute path to the file.

You will also at a minimum need to define

- `GVXR.Name`
- `GVXR.Mesh`
- `GVXR.Materail_list`

As well as possibly amounts and density depending on what materials you have specified. All other parameters are either optional or will be taken from the equivalent parameters in the nikon file.

Action

As an example we will perform the same simulation as the previous example only this time we will define the setup with a nikon file.

This will require changing *Parameters_Master* to 'TrainingParameters_GVXR_Nikon'

The *RunFile* `RunTutorials.py` should be setup as follows to run this simulation:

```
Simulation='Examples'
Project='Dragon'
Parameters_Master='TrainingParameters_GVXR_Nikon'
Parameters_Var=None

VirtualLab=VLSetup(
        Simulation,
        Project
        )
```

(continues on next page)

(continued from previous page)

```
VirtualLab.Settings(
        Mode='Interactive',
        Launcher='Process',
        NbJobs=1
        )

VirtualLab.Parameters(
        Parameters_Master,
        Parameters_Var,
        RunCT_Scan=True
        RunCT_Recon=False,
        )

VirtualLab.CT_Scan()
```

Launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

The resulting xray image will be saved to `Output/GVXR/Tutorials/GVXR_Images/Dragon_Nikon/Dragon_Nikon_1.tiff` and will be identical to Fig. 6.51.

The following is a table of parameters in the nikon file and there equivalent parameters in VirtualLab.

Table 6.1: Parameters used from Nikon files

Nikon Parameter	Notes	Equivalent Parameter	
Units	Units for position of all objects	GVXR.Beam_Pos_units, GVXR.Det_Pos_units, GVXR.Model_Pos_units	
Projections	Number of projections	GVXR.num_projections	
AngularStep	Angular step between images in degrees.	GVXR.angular_step	
DetectorPixelsX/Y	number of pixels along X/Y axis	GVXR.Pix_X/Pix_Y	
DetectorPixelSizeX/Y	Size of pixels in X and Y	GVXR.Spacing_X/Y	
SrcToObject	Distance in z from X-ray source to object, Note this is y in GVXR co-ordinates thus our beam position is defined as: [0,-SrcToObject,0]	GVXR.Beam_PosY	
SrcToDetector	Distance in z from source to center of detector. Again this is equivalent to y in GVXR. Thus Detect_PosY is defined as: SrcToDetector-SrcToObject	GVXR.Detect_PosY	
DetectorOffsetX/Y	detector offset from origin in X/Y	Detect_PosX/Z	
XraykV	Tube voltage in kV	GVXR.Tube_Voltage	
Filter_Material	Material used for beam filter	GVXR.Filter_Material	
Filter_ThicknessMM	Thickness of beam filter in mm	GVXR.Filter_ThicknessMM	

Please note however that a real nikon file will in general have a lot more parameters than these. As such any additional parameters defined in the file, along with comments in square brackets will simply be ignored.

Overriding values defined in a Nikon file.

You can define parameters in the input file that are also defined in the nikon file. If you do the parameters in the input file will override those in the nikon file.

6.8.5 Example 3: X-Ray CT-Scan with Multiple Materials

In this example we will Simulate a X-ray CT scan using the AMAZE mesh that was previously used for the HIVE analysis in tutorial 3.

Note: If you haven't completed tutorial 3 you will need to run the following command to generate the mesh

```
VirtualLab -f RunFiles/Tutorials/X-ray_imaging/Task3_mesh.py
```

Action

The *RunFile* `RunTutorials.py` should be setup as follows to run this simulation:

```
Simulation='HIVE'
Project='Tutorials'
Parameters_Master='TrainingParameters_GVXR'
Parameters_Var=None

VirtualLab=VLSetup(
        Simulation,
        Project
        )

VirtualLab.Settings(
        Mode='Interactive',
        Launcher='Process',
        NbJobs=1
        )

VirtualLab.Parameters(
        Parameters_Master,
        Parameters_Var,
        RunSim=False,
        RunCT_Scan=True,
        RunCT_Recon=False
        )

VirtualLab.CT_Scan()
```

A copy of this run file can be found in `RunFiles/Tutorials/X-ray_imaging/Task3_Run.py`

Looking at the file `Input/HIVE/Tutorials/TrainingParameters_GVXR.py` you will notice the Namespace `GVXR` has a few new options defined. Firstly, we are now using a more realistic beam spectrum instead of a monochromatic source. This is achieved by replacing `GVXR.Energy` with `GVXR.Tube_Voltage`. This tell VirtualLab to generate a beam spectrum from a simulated X-Ray Tube using xspecgen, in this case running at 440 KV. This is a more realistic X-Ray source than a simple monochromatic beam.

VirtualLab also has three other optional parameters related to X-Ray Tube spectrums which we are not used in this example.

- `GVXR.Tube_Angle` common setting used by X-ray tubes default is 12.0
- `GVXR.Filter_Material` material used for beam filter, used to remove certain frequencies
- `GVXR.Filter_ThicknessMM` Thickness of beam filter

The second change to note here is we are now using a mesh with multiple materials. As mentioned earlier this is only currently implemented for salome med meshes using mesh regions. In our case the mesh has 3 regions Pipe, Block, and Tile.

For GVXR we have to define the corresponding materials using `GVXR.Material_list` in this case the pipe and block are both made from Copper. whilst the tile is made from the much denser Tungsten.

Action

Launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

The x-ray image generated for this sample can be found in `Output/HIVE/Tutorials/GVXR-Images/AMAZE_single`, and should look like Fig. 6.52.

Fig. 6.52: X-Ray Image of the AMAZE component.

6.8.6 Appendix

Here is a complete list of all the available parameters that are used with GVXR alongside a brief explanation of there function. Note a default value of "-" indicates that this is a required parameter.

