CS Capstone Design Document

March 17, 2020

Converting Fortran to Python

Prepared for

Genevieve Segol

Prepared by

Group40

Brian Hambleton
Dakota Alton
Ian Band
Ivan Halim

Abstract

This document describes the initial design choices for the development of a python based variably saturated flow and mass transport model. Design choices for user interface, development, and distribution are discussed herein. Specific design choices regard:

CONTENTS

1 Introduction

1.1 Purpose

This document serves as the outline for the application of research done by group40, with respect to the development of a python based variably saturated flow and mass transport model in two and three dimensions.

1.2 Scope

The scope of this document is to identify each design and development component for our application, along with predicting how it will be done.

1.3 Time Line

Below in fig. 1 the development time line for our application is outlined. In addition to what is below, every other week a test release of our application will be created and tested for quality assurance. House Keeping, in this context, refers to the initial set that must be done before development can be started.

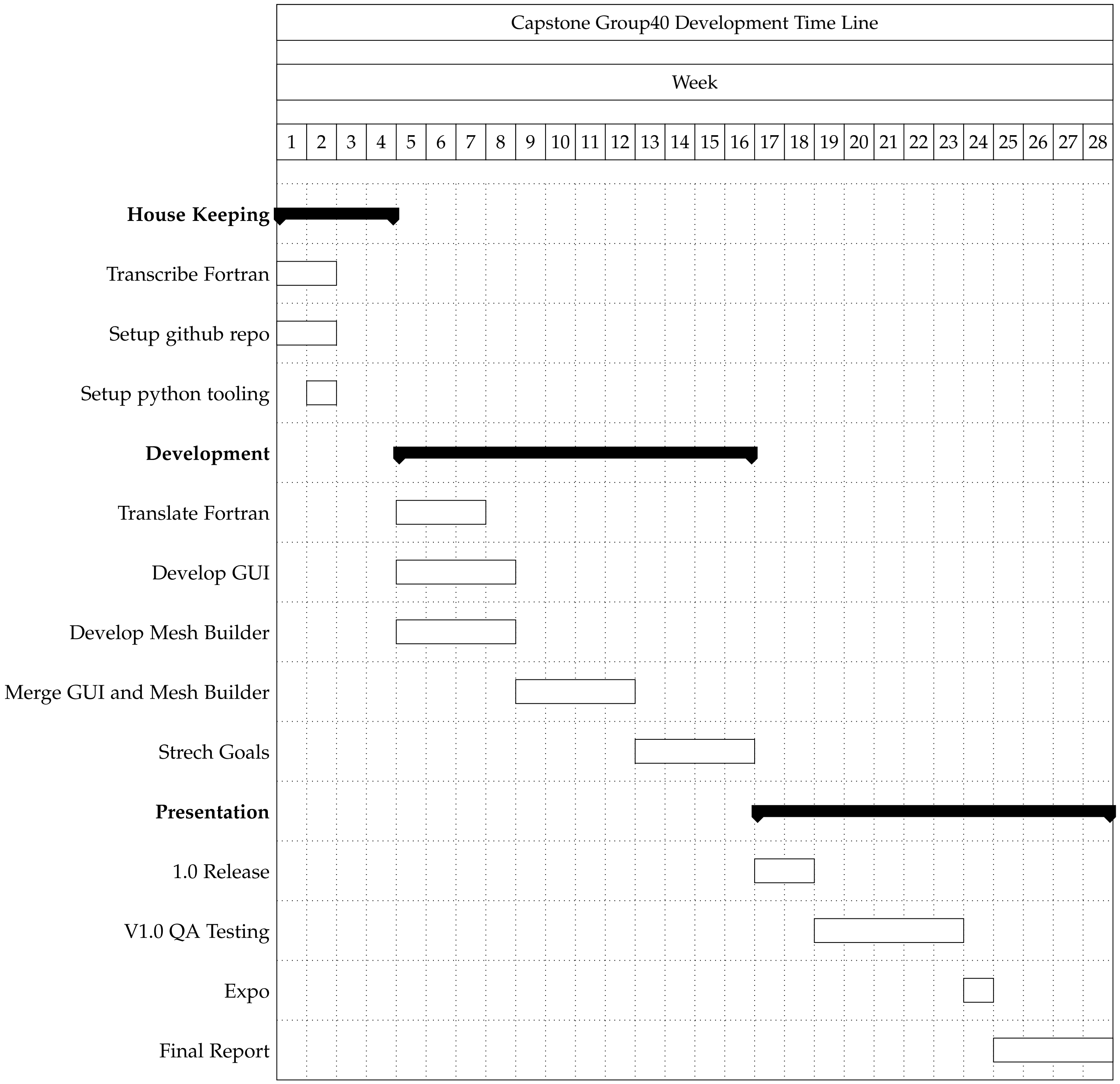


Fig. 1: Gantt chart representing development time line from the first week in December to June 15

1.4 Summary

Our application is meant to replace a legacy fortran version of the variably saturated flow and mass transport model. It will do this by converting the legacy code base to python and building a graphical user inter faces on top of it to better facilitate ease of use. Accomplishing these tasks will allow for two goals to be met. First, converting to python means that the application will be able to run on a wider array of devices. Second, adding a GUI to the application will improve user experience by making data input and output cleaner.

1.5 Revision History

2 References

3 Glossary

- *freezing* - In the context of Python, ”freezing” refers to bundling a program’s dependencies along with a python interpreter into an executable file.
- *pip* - This is a Python tool used for installing external python libraries.
- *requirements.txt* - This is a file that hold information on what external libraries are being consumed by our application.

4 Component Overview

Fig. 2 states who is, at a high level, responsible for which component. Transcribing and translating will be done by all of group40. The 3D Approximations component represents a stretch goal as such the owner will be assigned at a point in time when it is clear if the goal can be met.

