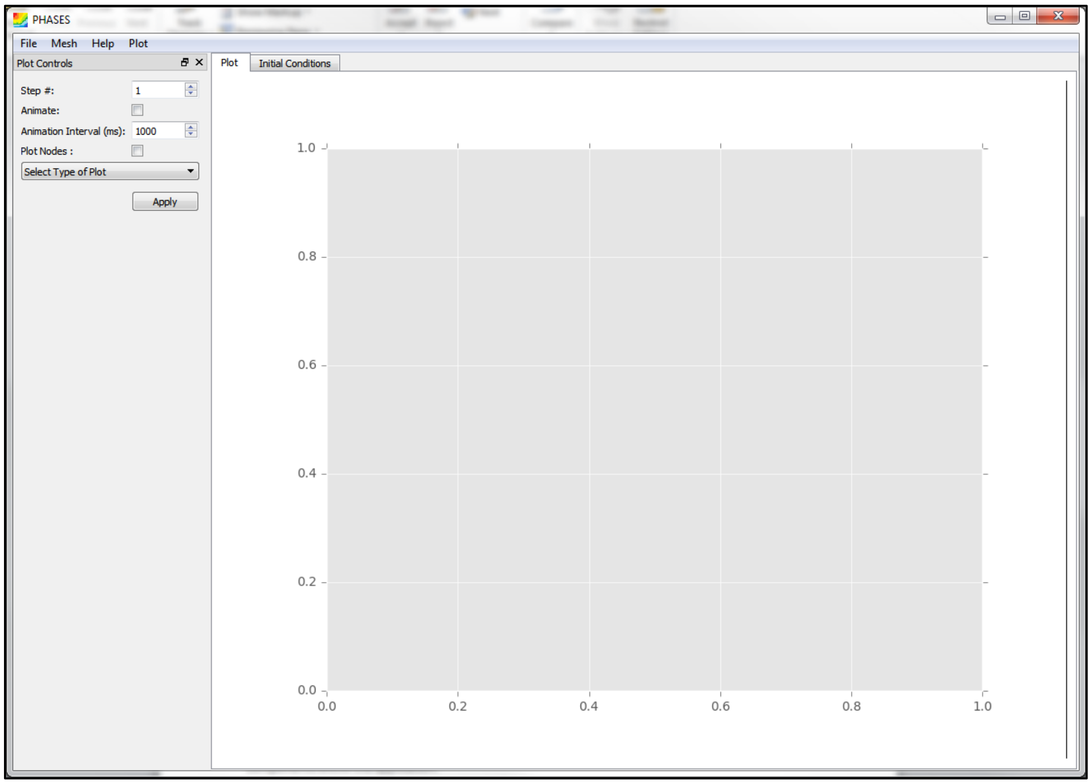
Contents

Introduction

The current implementation of the Phases GUI is based in Python using the PyQt5 and matplotlib modules. PyQt5 is a python based implementation around the Qt GUI framework. Extensive documentation regarding the Qt library can be found online on Qt's website. Although this documentation is written in C++, it works the same way in Python with PyQt with slight adjustments for Python's different syntax. Matplotlib is a common python package that is used for visually graphing data and comes with many sophisticated graphing methods, and with integrated support for embedding graphs in Qt Applications. Matplotlib also supplies documentation on its website.

Functionality

When the application is loaded, it begins with the screen shown in Figure 1. This is the area which is used to plot output data from simulations, and options for the plot area are also displayed to the left. When the application is first opened there is no data to plot. The first step in creating this data is to create the mesh. This can be done through either loading the mesh from a text file that has been created previously by the application or by generating the mesh from scratch. Both of these functionalities can be found in the 'Mesh' menu located on the menu bar.