Table 6.2: Parameters in the GVXR Namespace

Parameter	Notes	Default Value
Name	Name of the simulation	–
mesh	Name of mesh file used	–
Beam_PosX	Position of beam in X	–[2]
Beam_PosY	Position of beam in Y	–[Page 135, 2]
Beam_PosZ	Position of beam in Z	–[Page 135, 2]
Beam_Pos_units	units for Beam position[1]	mm
Beam_Type	Type of Source used, can be either point or parallel	point
Energy	Energy of Beam	0.0
Intensity	Number of Photons	0
Tube_Angle	Tube angle, if using spectrum calculation	12.0
Tube_Voltage	Tube Voltage, if using spectrum calculation	0.0
Filter_material	material for beam filter, optional parameter used in spectrum calculation.	None

continues on next page

Table 6.2 – continued from previous page

Parameter	Notes	Default Value
Filter_ThicknessMM	Beam filter thickness in mm, optional parameter used in spectrum calculation.	None
energy_units	Units for Energy can be any of 'eV' 'KeV', 'MeV'	Mev
Model_PosX	Position of center of the Cad Mesh in X	0.0[Page 135, 2]
Model_PosY	Position of center of the Cad Mesh in Y	0.0[Page 135, 2]
Model_PosZ	Position of center of the Cad Mesh in Z	0.0[Page 135, 2]
Model_ScaleX	CAD Model scaling factor. Used to scale the model if needed.	1.0
Model_ScaleY	CAD Model scaling factor. Used to scale the model if needed.	1.0
Model_ScaleZ	CAD Model scaling factor. Used to scale the model if needed.	1.0
rotation	Initial rotation, in deg of Cad Model about X,Y and Z axis. Useful if the cad model is not aligned how you would like.	[0.0,0.0,0.0]
Model_Pos_units	units for Cad Mesh position[Page 135, 1]	mm
Model_Mesh_units	units for Mesh itself[Page 135, 1]	mm
Detect_PosX	Position of X-Ray detector in X	–[Page 135, 2]
Detect_PosY	Position of X-Ray detector in Y	–[Page 135, 2]
Detect_PosZ	Position of X-Ray detector in Z	–[Page 135, 2]
Detect_Pos_units	units for X-Ray detector position[Page 135, 1]	mm
Pix_X	Number of pixels for the X-Ray detector in X	–[Page 135, 2]
Pix_Y	Number of pixels for the X-Ray detector in Y	–[Page 135, 2]
SpacingX	distance between Pixels in X	0.5
SpacingY	distance between Pixels in Y	0.5
Spacing_units	units for Pixel spacing[Page 135, 1]	mm
Material_list	list of materials used for each mesh or sub-mesh. See materials section for detailed usage.	–
Amounts	relative amounts of each material used. Note values used here must add up to 1.0	None
Density	density of each material used in g/cm^3.	None
num_projections	Number of projections generated for X-Ray CT Scan	1[Page 135, 2]

continues on next page

Table 6.2 – continued from previous page

Parameter	Notes	Default Value
angular_step	Angular step in deg to rotate mesh between each projection, Note: rotation is about the Y-axis in GVXR co-ordinates	0[Page 135, 2]
use_tetra	Flag to tell GVXR you are using a volume mesh based on tetrahedrons. Not the default triangles. When this is set it tells GVXR to perform an extra step to extract just the mesh surface as triangle mesh. Note: whilst this is reasonably efficient it does add a small amount of overhead to the first run. However to mitigate this with multiple runs the new mesh is saved as '{filename}_triangles.{mesh_format}' and is automatically re-used in future runs.	False
fill_percent	This setting, along with fill_value is used for removing ring artifacts during CT reconstruction. It allows you to fill a given percentage of the pixels from the 4 edges of the image (Top, bottom, left and right) with a specific value fill_value. If fill_value is not specified then the value used is calculated automatically from the average of the image background.	0.0
fill_value	value used to fill pixels at the image edges, when using fill_percent.	None
Nikon_file	Name of or path to a Nikon parameter .xtekct file to read parameters from, see section on Nikon file for more detailed explanation.	None
image_format	This option allows you to select the image format for the final output. If it is omitted (or set to `None`) the output defaults to a series of tiff images. However, when this option is set the code outputs each projection in any format supported by Pillow (see the PILLOW docs for the full list). Simply specify the image format you require as a string, e.g., `GVXR.image_format='png'`.	Tiff
bitrate	bitrate used for output images. Can be 'int8'/'int16' for 8 and 16 bit greyscale or 'float32' for raw intensity values.	float32

6.9 Performing X-Ray CT Reconstruction

6.9.1 Introduction

a CT reconstruction involves the creation of a 3D image of a sample by mathematically "stitching together" a series of 2D X-ray images taken from many different angles around a sample. VirtualLab allows users to such reconstructions using a python package called the Core Imaging Library (CIL).

This tutorial will not cover the specifics of how to use CIL, however training material on this is provided by the CIL team in the form of jupiter notebooks, which can be found here:

Our goal instead is to show how CIL can be run as a method inside a container within the VirtualLab workflow. As such we will cover similar examples to the training material but not the detailed theory behind them.

6.9.2 Prerequisites

The examples provided here are mostly self-contained. However, in order to understand this tutorial, at a minimum you will need to have completed the first tutorial, to obtain a grounding in how **VirtualLab** is setup. You should also have completed the tutorial on X-ray imaging.

We also recommend you have at least some understanding of how to use CIL as a standalone package and have looked through the CIL Training material since, as previously mentioned we will not be coving the theory behind these examples in any great detail.

Note: I have not had time to thoroughly test this and there is nothing in the CIL docs that confirms this. However I believe CIL requires a dedicated GPU to run. On my laptop with only integrated graphics it crashes with very strange errors. I suspect this is due to a lack of Video Ram.

However the only other systems I have tested on both have beefy Nvidia GPUs so I can't confirm it's not just my machine.

The main takeaway is the container is setup for GPU compute and I have put in a crude check for a working Nvidia GPU (line 60 CT_Reconstruction.py).

Although it has been setup for Nvidia GPUs, the container should be GPU agonistic. This is because AMD and Intel GPUs use the mesa drivers which are part of the mainline linux kernel so should work with any container out of the box.

However I cannot confirm this as I have have no other cards to test with. Thus for now locking the production version to just Nvidia, given we know it works, seemed a sensible compromise.

---

[2] These values are not required when using a Nikon .xect file as their corresponding values will be read in from that. If they are defined when using a nikon file they will override the corresponding value in the Nikon file. See section on Nikon files for more details.

[1] Note for real space quantities units can be any off: "um", "micrometre", "micrometer", "mm", "millimetre", "millimeter", "cm", "centimetre", "centimeter", "dm", "decimetre", "decimeter", "m" "metre", "meter", "dam", "decametre", "decameter", "hm", "hectometre", "hectometer", "km", "kilometre" "kilometer"

6.9.3 Example 1: A simple CT-Reconstruction

In this example we will demonstrate how to simulate a 360 degree X-ray CT scan and reconstruct it using CIL. This is a continuation of example 3 from the Xray imaging tutorial, which used the AMAZE mesh that was previously used for the HIVE analysis in tutorial 3.