Component	Owner
Code Transcribing	Ian Band
Code Translation	Ian Band
Code Reuse	Ian Band
3D Approximations	N\A
Environment Manager	Dakota Alton
Distribution	Dakota Alton
Automation	Dakota Alton
Finite Element Mesh Generator	Ivan Halim
File Format	Ivan Halim
3-Dimensional Mesh Plotter	Ivan Halim
PyQT GUI	Brian Hambleton
CLI	Brian Hambleton

Fig. 2: Components and associated owner

5 Simulation

5.1 Transcribing

The Fortran code provided by the client must be transcribed into text files that can be compiled.

5.1.1 Context

Source code was provided to our team in the form of a scanned PDF document. In order to use this code, it must be manually transcribed by our team into text files. Each subroutine in the source code will be transcribed in its own text file to make reuse of that routine more convenient. Manually transcribing the code will give our team familiarity with the codes functionality.

5.1.2 Set Up

Each team member will be assigned a number of subroutines to transcribe.

5.2 Code Translation

Functioning simulation code provided by the client will be translated from Fortran to Python 3.

5.2.1 Context

The simulation wrapper and interfaces between the simulation, data input, data output, and data visualization will all be written in Python 3.

5.2.2 Set Up

Developers will directly translate the main routine of each simulation. This mainly consists of calling the right Fortran subroutines from Python in the right order, and correctly passing arguments to each subroutine.

5.3 Code Reuse

5.3.1 Context

It is possible to reuse a large portion of the Fortran code base along side code written in Python. Doing so introduces design trade offs which are not discussed in depth in this section. In essence, reusing Fortran code alongside Python ensures an accurate, high performance, and easy to develop simulation will be produced. Time saved in this part of development allows for more time and resources to be spent on other parts of the project where code reuse is not an option.

5.3.2 Set Up

Developers on our team will use the F2PY library to create a signature file from the Fortran source that is usable within Python code. Developers will create signature files by running:

```
$ python −m numpy.f2py −c <Fortran source file> −m <package name>
```

5.4 2D and 3D Approximations

5.4.1 Context

The original code base consists of two related, but independent simulation tools. Our team will aim to recreate the 2D simulation before moving onto the 3D simulation as a stretch goal. In this way, we can use what we learn from creating the 2D simulation to help us with the 3D simulation.

6 DEVELOPMENT & DEPLOYMENT

6.1 Environment Manager

For this project we will be using the tool virtualenv to manage our applications external packages.

6.1.1 Context

The main point of having an environment manager is to be able to develop on any computer while maintaining a consistent set of external libraries. This is a necessary component of our design for we will be developing on different operating systems. Just as well, our freezing tool of choice is meant to run on the operating system that our application will run on, so keeping a consistent environment will mitigate any issues concerning building on any operating system.

6.1.2 Set Up

Setting up virtualenv is something that each developer will have to do. They will set virtualenv up by cloning the source repository and running the following command at the root of the repository:

```
$ python3 -m venv env
```

This will create a directory called ”env” that represents a virtual environment for managing python. At this point the workflow outlined in the following section must be adhered to.

Note that the directory created should not be placed into source control.

6.1.3 Workflow

1) enable the virtual environment with:

```
$ source env/bin/activate
```

2) program and install packages as one normally would.
3) deactivate the virtual environment with:

```
$ deactivate
```

Note that when a package is installed with pip, one musts run *$ pip freeze > requirements.txt*, to make sure that the library is known in source control.

6.2 Distribution

To build an executable of our application, we have chosen to use the tool PyInstaller.

6.2.1 Context

Utility of an application is predicated on the ability to run the application. Instead of requiring users to install a python interpreter to run our application, it would be more convenient to download an executable version of our code. To do this, our code base must be ”frozen” that is, all dependencies must be bundled into a sole executable file. Furthermore, our code must be usable on Windows and Mac so configuring PyInstaller to build an executable must be done to allow building Windows and Mac.

6.2.2 Set Up

PyInstaller is a package that can be installed with pip. It will be added to the projects requirements.txt just after setting up the git hub repository. PyInstaller can then be run on the main file of our application to build an executable.

For actually distributing our application we will tag branches in git for our current release version and add the appropriate executable files there.

6.3 Automation

To automate the testing our application we will be using the continuous integration (CI) service provided by TravisCI.

6.3.1 Context

Group development is best done when there is an identifiable standard to meet. Requiring that each developer run tests on their local machines is fickle; one's system, despite preventative measures, can be slightly different from another person's system leading to a false sense of quality. This is the primary case for using an independent CI service.

6.3.2 Set Up

TravisCI is will be set up by linking our git hub repository to our TravisCI account. Then we just add a travis.yml that specifies our version of python and which commands that must be run for our tests to run. An example travis.yml file would be:

```
language: python
python:
  - "3.3"
install:
  - pip install -r requirements.txt
script: pytest
```

Fig. 3: Example travis.yml file

7 Visualization

7.1 Finite Element Mesh Generator

To generate our mesh, we have chosen PyVista.

7.1.1 Context

A mesh generator is a very important component to any finite element solver. The purpose of a mesh generator is to create complex geometries and then discretize it into small simple shapes, such as triangles or quadrilaterals. The purpose of the discretization is to allow us to perform finite element analysis on complicated shapes, because we know how to perform calculations on a triangle but not directly on complicated spaces and shapes such as a roadway bridge.

7.1.2 Dependencies

The following modules are the required dependencies of PyVista:

- vtk - PyVista is built on top of the VTK library
- numpy - Provides direct data access for meshes
- imageio - Used for saving screenshots
- appdirs - Data management for example datasets.

7.1.3 Set Up

To install PyVista simply type on the command line,

```
$ pip install pyvista
```

7.1.4 Mesh Generation

A mesh is a geometrical representation of a 2D surface or 3D volume. A mesh consists of **nodes**, **edges** and **elements**.