Step #: 1

Animate:

Animation Interval (ms): 1000

Plot Nodes :

Figure 1 - Main Window

Mesh Generation

When 'Load Mesh From File' has been selected from the 'Mesh' menu a dialog (Figure 2) pops up prompting the user to specify the files from which to load the mesh. The mesh is fully defined by a total of 5 files: the project file, mesh file, initial conditions file, boundary conditions file, and the output file. The mesh file defines the geometry of the mesh which includes all nodes, elements, and boundaries contained within it. The initial conditions file defines the conditions of the mesh at the start of a simulation. The boundary conditions file defines the conditions at the boundary. The output file defines data that has been calculated from a previous simulation. Finally the project file specifies the parameters of the simulation along with all of the other files mentioned before. Once a project file is selected all other files specified in the project file are loaded

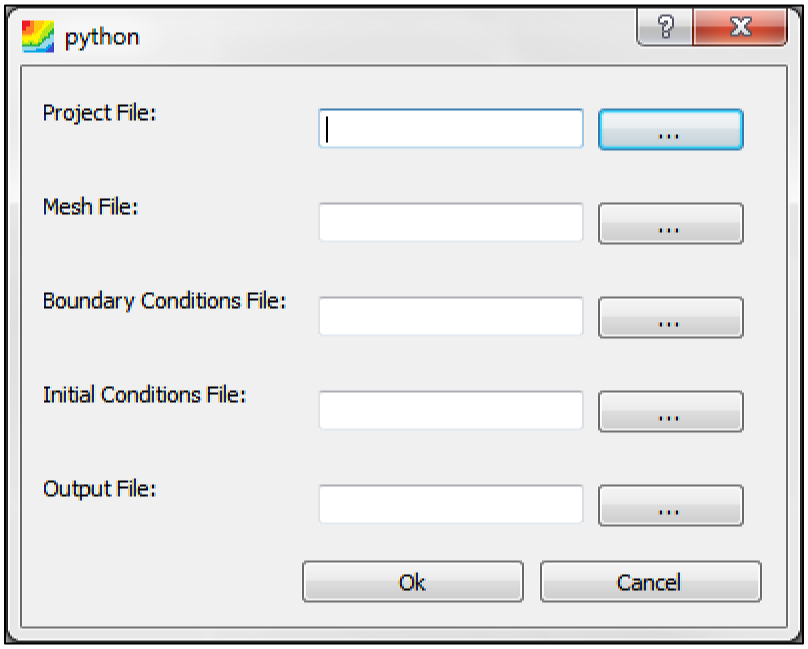


Figure 2 - File Input Dialog

When 'Generate Mesh' is selected from the 'Mesh' menu the mesh generation widget is opened inside the main window. Generating a mesh has three stages. First the user must define several points within the mesh domain that will define the corners between edges. Secondly, edges must be defined that connect those points. These edges will define the boundary of the mesh. Currently only straight edges are implemented but other types could be added such as circular arcs to define curves. Once edges have been defined the boundaries can be sent through a mesh generation algorithm to create a quadrilateral. This final part of the mesh yet been implemented and this algorithm will need to be added. Plenty of examples can be found at http://www.robertschneiders.de/meshgeneration/software.html#public_domain