Action

The *RunFile* `RunTutorials.py` should be setup as follows to run this simulation:

```
Simulation='HIVE'
Project='Tutorials'
Parameters_Master='TrainingParameters_CIL_Ex1'
Parameters_Var=None

VirtualLab=VLSetup(
        Simulation,
        Project
        )

VirtualLab.Settings(
        Mode='Interactive',
        Launcher='Process',
        NbJobs=1
        )

VirtualLab.Parameters(
        Parameters_Master,
        Parameters_Var,
        RunCT_Scan=True,
        RunCT_Recon=True
        )

VirtualLab.CT_Scan()
VirtualLab.CT_Recon()
```

A copy of this run file can be found in `RunFiles/Tutorials/CT_Reconstruction/Task1_Run.py`

The main changes to note in *Runfile*, other than the change of input file is the addition is the call to `VirtualLab.CT_Recon()`. This is the method used to reconstruct the CT data.

Looking at the file `Input/HIVE/Tutorials/TrainingParameters_CIL_Ex1.py` you will notice that the only Namespace is `GVXR`. This is intentional as CIL shares the `GVXR` Namespace with GVXR. The reason for this simple convenience as CIL shares most of the same parameters as GVXR and although confusing at first it saves us doubling up on parameters.

Action

Launch **VirtualLab** using the following command:

```
VirtualLab -f RunFiles/RunTutorials.py
```

The raw x-ray tiff images generated by GVXR using the CT_Scan method can be found in `Output/HIVE/Tutorials/`

`GVXR-Images/AMAZE_360`. X-ray images of the component after being rotated by 90 and 180 degrees are shown in Fig. 6.53 6.54 respectively.

Fig. 6.53: CT scan after component rotated by 90 degrees.

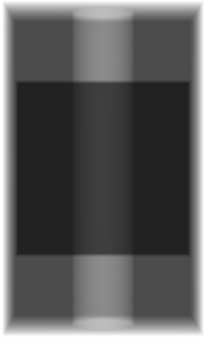


Fig. 6.54: CT scan after component rotated by 180 degrees.

The resulting reconstruction can be found as tiff images in `Output/HIVE/Tutorials/CIL_Images/AMAZE_360`, with each image representing a slice in Z. Tiff images of the component at slice 70 and 120 are shown in Fig. 6.55 6.56 respectively.

Note: To observe the reconstructed component the external package ImageJ will be required, which is currently not incorporated in **VirtualLab**.

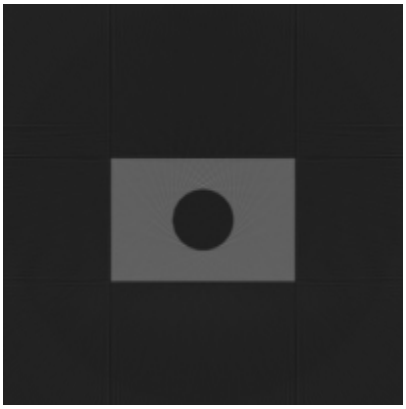


Fig. 6.55: Reconstructed component at slice 70 (of 250)

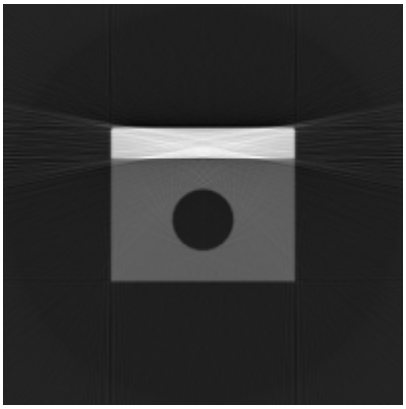


Fig. 6.56: reconstructed component at slice 120 (of 250)

6.9.4 Parameter's used by CIL:

The following parameters are used by both CIL and GVXR:

- `GVXR.Name`
- `GVXR.Beam_PosX/Y/Z`
- `GVXR.Beam_Type`
- `GVXR.Detect_PosX/Y/Z`
- `GVXR.Spacing_X/Y`
- `GVXR.Pix_X/Y`
- `GVXR.Model_PosX/Y/Z`
- `GVXR.Nikon_file`
- `GVXR.num_projections`
- `GVXR.angular_step`
- `GVXR.image_format`
- `GVXR.bitrate`

Units

Helpfully CIL is unit agnostic, that is CIL does not actually care what units you use to define the setup. The only thing that matters is that you are consistent. As such any definition of `GVXR.{OBJECT}_units` are entirely ignored by CIL as it does not need to know what they are.

Thus you can use any units you like (inches, furlongs, elephants) as long as they are consistent. That is if you use mm for the beam position you just need to ensure use mm for all other cases ie. model position, detector position and the pixel spacing.

Parameters that are unique to CIL

There is currently only one parameter that is unique to CIL `GVXR.Recon_Method` which can be either *"FBP"* or *"FDK"*. We will be using the default *FDK* for all our examples.

All these parameters work in exactly the same manner as GVXR as such they have already been explained in detail in the previous tutorial so I wont repeat myself here. However the parameters that are relevant to CIL are listed in the appendix.

The only slight exception is the default for `GVXR.image_format` is a single multi-page Tiff stack. I you would like individual tiff images for each slice in Z simply set `GVXR.image_format = 'Tiff'`.

CHAPTER

SEVEN

ADDING TO VIRTUALLAB

VirtualLab has been designed so that adding work is as easy as possible. There are four ways in which new work can be added:

1. New analysis scripts,
2. New methods of performing analysis,
3. New virtual experiments,
4. Containers with new software or code.

Description on how work can be added to these is discussed below, along with the best practice for adding your work to the **VirtualLab** repository.

7.1 Scripts

The easiest way to add to **VirtualLab** is by creating scripts for experiments and methods that already exist. For example, to create a new mesh script 'NewComponent' for the tensile experiment one would need to create the file `Scripts/Experiments/Tensile/Mesh/NewComponent.py` which describes the steps that **SALOME** must follow to create a CAD geometry and mesh. This file can then be used by specifying it as the 'file' attribute to the 'Mesh' namespace, e.g. Mesh.File = 'NewComponent' in the parameter file.

Similarly 'Sim' and 'DA' scripts can be created and placed in the relevant directories in the experiment directory.