- **Node:** The XYZ coordinates of the underlying structure
- **Edge:** The connectivity between nodes
- **Element:** The area or volume bounded by the edges.

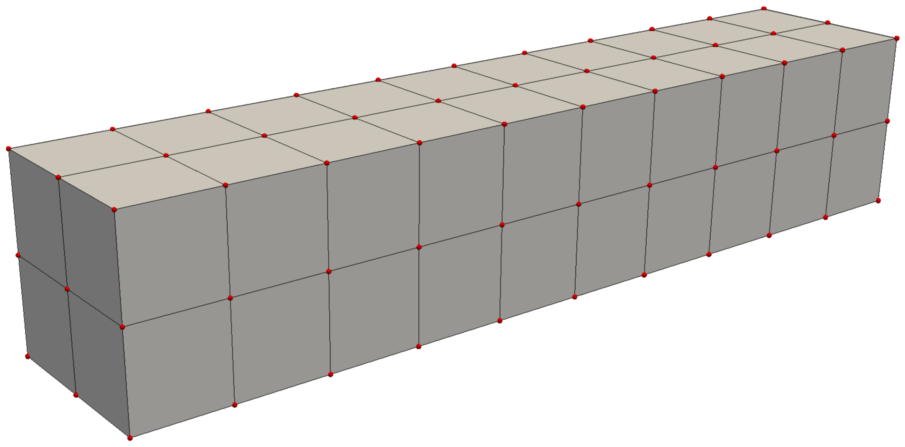


Fig. 4: An example mesh.

To create a mesh in PyVista, we will be using the PolyData object. The PolyData object accepts NumPy arrays of vertices and faces. The vertex array contains the XYZ coordinates of each point in the mesh and the face array contains the number of points of each face and the indices of the vertices which comprise that face.

```
# mesh points
vertices = np.array([[0, 0, 0],
                     [1, 0, 0],
                     [1, 1, 0],
                     [0, 1, 0],
                     [0.5, 0.5, -1]])

# mesh faces
faces = np.hstack([[4, 0, 1, 2, 3],  # square
                   [3, 0, 1, 4],     # triangle
                   [3, 1, 2, 4]])    # triangle

mesh = pv.PolyData(vertices, faces)
```

7.2 File Format

For the representation of mesh, we have chosen the VTK XML file format.

7.2.1 Context

Before we visualize our mesh, we have to make sure that the data produced by the mesh generator is in a file format that the plotter can accept. Not all file formats are created equal. Some formats are more space-efficient or support more features such as parallel reading and rendering for large datasets. For this reason, we have chosen the VTK XML file format due to its flexibility.

7.2.2 Dataset Types

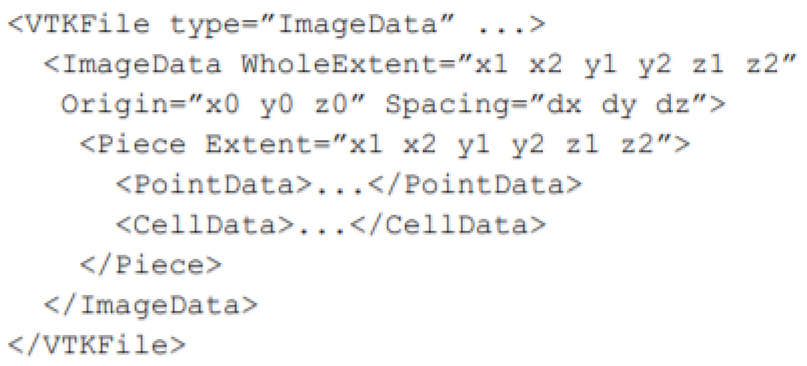
VTK can handle several kinds of datasets, for points data and cells data. The primary are:

- **ImageData:** Regularly spaced points data.
- **RectilinearGrid:** Regularly spaced points data but spacing can be not uniform.
- **StructuredGrid:** Not regular and not uniform points of data.
- **UnstructuredGrid:** Not regular and not uniform points data, but can handle all cell types.
- **PolyData:** Can be used for any polygonal data.

7.2.3 General Structure

The general structure for each dataset format is as follows:

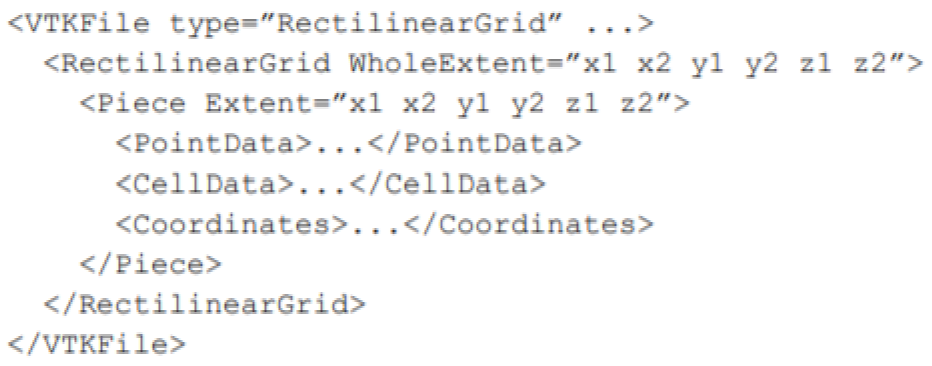
ImageData: For ImageData you only have to set the mesh dimensions (nx, ny, nz), mesh origin (x0, y0, z0) and cell dimensions (dx, dy, dz). Thus, points data are regularly and uniformly spaced.



```
<VTKFile type=”ImageData” ...>
  <ImageData WholeExtent=”x1 x2 y1 y2 z1 z2”
   Origin=”x0 y0 z0” Spacing=”dx dy dz”>
    <Piece Extent=”x1 x2 y1 y2 z1 z2”>
      <PointData>...</PointData>
      <CellData>...</CellData>
    </Piece>
  </ImageData>
</VTKFile>
```

Fig. 5: ImageData format.