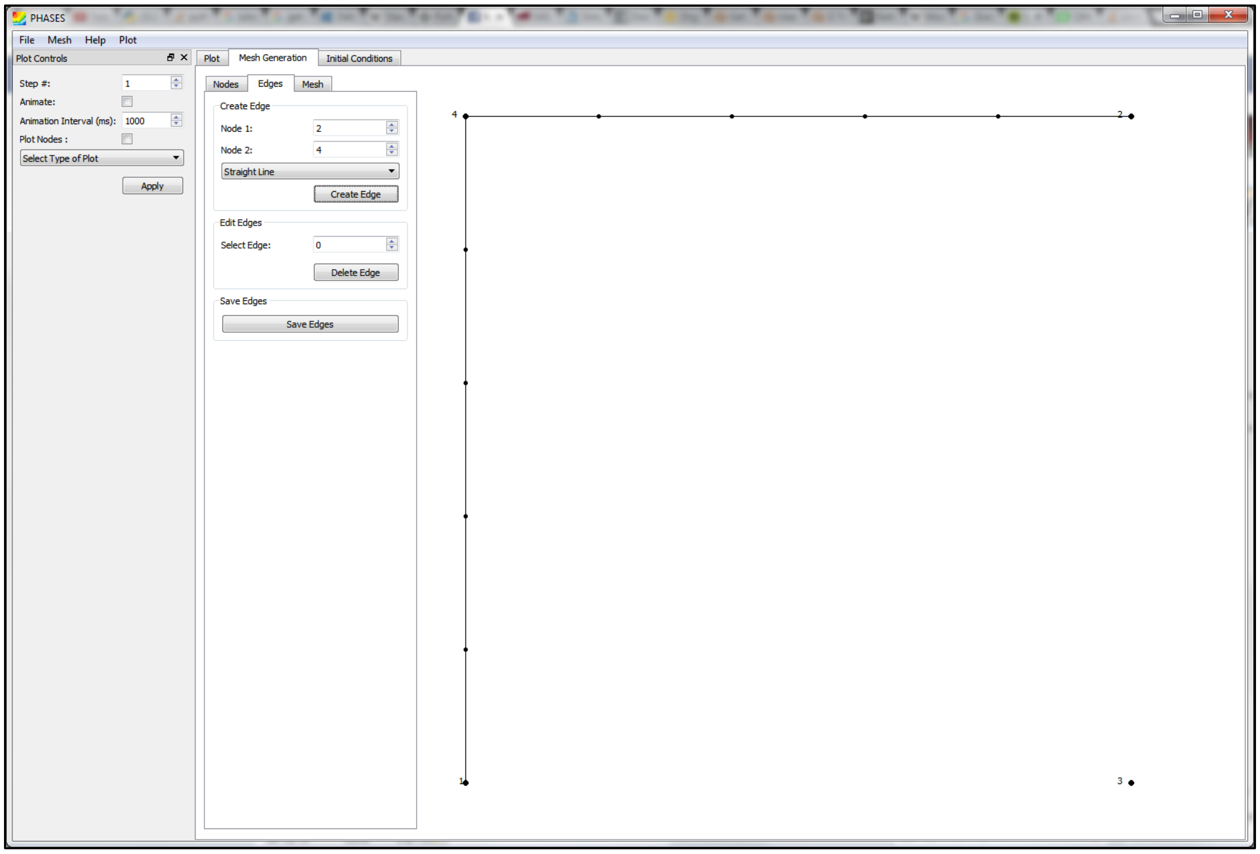
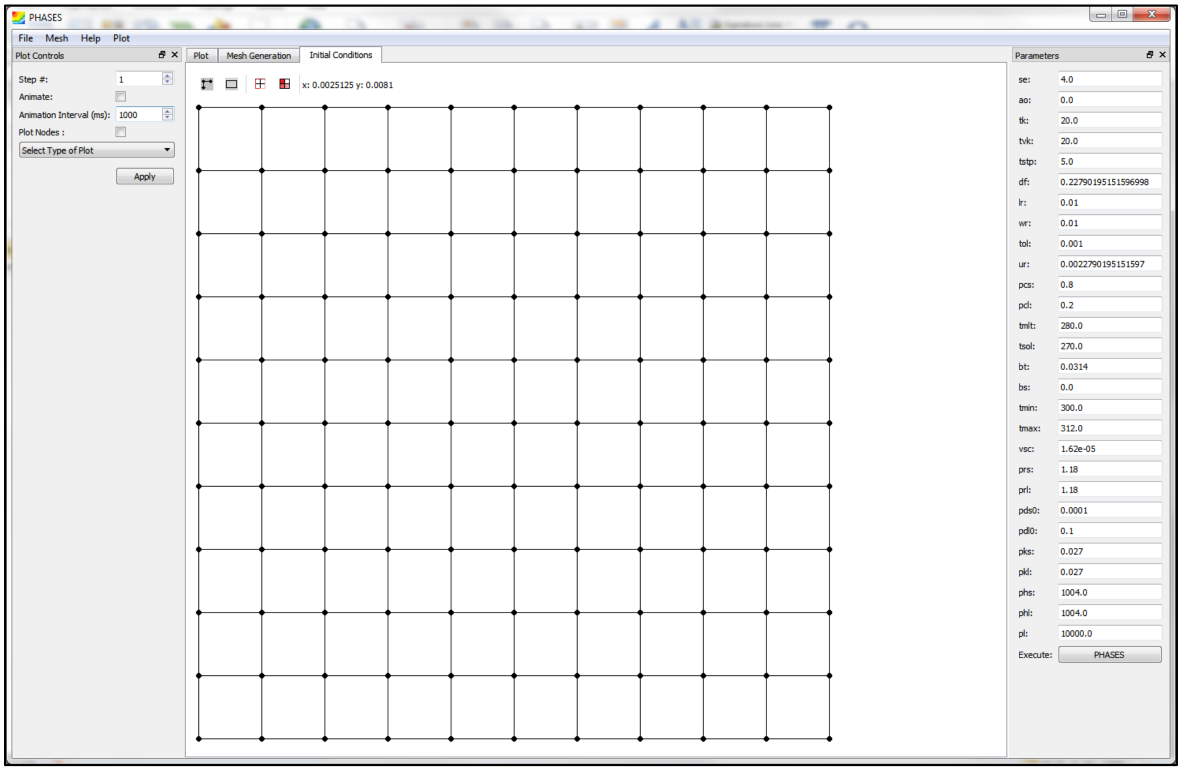