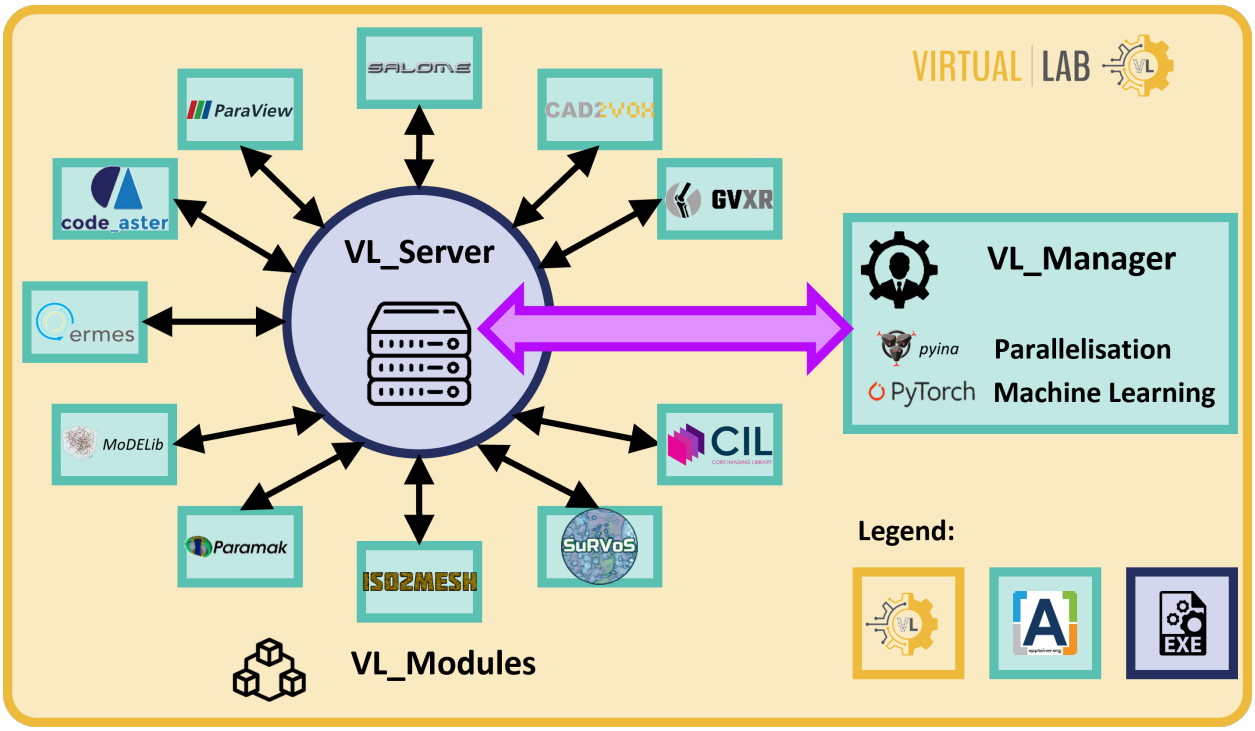
7.2 Experiments

Adding a new experiment to **VirtualLab** will require creating a new experiment directory `Scripts/Experiments/#ExpName`, where #ExpName is the name of the experiment. Within this experiment directory, sub-directories for the methods required to perform the analysis are needed. For example, if a mesh of a component is required using **SALOME**, then a directory named `Mesh` is required within the experiment directory which contains a relevant **SALOME** file.

A relevant directory would also need to be created within the input directory, i.e. `Input/#ExpName`. The parameter file(s) which passes information to the relevant methods and files used are to be included within this directory.

7.3 Containers and Methods

In **VirtualLab**, 'Containers' and 'Methods' are closely linked and are the heart of how **VirtualLab** can pull together many different pieces of software. The **VirtualLab** executable actually starts out as a tcp (networking) sever running on the host machine defined by the script `VL_server.py`. The server first spawns a manager container, **VL_Manager**, and passes in the RunFile. **VL_Manager** then executes the RunFile in a python environment. The RunFile itself begins by creating an instance of the VLSetup class. This then acts to spawn, control and co-ordinate all the other containers that will run the software to perform the actual task/analysis.



The Containers are spawned using various Methods which are functions assigned to the **VirtualLab** class to perform different types of tasks/analyses, e.g. Mesh and Sim. When a method is called by the **VL_Manager** a second container is spawned which is setup to perform the required task/analysis.

For example, a call to `VirtualLab.Mesh()` will spawn a container which has SalomeMeca installed inside. This will then run a script that will perform the actual task using the parameters supplied by **VL_Manager**. The full list of different methods can be found in the methods directory `Scripts/Methods`.

Each method file has a base class called 'Method' within it. These classes have a function called 'Setup' where information from the parameter file(s) are passed to build up the work to perform the task/analysis, e.g., the information attached to the namespace 'Mesh' in the parameter file(s) is available in the Setup function of the method 'Mesh'.

The 'Method' class must also have two other functions 'Spawn' and 'Run' which change how the method should behave when called, e.g., VirtualLab.Mesh(). The first function, 'Spawn', is selected when the method is called by the **VL_Manager** container. This is handled automatically in the base method class. 'Spawn', as the name suggests, configures a number of parameters and then communicates with the server on the host to spawn the container linked to the method and pass in the parameters for the task/analysis in question.

The second function, 'Run', is selected when the method is called within a container other than **VL_Manager**, again this is handled transparently. 'Run' is the function that will perform the required task/analysis with the supplied parameters.

Although not compulsory, these classes usually have a function called PoolRun which helps perform the tasks/analyses in parallel. For example, in the 'Mesh' method, the meshes are created using **SALOME** in the PoolRun function.

Placing the task in a separate function enables the use of **VirtualLab**'s parallelisation package. This allows multiple tasks/analyses to be performed in parallel using either the pathos (single intra-node) or pyina (multi inter-node) packages. Please see one of the available methods to understand how this is achieved.

Note: Any file in the methods directory starting with '_' will be ignored.

7.4 Amending Available Methods

Amendments can be made to the methods available by using the `config.py` file in the relevant methods directory. For example, due to the HIVE experiment being a multi-stage multi-physics experiment, 'Sim' needs to include a few additional steps. These are added in the file `Scripts/Experiments/HIVE/Sim/config.py`. There is a similar config file for the meshing routine of HIVE also.