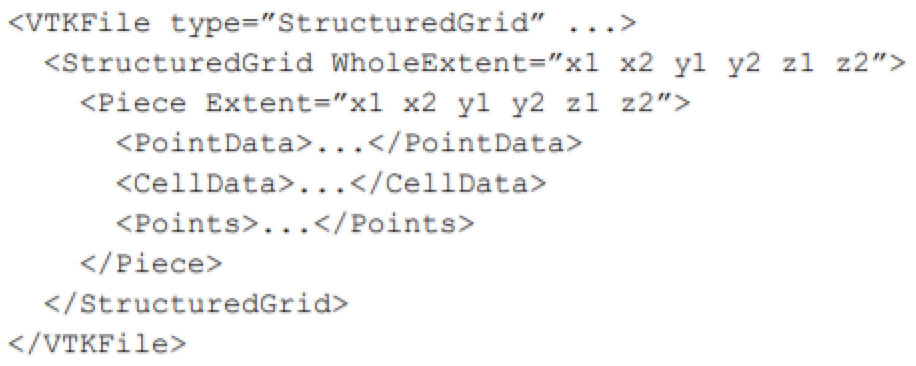
RectilinearGrid: In RectilinearGrid, you have to specify the nodes coordinates along the three axes, Ox, Oy, Oz.



```
<VTKFile type=”RectilinearGrid” ...>
  <RectilinearGrid WholeExtent=”x1 x2 y1 y2 z1 z2”>
    <Piece Extent=”x1 x2 y1 y2 z1 z2”>
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Coordinates>...</Coordinates>
    </Piece>
  </RectilinearGrid>
</VTKFile>
```

Fig. 6: RectilinearGrid format.

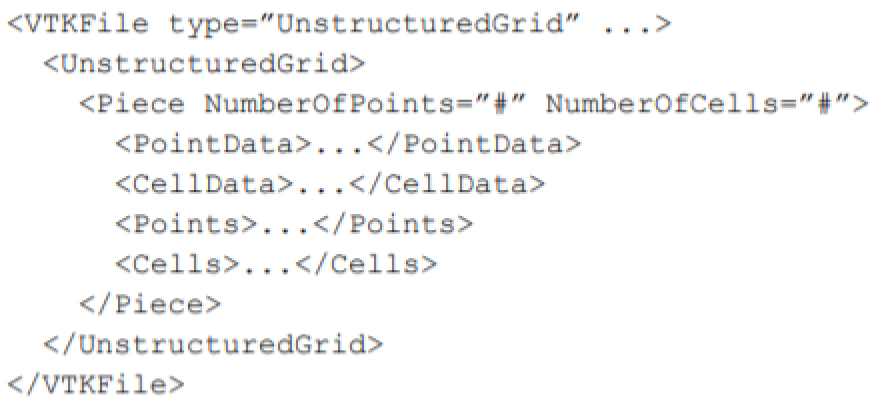
StructuredGrid: For StructuredGrid, you have to specify the coordinates for all mesh nodes.



```
<VTKFile type=”StructuredGrid” ...>
  <StructuredGrid WholeExtent=”x1 x2 y1 y2 z1 z2”>
    <Piece Extent=”x1 x2 y1 y2 z1 z2”>
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Points>...</Points>
    </Piece>
  </StructuredGrid>
</VTKFile>
```

Fig. 7: StructuredGrid format.

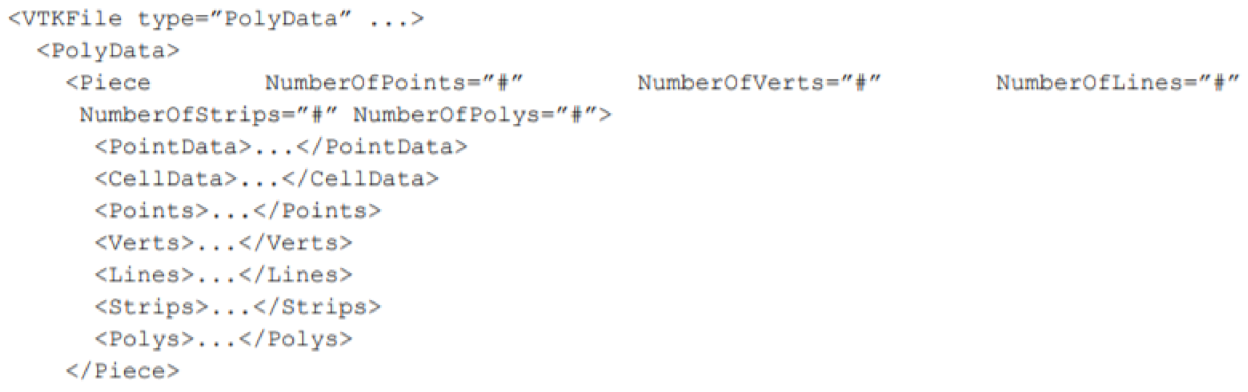
UnstructuredGrid: Each UnstructuredGrid piece specifies a set of points and cells independently from the other pieces. The points are described explicitly by the Points element. The cells are described explicitly by the Cells element.



```
<VTKFile type="UnstructuredGrid" ...>
  <UnstructuredGrid>
    <Piece NumberOfPoints="#" NumberOfCells="#">
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Points>...</Points>
      <Cells>...</Cells>
    </Piece>
  </UnstructuredGrid>
</VTKFile>
```

Fig. 8: UnstructuredGrid format.