Figure 3 - Mesh Generation

Once a Mesh has been loaded some of its characteristics can be edited by selecting ‘Edit Mesh Conditions’ in the ‘Mesh’ menu. This can also be used to view the structure of a mesh loaded from a file. This opens up the Mesh Editor widget (Figure 4) which can be used to change the mesh’s initial conditions as well as its boundary conditions.



Step #: 1
Animate: ☐
Animation Interval (ms): 1000
Plot Nodes : ☐
Select Type of Plot

x: 0.0025125 y: 0.0081

se:	4.0
ao:	0.0
tk:	20.0
tvk:	20.0
tstp:	5.0
df:	0.22790195151596998
lr:	0.01
wr:	0.01
tol:	0.001
ur:	0.0022790195151597
pcs:	0.8
pcl:	0.2
tmlt:	280.0
tsol:	270.0
bt:	0.0314
bs:	0.0
tmin:	300.0
tmax:	312.0
vsc:	1.62e-05
prs:	1.18
prl:	1.18
pds0:	0.0001
pdl0:	0.1
pks:	0.027
pkl:	0.027
phs:	1004.0
phl:	1004.0
pl:	10000.0
Execute:	PHASES

Figure 4 - Mesh Editor

Running a Simulation

Once a mesh has been created the Parameters widget appears on the right side of the application. This can be used to change some of the parameters of a simulation before running it. Two common ones are df, the length of a time step, and tstp the number of time steps. When a simulation is run it will calcite data for every node in the mesh at discrete time steps defined by df and tstp. A simulation can be started by clicking the PHASES button at the bottom of the Parameters widget.

Plotting Data

Plotting data is done through the Python package matplotlib. It is displayed in the Plot widget, and controlled by the Plot Controller widget. The Plot Controller allows the user to select which time step they want to view and which type of graph to plot. When a graph type is selected additional settings will show up depending on the graph so that the user can control how it is viewed. There is also an animation feature which when selected will cause the application to automatically loop through and display each time step in the plot area.

Outputting Data

Any mesh can be saved to text files so that it can be loaded, viewed and simulated again later by selecting ‘Save Mesh To File’ in the ‘Mesh’ menu. It’s also possible to save matplotlib graphs as pngs or jpegs, although this hasn’t been implemented in the application yet. It might also be possible to save gifs of animated simulations this way too.

General Code Structure

QtPhases

QtPhases is the entry point into the application. It contains the class ApplicationWindow which initializes all of the widgets, organizes the main layout, and controls how the widgets communicate between each other.

Simulation

The majority simulation code is written in C++ and compiled as a Dynamic-Link Library (dll) file. The source code is contained in the phases Visual Studio solution. Currently the source code is only compiled for Windows machines. To port to other systems it would need to be compiled on a MacOS device or Linux machine.

The PhasesCaller class is responsible for loading this dll and providing the API for calling its functions.

Mesh

Meshes are defined in the Mesh.py file. Nodes, Elements, and Boundaries are also defined in this file. The protocol for reading and writing meshes to text file can also be found here. The Mesh class contains a method called sendToAdda which when called will initialize a PhasesCaller, pass on the mesh data through it, and then read Phases' output data.