7.5 Adding New Methods

To create a new method you will need a few things. Firstly, you will need a script to place in the methods directory. You may create a copy of the file `_Template.py` in the methods directory and save it as #MethodName.py, where #MethodName is the name of the new method type. Edit this file to perform the steps you desire. Not forgetting to edit the 'Spawn' function to associate your new method with a new or existing container. #MethodName will then be available to add information to in the parameter file(s) and to perform analysis using VirtualLab.#MethodName() in the run file.

Next, you will need a Container configured with the appropriate software to run your task/analysis. This can either be one of our existing containers, found in the Containers directory, or a custom one you have created (see adding new containers). You will also need to create both a bash and python script to start the container and perform the task/analysis respectively. We have templates for both of these in the bin and bin/python directories.

Finally, you will need to add your method to the config file `Config/VL_Modules.json`. Currently, this only requires one parameter, a namespace to associate with your method. This is the name that is used in the parameters file for **VirtualLab** and allows you to use a different name if you wish. For example, Cad2vox uses the method 'Voxelise' but the namespace 'Vox' because it's easier to type.

Note: Each method can only have a single namespace, however, namespaces do not need to be unique to particular methods.

Say for example you have several methods which share parameters, they can share the same namespace. This is the case for CIL and GVXR where they share the 'GVXR' namespace since they share many of the same parameters.

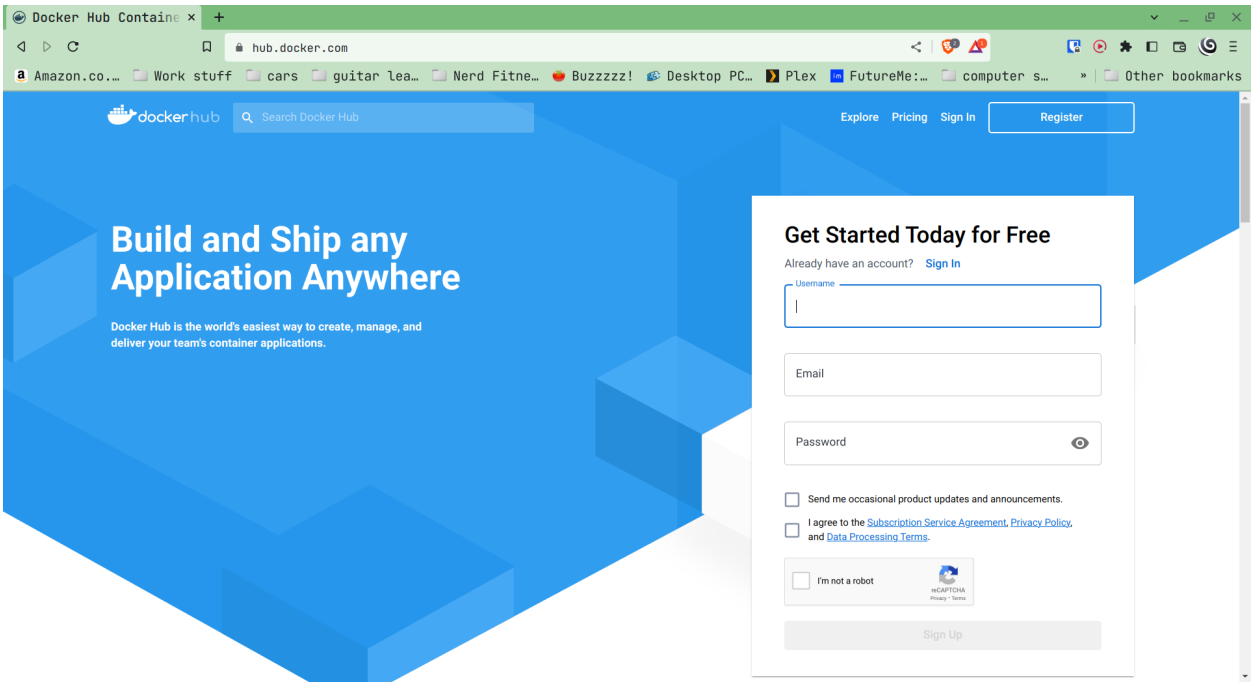
7.6 Adding New Containers

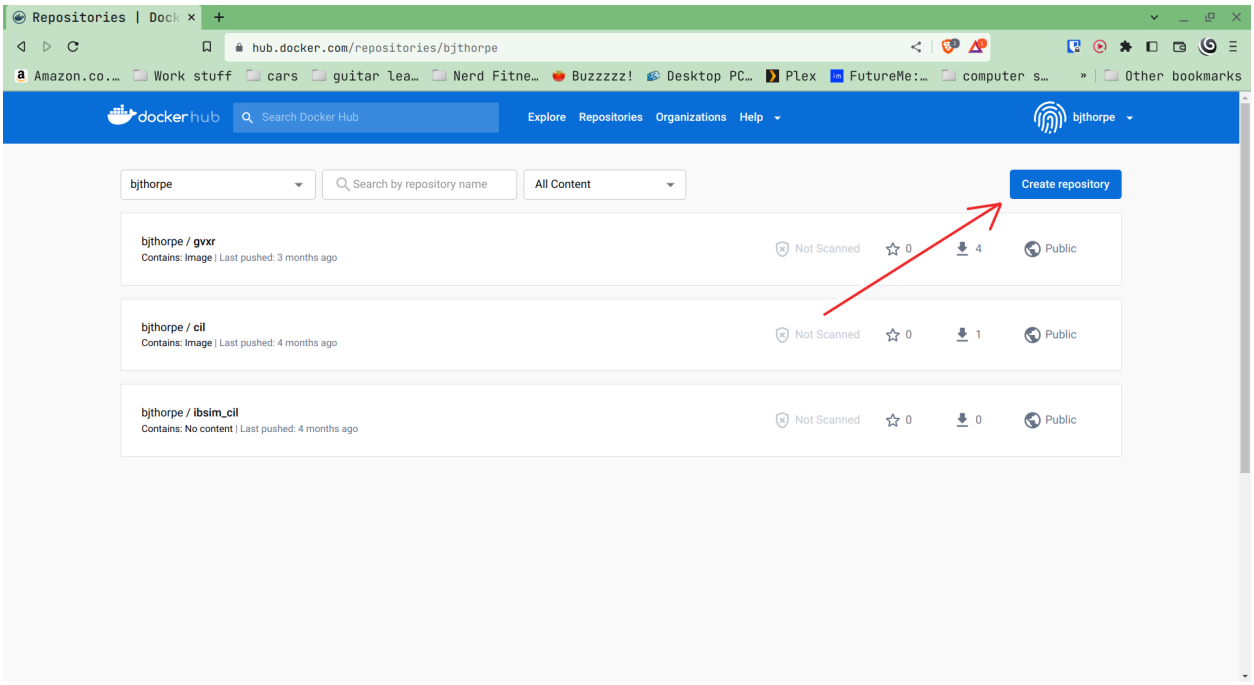
Our aim is that **VirtualLab** grows to accommodate applications that we might not have originally envisaged being part of the workflow. As such, our recommendation is that you contact us by raising an issue on gitlab with 'Type: Enhancement'. We will then be able to work with you to add or create a new Container such that others may also benefit from its inclusion. If you're keen to create a container which will only be used by yourself, then please follow these instructions.