PolyData: Each PolyData piece specifies a set of points and cells independently from the other pieces. The points are described explicitly by the Points element. The cells are described explicitly by the Verts, Lines, Strips, and Polys elements.



```
<VTKFile type="PolyData" ...>
  <PolyData>
    <Piece        NumberOfPoints="#"        NumberOfVerts="#"        NumberOfLines="#"
     NumberOfStrips="#" NumberOfPolys="#">
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Points>...</Points>
      <Verts>...</Verts>
      <Lines>...</Lines>
      <Strips>...</Strips>
      <Polys>...</Polys>
    </Piece>
```

Fig. 9: PolyData format.

7.2.4 File Conversion

Rather than writing our mesh file from scratch, it is better to generate our mesh from one of the mesher programs and convert the output into our desired file format. To do this, we will be using a Python module called **Meshio**.

Meshio is a Python library that provides interoperability between all the various mesh formats. Meshio can read and write all of the mesh formats mentioned above and smoothly converts between them.

To convert a file, simply call,

```
$ meshio−convert input.msh output.vtu
```

In Python, simply do,

```
import meshio

mesh = meshio.read(filename)
meshio.write("foo.vtk", mesh)
```

7.3 3-Dimensional Mesh Plotter

For this project, we have chosen PyVista as our visualization library.

7.3.1 Context

After generating our mesh, we want to be able to visualize it so we can see directly the results of our analysis. PyVista supports many plotting functions that are highly controllable and Pythonic.

7.3.2 Dependencies

The following modules are the required dependencies of PyVista:

- vtk - PyVista is built on top of the VTK library
- numpy - Provides direct data access for meshes
- imageio - Used for saving screenshots
- appdirs - Data management for example datasets.

7.3.3 Set Up

To install PyVista simply type on the command line,

```
$ pip install pyvista
```

7.3.4 Plotting

To plot an object, simply call the **pyvista.plot** function that is binded to each PyVista data object.

```
mesh = pv.PolyData(vertices, faces)
mesh.plot()
```

To better customize your plot, you can also create a **pyvista.plotter**

```
plotter = pv.Plotter()
plotter.add_mesh(mesh)
cpos = plotter.show()
```

To display interactive plots, we can also use the **pyvista.BackgroundPlotter** to create a rendering window in the background that remains interactive while the user performs their processing. This creates the ability to update the plotter in real-time.

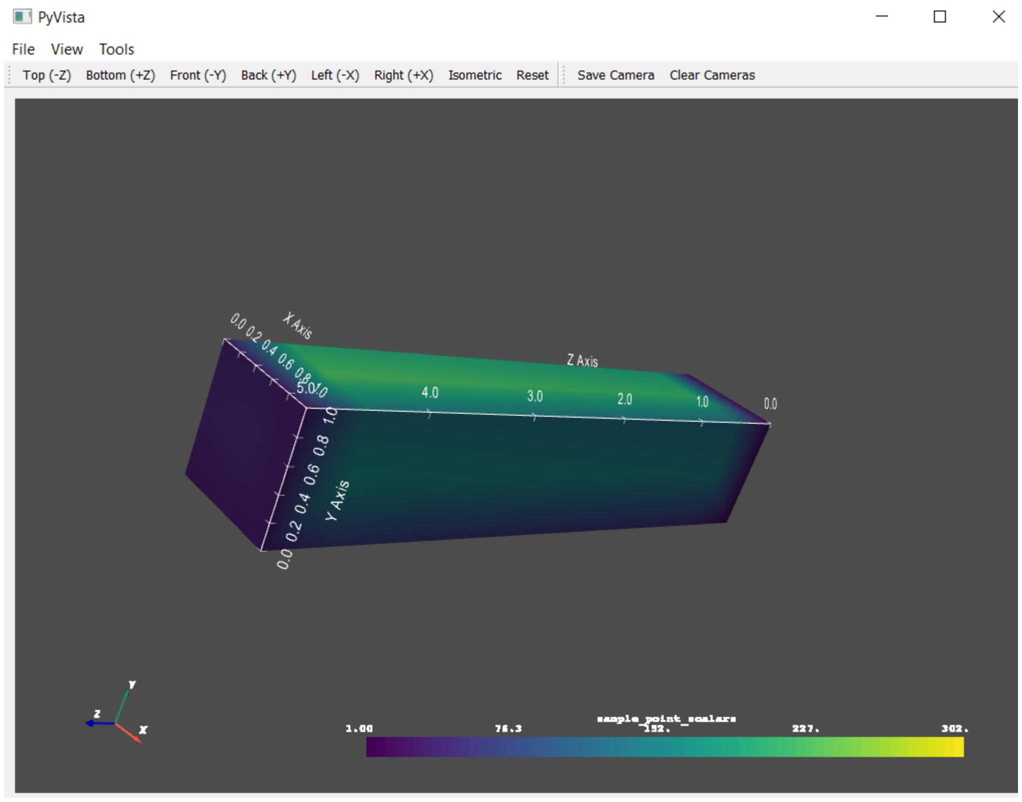

PyVista
File View Tools
Top (-Z) Bottom (+Z) Front (-Y) Back (+Y) Left (-X) Right (+X) Isometric Reset Save Camera Clear Cameras
X Axis
Y Axis
Z Axis
sample_point_scalars
1.00
76.3
152.
227.
302.

Fig. 10: A rendering window that runs in the background, allowing you to update the data in real-time.

8 User Interface

The user interface represents the frontend of the system which allows users to setup, run, and review simulations. The two ways for users to perform these three tasks will be through the graphical user interface and a command line interface. Both interfaces will allow users to input the numerous parameters required to run a simulation. These can be supplied via the GUI's input forms or by submitting a file path to a file containing appropriately formatted data.