Mesh generation is split into two different widgets. The Mesh Generator, which can be used to generate a mesh from scratch, and the Mesh Editor which can be used to set a mesh's initial and boundary conditions. The Mesh Generator along with its associated widgets can be found in the Mesh\Generator subdirectory, and the Editor and its associated files can be found in the Mesh\ConditionEditor subdirectory.

The MeshGenerator is split into 4 other widgets. The NodeEditor, EdgeEditor, MeshCreator, and the GeneratorDisplay. The NodeEditor provides functionality for adding nodes which are to act as corners. The edgeEditor provides functionality for adding edges for the boundary. The MeshCreator will take this boundary and generate a mesh from it. The GeneratorDisplay will visually display the mesh while it is being created. Currently the MeshCreator class has not been implemented.

The MeshEditor widget has a display as well as 4 tools for editing the mesh initial and boundary conditions. Two of these tools are used to define how conditions are edited, one for initial conditions and one for boundary conditions. The other two define how elements or boundaries are selected, one for single select and one for box select. Single select allows the user to select objects one at a time, with holding Ctrl allowing them to select multiple items. Box select allows the user to draw a box which will then select all objects inside of it. When the user right clicks an option will appear which opens a dialog prompting the user to input the conditions. The dialog options will depend on whether the initial conditions or boundary conditions tool is selected. These changes will then be applied to all objects that are selected.

The initial conditions edit tool also has a function which will subdivide all selected elements. Current this only works on rectangular elements. This can be used to quickly edit the mesh to allow for more detail in simulations and plotting.

Plotting

Plotting is split into two parts, the Plot Controller for editing plot settings, and the mplCanvas for doing the actual plotting.

The mplCanvas file defines the canvas which is placed into the main window. It also defines the PlotType class. Any type of graph which is to be added should inherit from the PlotType class. The PlotType class is used as an abstract class and contains two methods that need to be implemented by its children: settings and plotFunction. The settings method should return a QWidget that can be used by the PlotController to allow the user to control settings specific to the type of graph. The plotFunction takes the data of the simulation and tells the mplCanvas how to plot it. Two main child classes of PlotType are defined which are the ContourPlot and VectorField plots. These provide some common functionality between all contour plots and vector field plots respectfully, although neither implements the plotFunction method. These classes are also meant to be inherited and have the plotFunction method defined through their children.

Building for Distribution

Unlike C++ a Python program is typically run without compiling and building it. This is useful during development but makes a Python application difficult to distribute as it requires Python to be installed on the computer in order to run it. The setup. py file contains a script which uses the cx_Freeze library to compile the application into an executable which can be run independently of a Python install. To build the application the command 'python setup.py build' needs to be run through a windows command line in the directory where setup.py is placed

However the setup.py script is incomplete in its current form. Firstly there is an error in importing the numpy package which is necessary for handling arrays of data. The dlls files in numpy\core do not copied properly and must be manually copied to the same folder as the compiled executable. However, once these files only need to be copied into the build directory they will remain there in between builds. The script can be changed to automatically find and copy these files over automatically during a build.

A bigger error exists in the script in that when bigger simulations are run from the compiled executable it creates a stack overflow error which crashes the application. It is uncertain whether this is caused by the cx_Freeze build, the C++ source code, or the QtPhases application itself, although no simulations have caused the same errors in un-compiled python.

Detailed Documentation

This section outlines the structure of the files and classes contained in the Python GUI implementation.

QtPhases.py

Class ApplicationWindow

Inherits from the Qt class QMainWindow (http://doc.qt.io/qt-5/qmainwindow.html) and contains all of the same attributes and methods.

This is the main window of the entire application. It is what you see when the application is first opened. It controls the main layout of the application and it is what initializes all of the other components within the application.