To build new containers for **VirtualLab** you will first need to Install Docker. We use Docker for development of containers as opposed to Apptainer because Dockerhub provides a convenient way of hosting and updating containers

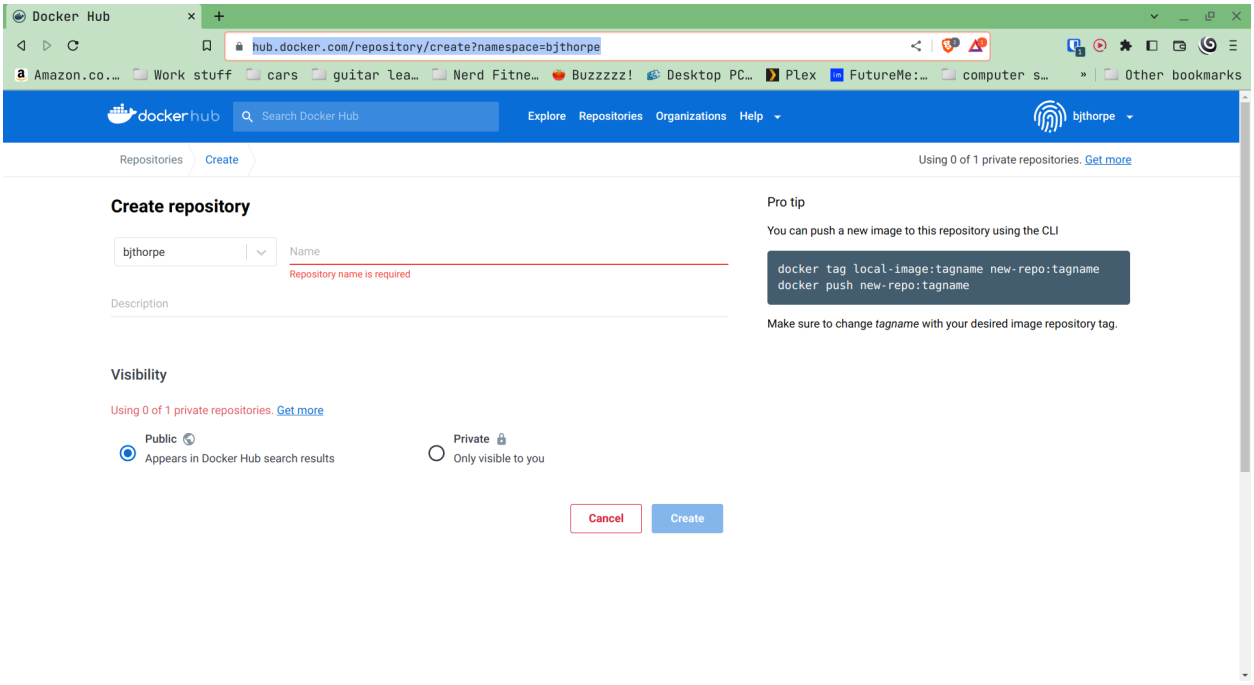
which Apptainer can pull from natively. The next step is to create your DockerFile configured with the software that you wish to use. We won't go into detail how to do this because it's out of the scope of this document. However, most popular software already have pre-made DockerFiles you can use as a starting point or failing that there are already plenty of *resources online <https://docs.docker.com/develop/develop-images/dockerfile_best-practices/>* `_ to get you started.

Once you have a DockerFile you will need to convert it to Apptainer. Annoyingly, Apptainer can't build directly from a local Docker file instead you need to point it to a repository on a docker registry. The easiest way to do this is to use DockerHub. You will first need to create an account. Once this is done you will need to log into the DockerHub website then click on the blue "Create Repository" button (see screenshots).





From there you will need to give your repository a name and decide if you want it to be public or private (Note: DockerHub only allows you have 1 private repository for free).



Once this is complete you will need to push your docker image to the repository which can be easily achieved at the command line.

First build your image locally, if you have not done so already. Replacing <image-name>, <tag-name> and <my_dockerfile> with whatever image name, tag and DockerFile you want to use.

```
Docker build -t <image-name>:<tag-name> -f <my_dockerfile>
```

Next, login to DockerHub with the account you created.

```
docker login
```

Next, we need to tag the image in a particular way to tell docker to point it to your repository. In this case <user-name> and <repo-name> are your username on DockerHub and the name of the repository you wish to push to.

```
docker tag <image-name>:<tag-name> <user-name>/<repo-name>:<tag-name>
```

Finally, we can push the image with:

```
docker push <user-name>/<repo-name>:<tag-name>
```

With that done we can finally convert our Docker image to Apptainer with the following command. Replacing <My-Container>.sif with whatever name you'd like to give the Apptainer sif file.

```
apptainer build <My_container>.sif docker://<user-name>/<repo-name>:<tag-name>
```

Using a local Docker Repository

Whilst DockerHub is free to use and a convenient solution it may not be the best solution for your situation. If privacy is your concern, you could use an alternative registry like singularity hub or even host your own.

However, suppose you are doing lots of testing and have a slow or limited internet connection. It's conceivable you may have to wait several minutes for uploading of your container to DockerHub only to re-download it through Apptainer. Fortunately, it is entirely possible to host a Docker registry on your local machine. Unfortunately, there are a number of caveats to consider:

1. It's quite fiddly and unintuitive to actually set up.
2. You are essentially doubling the amount of space needed to store docker images as you will have both a local and remote copy of the image to deal with.
3. You won't be able to share these images with anyone else as they will be local to your machine.