8.1 PyQT GUI

8.1.1 Context

The GUI will be constructed using the Python bindings of the QT framework, PyQT. This includes substantial classes, e.g. QApplication, QWidget and QInputDialog, and methods to generate all the necessary parts of the GUI. PyQT was chosen over the numerous other Python libraries for constructing GUIs because of its widespread use in modern software applications, seamless interaction with other libraries being used (PyVista, NumPy, etc.), and availability of training documentation.

8.1.2 Install

Before PyQT5 can be used to implement the GUI it must first be downloaded and installed. After downloading use the command:

```
pip install PyQt5
```

The base object used by PyQT is the QApplication object. The base class for all objects included in the GUI is QWidget. As the name suggests, QInputDialog contains methods for including various input containers in the GUI. These QT classes are the main sources of objects that will be used within the programs GUI.

8.1.3 Composition

The GUI must first provide a way for the user to build their mesh before any calculations can be performed. The mesh is constructed of nodes described by x, y, and z coordinates for the 3D model, x and y for the 2D model. Individual nodes also have attributes to define their material and state. Groups of nodes represent an element which also have attributes to define their state. The GUI will allow the user to supply these sets of attributes through numerical input or via a formatted file. Figure 11 shows an application window with navigation buttons to accomplish these tasks via manual input through the navigation buttons and input fields, or the ability to provide the information in a pre-formatted file. Figure 13 shows an example of a way for a user to provide their nodal information. Each individual cell below the header row is an input field where numerical values can be entered to supply the attributes of each node. Figure 14 contains similar input for each element in the mesh.

After building the mesh the user will have the ability to tweak the attributes of each node, face and model variables before running the model. Figure 15 shows an example of a screen with these capabilities.

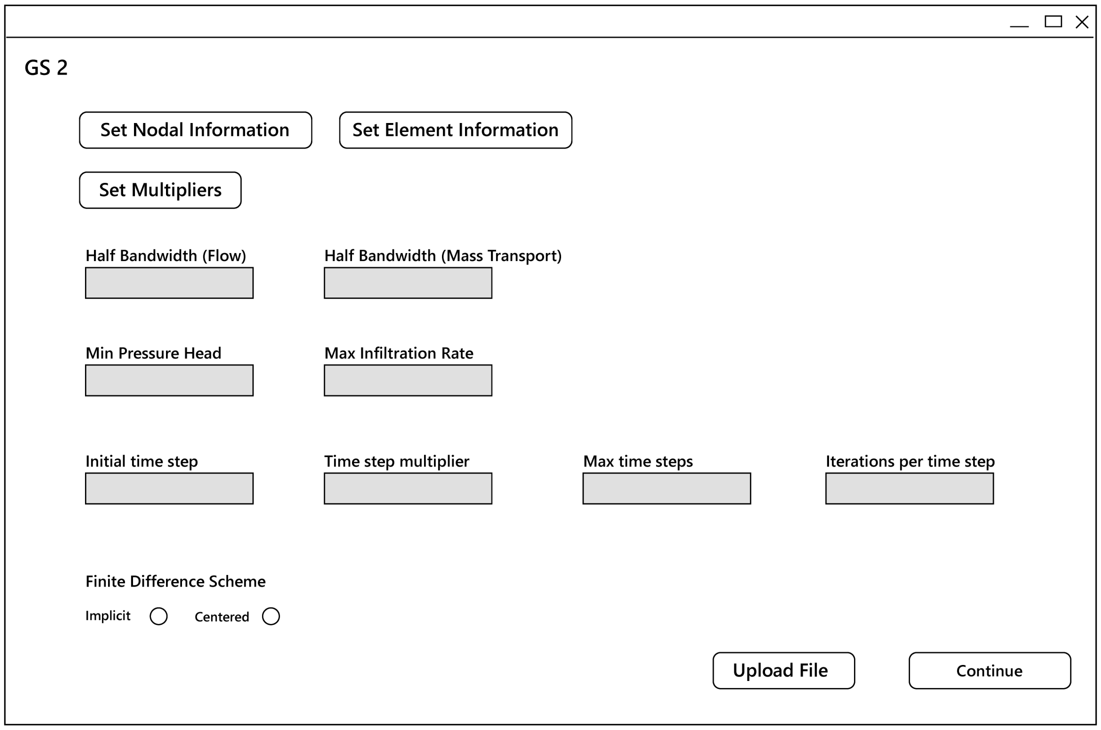
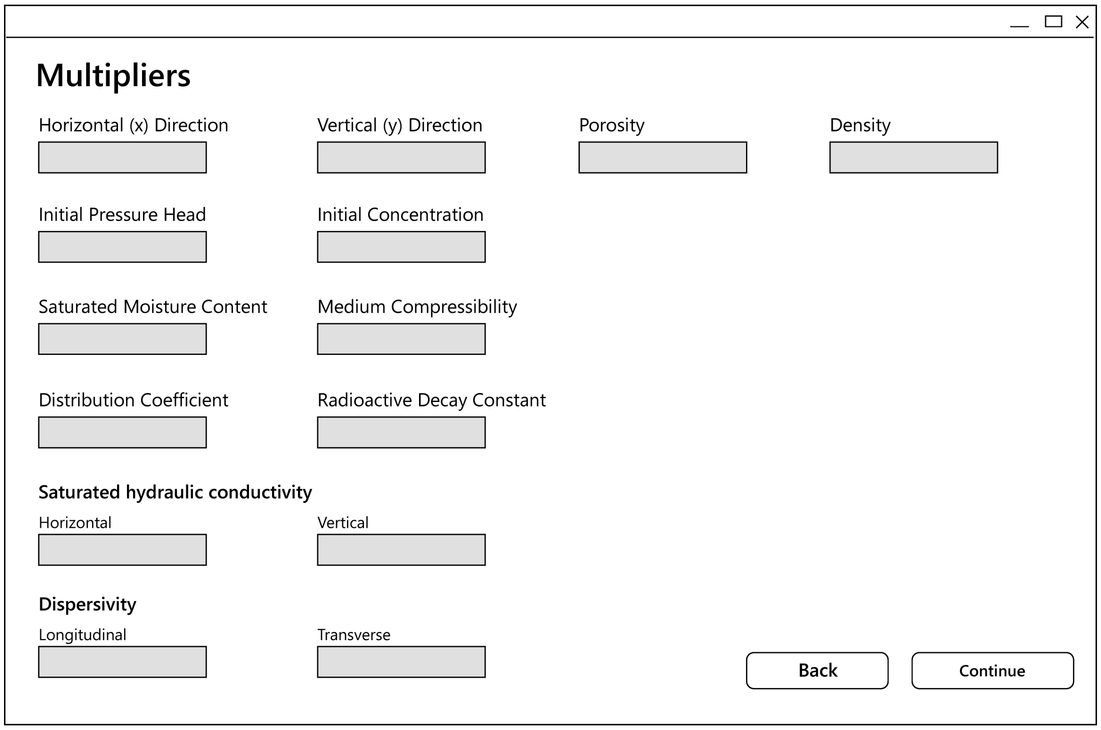