Attributes

Name	Type	Description	Documentation
main_widget	QTabWidget	The central widget of the main window.	http://doc.qt.io/qt-5/qtabwidget.html
mesh	Mesh	Stores all of the data necessary for a phases simulation.	Class Mesh
plot_widget	QWidget	The widget used for displaying graphs.	http://doc.qt.io/qt-5/qwidget.html
mainPlot	DynamicCanvas	The class which makes the plots and contains the graphing functions.	Class DynamicCanvas
plotcontrols	PlotController	A widget for controlling the plot settings.	Class PlotController
meshEdit	MeshEditor	A widget used for editing a mesh's initial and boundary conditions.	Class MeshEditor
meshGenerate	MeshGenerator	A widget used for generating a mesh from scratch. Not fully developed.	Class MeshGenerator
controlsDock	QDockWidget	A container for the plotcontrols.	http://doc.qt.io/qt-5/qdockwidget.html
phasesControls	PhasesController	A widget for editing project parameters and executing a simulation.	Class PhasesController
phasesDock	QDockWidget	A container for the phasesControls.	http://doc.qt.io/qt-5/qdockwidget.html
datafiles	dict	A container to keep track of the most recent text files to be selected to load a mesh.	N/A
file_menu	QMenu	A menu bar item.	http://doc.qt.io/qt-5/qmenu.html

mesh_menu	QMenu	A menu bar item.	http://doc.qt.io/qt-5/qmenu.html
plot_menu	QMenu	A menu bar item.	http://doc.qt.io/qt-5/qmenu.html
help menu	QMenu	A menu bar item.	http://doc.qt.io/qt-5/qmenu.html

Methods

updatePlot(plotType,timeStep)

This method is used to pass updated plotSettings to the matplotlib widget. The plotController does not have a reference to the mesh and the data assoxiated with itso the signal needs to be passed here first so the mesh can be passed along with the plot settings.

plotType - a member of PlotType instance
timeStep - an integer specifiying the timeStep which is to be graphed

dockcontrols()

A method for docking the plotcontrols to the left side of the window. If they are already docked there this should do nothing.

dockphases(parameters)

A method for docking the phasescontrols to the right side of the window. If they are already docked there this should do nothing.

parameters – a dictionary specifying initial parameters which will populate the phasesController parameter input text boxes .

phasesExec(steps = stepsDefault, params = None)

This is what tells the mesh to send itself to phases, the C++ library which will then perform all of the number crunching. The results of the simulation will be stored in the mesh class.After the simulation is completed, the graph widget is reset.

steps - the number of time steps which are to be simulated.
If not specified defaults the the global variable stepsDefault.

params - the parameters to run the simulation with. If not specified, will use the parameters already contained in mesh.

about()

Short dialog explaining what the application is.

loadMeshFromFile()

Opens a dialog prompting the user to specify project files that specify a mesh and its initial conditions. A mesh will be created immediately form the specified data files. File validation has not yet been implemented so improperly formatted files will likely crash the application or case significant errors.

openPainter()

A method to open the mesh generation widget in a new tab within the main application widget.

tabChanger(index)

Stops the graph widget from animating while other widgets are active. The animation causes some slow down which is annoying if the user is trying to do other things.

> **index** - the index of the tab that now has focus

message(msg, title = 'Error!')

A method for creating popup boxes to display error messages.

> **msg** - string which will be displayed to the user.
> **title** - title of popup, defaults to 'Error!'

Class fileInputDialog(parent, files)

Inherits from the QDialog class (http://doc.qt.io/qt-5/qdialog.html)

This is a dialog window that allows the user to specify files which will be used to define a mesh, its boundary conditions, its initial conditions, some parameters and an optional output text file for storing output data. Once the files have been specified the newFiles_Specified signal will be emitted.
Parent specifies the parent widget for the dialog, and files specifies some initial values to be placed in the file dialog boxes. Should contain the keys ‘Project’, ‘Boundary Conditions’, ’Initial Conditions’, ’Mesh’, and ‘Output’.

Signals

newFiles_Specified(dict)

When the dialog is closed by the ‘Ok’ button this signal will be emitted sending the file directories in a dictionary with the keys ‘Project’, ‘Boundary Conditions’, ’Initial Conditions’, ’Mesh’, and ‘Output’.

Methods

execute()

Executes the dialog. Emits and returns specified files if ok. Does nothing if user quits dialog or hits cancel.

Mesh.py

Class Mesh

A class which contains the mesh which is to be the domain of a simulation. Organizes the mesh boundaries, elements and nodes while providing functionality for extracting their data.