With those caveats in mind, if you are still undeterred a good set of instructions can be found here.

Now that we have an Apptainer file making it available as a module in **VirtualLab** is a fairly straightforward process. First, place the sif file in the Containers directory of **VirtualLab**. You will then need to edit the modules Config file `Config/VL_Modules.json` to make the container available as a **VirtualLab** module.

This file contains all the parameters to allow for the configuration of the various containers used by **VirtualLab**. The outer keys are the Module name used in the 'Spawn' method and the inner keys are the various parameters.

Note: A single Apptainer file can be associated to multiple Modules. This name is only used to identify how to setup the container when 'Spawn' is called by a particular method. Thus, you can use a single container for multiple different methods that share the same software. Each method will simply need its own bash and python scripts to tell the container what needs to be done.

The following keys are required to define a module:

- Docker_url: The name of the image on DockerHub (that is "docker://<user-name>/<repo-name>" you used earlier).
- Tag: The image tag, again <tag-name> from earlier. Do not include the semi-colon.
- Apptainer_file: Path to the sif file used for Apptainer.
- Startup_cmd: Command to run at container startup.

You also have the following optional keys:

- cmd_args: custom command line arguments, only useful if using your own scripts to start the container.

Using custom startup scripts and custom_args

The default arguments used by the template script are: '-m param_master -v param_var -s Simulation -p Project -I container_id'. If cmd_args is set it will override these. You can also set it to an empty string (i.e. "") to specify no arguments.

Ideally, we would like you to contribute your Container to the official IBSim repository on DockerHub. We keep all our DockerFiles in a separate git repository this is linked to DockerHub such that all we have to do is push our updated DockerFiles to that repo and it will automatically update and re-build the container on DockerHub. To do this please contact us by raising an issue on gitlab with 'Type: Enhancement'.

7.7 Contributing to VirtualLab

To submit changes to **VirtualLab** we use the following procedure. This allows us to collaborate effectively without treading on each others toes.

7.7.1 Branch Structure

The current setup for VirtualLab is as follows:

1. **Main:** Public facing branch, only changes made to this are direct merges from the dev branch.
2. **Dev:** Main branch for the development team to pull and work from. In general, we do not work directly on this branch, the only changes to this are direct merges from temporary branches.
3. **Temporary branches:** Branches for new features or work in progress and bug fixes.

Each developer should create a branch from **dev** when they want to create a new feature or bug fix. The branch name can be anything you like although preferably it should be descriptive of what the branch is for. Branch names should also be prepended with the lead developer's initials (to show who's leading the effort). Once the work is complete These branches can be merged back into **dev** with a merge request and then deleted.

Creating a new branch should be done roughly as follows:

```
# First ensure you are on the dev branch
git checkout dev
# Create a new branch with a name and your initials
git branch INITIALS_BRANCH-NAME
# Change onto the newly created branch
git checkout BRANCHNAME-INT
git push --set-upstream origin INITIALS_BRANCH-NAME
```

Now that we have a new temporary branch, development can continue on this branch as usual with commits happening when desired by the user. The temp branch can be also pushed to GitLab without creating a merge request if working with collaborators (and also for backing up work in the cloud). To do this the collaborator just needs to ensure they have all the latest changes from all the branches of the code from GitLab using `git pull --all` then change over to your branch using `git checkout INITIALS_BRANCH-NAME`.

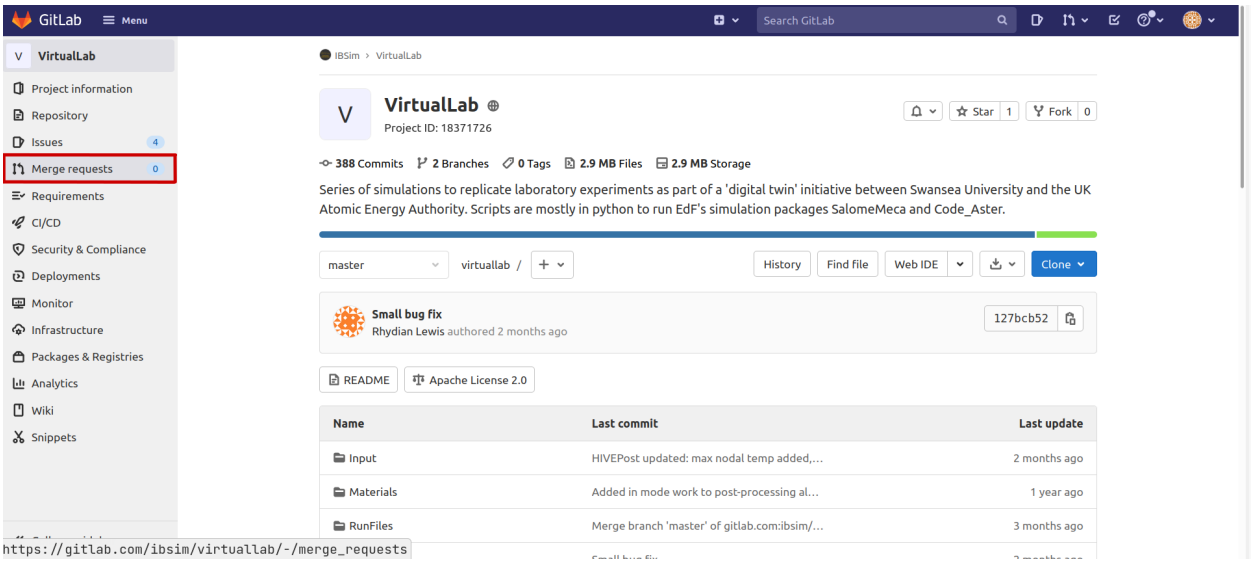
7.7.2 Creating a merge request

Once work on the temporary branch is complete and ready to be merged into the dev branch we need to first ensure we have pushed our changes over to the remote GitLab repo.:

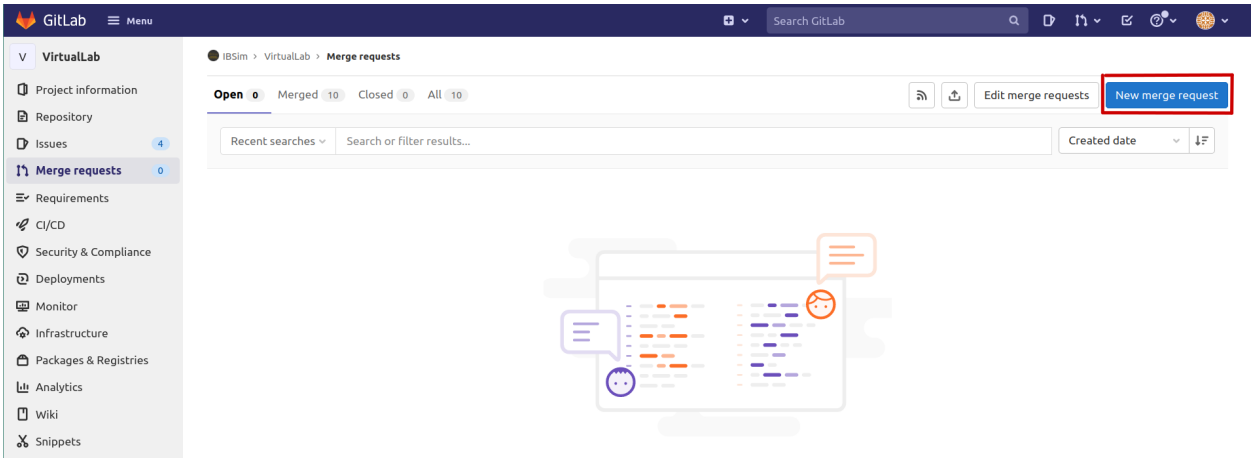
```
# First ensure we have the latest changes
git pull
# Push our changes to the GitLab repo
git push
```

Once this is complete we can go to the **VirtualLab** repo on gitlab.com and ensure we are logged into GitLab.

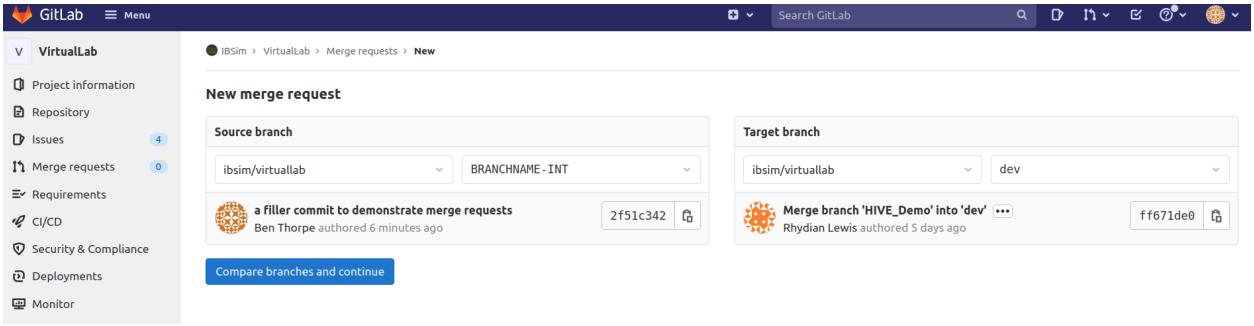
To create the request, from the left-hand side of the page click on "merge requests".



Then on the right-hand side of the next page click “New merge request”.



From here set the source branch as your temporary branch and the target branch as dev then click compare branches and continue.



The final step is to use the form to create the merge request:

- First give your merge request a title and a brief description of what features you have added or what changes have been made.
- For **Assignees** select “Assign to me”.
- For **Reviewers** select one of either Ben Thorpe or Rhydian Lewis.

- For **milestone** select no Milestone.
- For **Labels** select one if appropriate.
- For **Merge options** select "Delete source branch when merge request is accepted".

Once this is complete click "create merge request" this will then notify whoever you selected as reviewer to approve the merge.

7.7.3 Tidying up

Once the merge has been accepted, the final step is to pull in the latest changes to dev and delete your local copy of the temporary branch

```
# first ensure we have the latest changes
git checkout dev
git pull
# delete our local copy of the temporary branch
git branch -d INITIALS_BRANCH-NAME
```

CHAPTER

EIGHT

ABOUT

VirtualLab is the ouput of a collaboration between the Image-Based Simulation (IBSim) Group at Swansea University and UK Atomic Energy Authority. The majority of the initial work was carried out by Rhydian Lewis, during his PhD on *"Simulation driven machine-learning methods to optimise design of physical experiment and enhance data analysis for testing of fusion energy heat exchanger components"*, and Benjamin Thorpe, as Research Software Engineer on the EPSRC project *"Inline virtual qualification from 3D X-ray imaging for high-value manufacturing"* (EP/R012091/1).

8.1 Support

If you are having issues using **VirtualLab**, please let us know.

You may contact us by raising an issue on the project's gitlab repository.

8.2 Attribution

At the IBSim group, we are firm believers in open-access data and open-source code. Our main goal is that the output of our research will lead to wider use of virtual testing techniques, particularly for industrial applications. Therefore, access to the resources we develop is offered freely. This is with the aim that they will be used as part of your application.

The **VirtualLab** code is licensed under the Apache License, Version 2.0. That is, the user may distribute, remix, tweak, and build upon the licensed work, including for commercial purposes, as long as the original author is credited. We would also ask that you kindly inform us (resources@ibsim.co.uk) if the use of our resources has led to further output (e.g. industrial application, new software, journal publications). This enables us to evidence the value of our research when we apply for funding which will lead to further resource development for your benefit.

To cite this code in your publication please use the format below:

R. Lewis, Ll.M. Evans, B.J. Thorpe (2020) VirtualLab source code (Version !!!) [Source code]. https://gitlab.com/ibsim/virtuallab/-/commit/3c1d7727987def758df32a34933c964f54579325

NOTE: You will need to update the version number and url to the commit version you used.

8.3 Contribute

There are several ways to contribute to **VirtualLab**, all of which require a GitLab Account:

1. Use it and report issues, bugs and potential features that would benefit **VirtualLab**!

   https://gitlab.com/ibsim/virtuallab/-/issues

2. If you are a developer, the source code of **VirtualLab** is publicly available. Feel free to have a look at it, find and fix bugs, add new features or rework some areas of the tool and submit them as Pull Requests (PR).

   https://gitlab.com/ibsim/virtuallab/-/tree/master

Further instructions on developmental contributions are available here.

3. Similarly, this web-based documentation is also publicly available from the **VirtualLab** repository. New content for the documentation, correction typos or new tutorials in the form of PR are very welcome.

8.4 License