Fig. 11: Project Main Screen



Multipliers

Horizontal (x) Direction	Vertical (y) Direction	Porosity	Density
Initial Pressure Head	Initial Concentration		
Saturated Moisture Content	Medium Compressibility		
Distribution Coefficient	Radioactive Decay Constant		

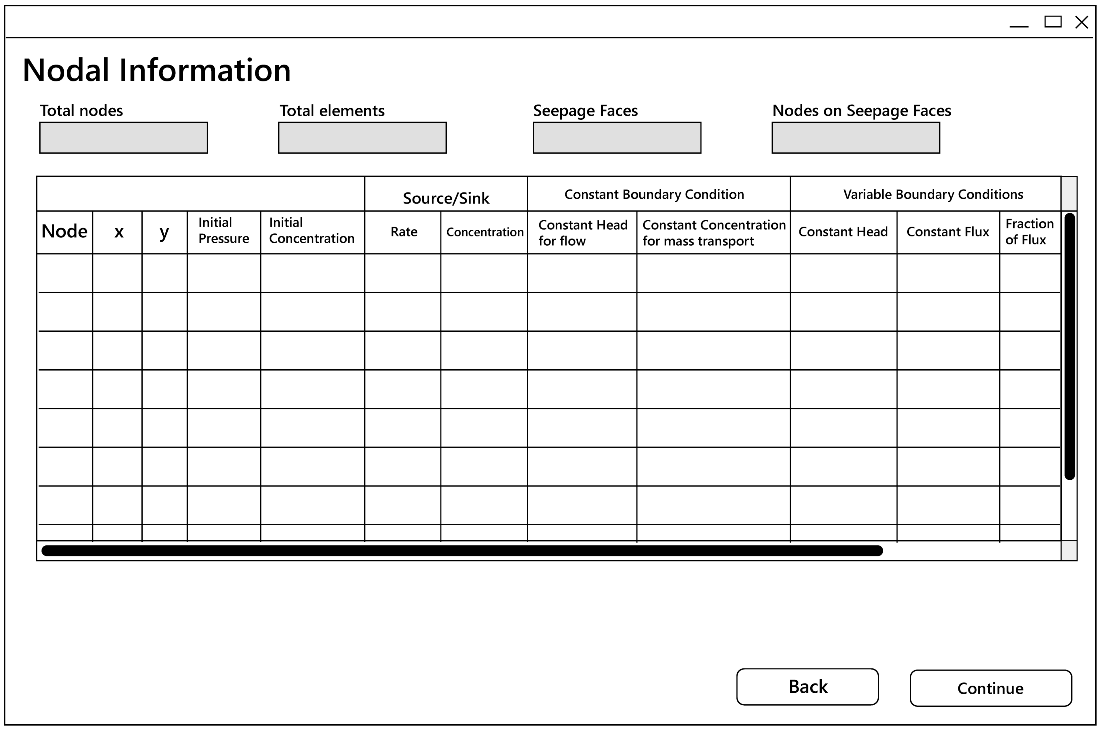
Saturated hydraulic conductivity

Horizontal	Vertical

Dispersivity

Longitudinal	Transverse

Fig. 12: Screen to initialize variable multipliers

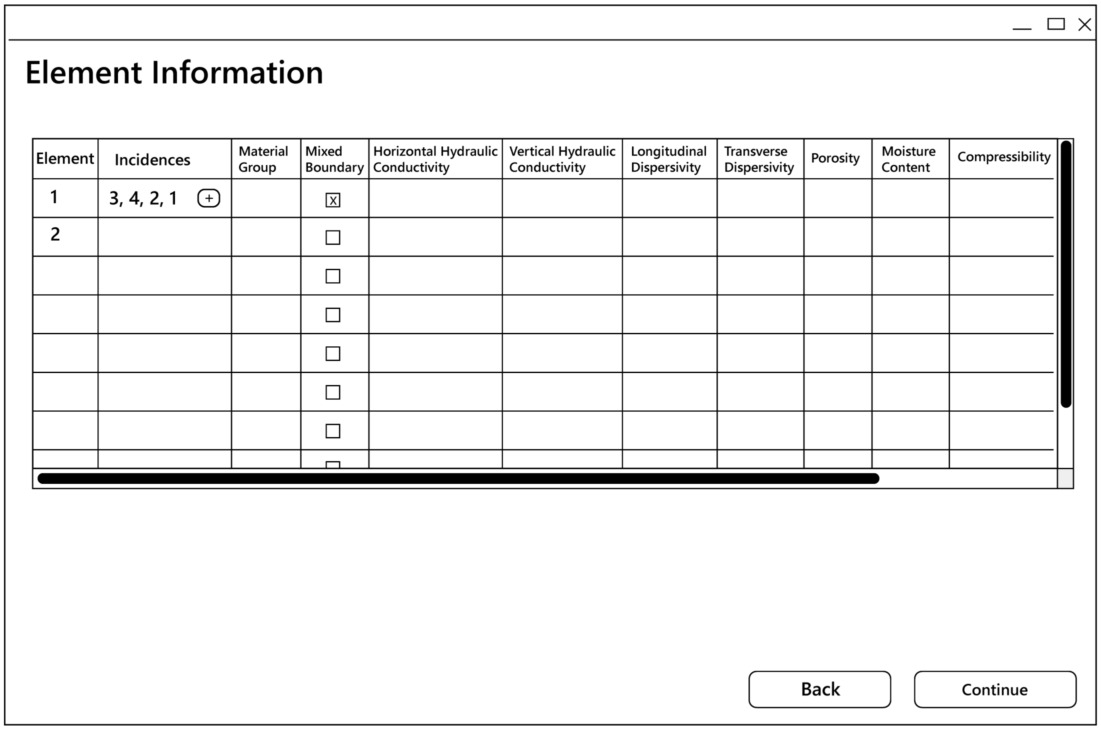


Nodal Information

Total nodes	Total elements	Seepage Faces	Nodes on Seepage Faces

					Source/Sink		Constant Boundary Condition		Variable Boundary Conditions		
Node	x	y	Initial Pressure	Initial Concentration	Rate	Concentration	Constant Head for flow	Constant Concentration for mass transport	Constant Head	Constant Flux	Fraction of Flux

Fig. 13: Screen to set nodal data



Element Information

Element	Incidences	Material Group	Mixed Boundary	Horizontal Hydraulic Conductivity	Vertical Hydraulic Conductivity	Longitudinal Dispersivity	Transverse Dispersivity	Porosity	Moisture Content	Compressibility
1	3, 4, 2, 1 ⊕		☒							
2			☐							
			☐							
			☐							
			☐							
			☐							
			☐							

Fig. 14: Screen to set element data

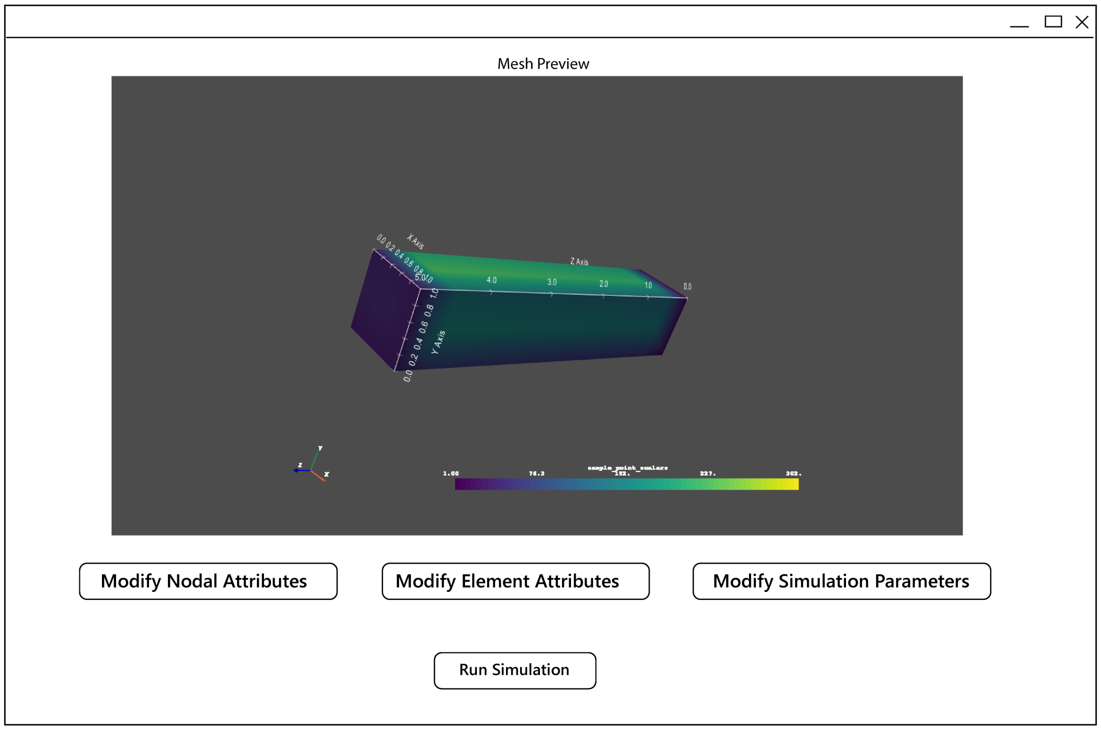


Fig. 15: Preview screen to show mesh and allow modifications

8.1.4 Design Decisions

The GUI provides numerous input fields for users to supply data specific to their mesh and the variables that influence this mesh. Because a mesh could contain hundreds if not thousands of individual nodes, a table was used to display information and allow users to modify that information. This decision was made to allow users to preview information for multiple nodes in a single window, and have the ability to modify the attributes that define these nodes. This same reasoning was used for the window containing element information. The GUI provides a sequential order for supplying data specific to each simulation: simulation parameters, nodal information, and element information but users will be able to transition between sections as they see fit.

8.2 CLI

8.2.1 Context

A command-line interface will also be available for users wanting to bypass the GUI and still be able to execute a simulation. This will require the user to include two file names when executing the main program. This can be done via a command prompt with the command:

```
$ python3 main_prog.py -in nodal_information.txt -out sim_results.txt
```

Order does not matter, as long as the command is immediately followed by the file path. The command '-in' is followed by the path to an input file containing all appropriate parameters for each node included in the model. The command '-out' is followed by the name of the file used to save the results of the simulation.