Attributes

Name	Type	Description	Documentation
nodes	List<Node>	A list of all the nodes, organized in ascending x-value then ascending y-value.	Class Node
elements	List<Element>	A list of all the elements organized based on the index of its top-left node.	Class Element
boundaries	List<Boundary>	A list of all the boundaries organized by the index of their associated elements.	Class Boundary
params	Dict<str:float>	Stores the parameters to be passed to the simulation.	N/A

Methods

addNode (node)

Adds the Node **node** to the mesh's nodes if it is not already there.

subNode(node)

Subtracts the Node **node** from the mesh's nodes, if it is there.

addElement(element)

Adds the Element **element** to the mesh's elements if it is not already there.

subElement(element)

Subtracts the Element **element** from the mesh's elements, if it is there.

addBoundary(boundary)

Adds the Boundary **boundary** to the mesh's boundaries if it is not already there.

subBoundary(boundary)

Subtracts the Boundary **boundary** from the mesh's boundaries, if it is there.

getXY()

Returns the xy-coordinates of the mesh's nodal points as a 2D numpy array.

getTemperature(timeStep)

Returns the temperature values of the mesh's nodal points at the given **timeStep** as a 2D numpy array.

getUV(timeStep)

Returns the u-Velocity and v-Velocity values of the mesh's nodal points at the given **timeStep** as a two 2D numpy arrays.

getFLIQ(timeStep)

Returns the liquid fraction values of the mesh's nodal points at the given **timeStep** as a 2D numpy array.

getPressure(timeStep)

Returns the pressure values of the mesh's nodal points at the given **timeStep** as a 2D numpy array.

sendToAdda(self)

Sends the mesh's data to the C++ source code through a PhasesCaller and stores the output data into the nodes of the mesh. This output data can be read through the various get___ methods

Class Node (x,y)

A data structure for containing information about a mesh's nodes

x - the node's x coordinate

y – the node's y-coordinate

Attributes

Name	**Type**	**Description**	**Documentation**
x	float	x-coordinate of the node	N/A
Y	float	y-coordinate of the node	N/A
index	int	The node index used by the phases C++ source.	N/A
isBoundary	Bool	True if the node is on the mesh boundary, False otherwise.	N/A
temperature	List<float>	Temperature data at the node for each time step.	N/A
concentration	List<float>	Concentration data at the node for each time step.	N/A
uVelocity	List<float>	U-velocity data at the node for each time step.	N/A
vVelocity	List<float>	V-velocity data at the node for each time step.	N/A
wVelocity	List<float>	W-velocity data at the node for each time step.	N/A
pressure	List<float>	Pressure data at the node for each time step.	N/A
liquidFrac	List<float>	Liquid Fraction data at the node for each time step.	N/A

Methods

clearData()
Erases all output data (temperature, concentration, velocity, pressure, liquid fraction)

__eq__(otherNode)
Two nodes are considered equal if they have the same x and y coordinates.

Class Element(nodes)
A data structure storing information about mesh elements

nodes – a list of Nodes that define the element. Must be a length of 4.

Attributes

Name	**Type**	**Description**	**Documentation**
nodes	List<Nodes>	The nodes that define the element, a total of four.	Class Node
index	int	The element index used by the phases C++ source.	N/A
isBoundary	bool	True if the element is on the mesh boundary, False otherwise.	N/A
boundaries	List<Boundary>	A list of the Boundary edges that are a part of this element.	Class Boundary
initialConditions	Dict<str:float>	The dicitionary has keys 'concentration','temperature','u-velocity','v-velocity' which are associated with float values representing the element's initial conditions.	N/A

Class Boundary(nodes)

A data structure storing information about mesh boundaries

nodes – a list of Nodes that define the boundary. Must be a length of 2.

Attributes

Name	Type	Description	Documentation
nodes	List<Nodes>	The nodes that define the element, a total of two.	Class Node
index	int	The boundary t index, used by the phases C++ source.	N/A
element	Element	The element that contains this boundary	Class Element
conditions	dict<str:list<list<float>>>	A dictionary with keys 'concentration','temperature','u-velocity','v-velocity' which are associated to the boundary conditions of the boundary. Boundary Conditions are stored as a list of two lists, each of which contain 3 elements representing the coefficients [a,b,c] in the boundary condition equation: aT'+bT=c	N/A

Function loadMeshFromFile(files)

Files - a dictionary with the keys 'Project', 'Boundary Conditions', 'Initial Conditions', 'Mesh', and 'Output', which point to directories of the appropriate text files. These text files are read, and their data is used to initialize a mesh object with all of its elements, boundaries and nodes. Returns the mesh object.

Function saveMeshToFile(mesh,directory,name)

Takes the data from the Mesh **mesh** and saves it in 5 text files: the mesh, boundary conditions ,initial conditions,project, and output files. These files a placed in the given file **directory** and given a label specified by **name**.

mplCanvas.py

Class DynamicCanvas

An implementation of the matplotlib FigureCanvas class in the Qt framework. matplotlib functions and graphs can be used through this class

Attributes

Name	**Type**	**Description**	**Documentation**
plotTypes	List<PlotTypes>	A list of all the implemented PlotTypes	Class PlotType

Methods

update_figure(plotType,timeStep, mesh, doNodes)

Updates the canvas with the specified graph and data.

plotType – the PlotType instance which is to be used to do the graphing

mesh – the Mesh class which contains the data

doNodes – if true a scatter plot of the nodes will be overlaid on the graph

Class PlotType(canvas)

An abstract class that can be inherited by other classes to implement plot functions

canvas - the associated graph widget

Attributes

Name	**Type**	**Description**	**Documentation**
label	str	A name for the plotType.	N/A
canvas	DynamicCanvas	the associated graph widget	Class DynamicCanvas

Methods

plotFunction(*args,**kwargs)

The function that defines the graph that gets plotted

settings ()

a function that returns a QWidget which can be used to control some parameters that are specific to a graph

Class Default(canvas)

The default PlotType. Sets the label to ‘Select Type of Plot’ and doesn’t graph anything

Class ColorMapPlot(canvas)
An implementation of the PlotType class which allows for the editing of plot color maps.

Overrides the settings function and creates a widget that can be used by the user to select a color map for a graph. Currently all other PlotTypes that are implemented inherit from this class.

Does not implement the plotFunction method and should still be considered an abstract class

Class ContourPlot(canvas,label)
Inherits from ColorMapPlot to add additional functionality to the widget returned by the settings method for making contour plots. It allows the user to define the number of contours that are plotted with a maximum of 500.

Does not implement the plotFunction method and is meant to be inherited.

Class tempContour(canvas)
Inherits from the ContourPlot class. Implements the plotFunction method to pull temperature data and plot it in contours.

Also adds additional functionality to the settings widget so that the user can select the temperature units as Kelvin, Celsius, or Farenheit.

Class isothermalLines(canvas)
Inherits from the tempContour class. Slightly modifies the plotFunction method to plot the contour without any fill.

Class fliqContour(canvas)
Inherits from the ContourPlot class. Implements the plotFunction method to pull liquid fraction data and plot it in contours. Does not change the settings method.

Class pressureContour(canvas)
Inherits from the ContourPlot class. Implements the plotFunction method to pull pressure data and plot it in contours. Does not change the settings method.

Class VectorField(canvas,label)
Inherits from ColorMapPlot to add additional functionality to the widget returned by the settings method for making vector field plots. It allows the user to define vector lengths and the option to add a contour plot below the vector field on the graph. This contour can be any of the currently implemented contour plots.

Does not implement the plotFunction method

Class velVectors(canvas)
Inherits from VectorField and implements the plotfunction method to pull u-velocity and v-velicty data to plot a velocity vector field.

PlotController.py

Class PlotController(parent, graphWidget)

Inhereits from the Qt class QWidget.

Parent – this widget's parent widget

graphWidget – the widget that is used for plotting. Must contain a list called plotTypes of objects that have an attribute called labels. Currently this is implemented with the Dynamiccanvas and PlotType classes but if other graphing libraries are used they should interface well.

Attributes

None that should accessed directly.

Signals

plotSettingsUpdated(PlotType, int, bool)

Emits the PlotType that has been selected, an integer representing the time step to be plotted, and a bool stating whether or not to plot a scatter plot of the nodal xy-coordinates. This is emitted under two circumanstances: once the 'Apply' button is clicked, or if the graph is being animated it is emitted when it is time for an update.

Methods

plotChanged(plotIndex)

When a PlotType is selected this method is called and will load its settings widget and place it at the bottom of its layout

updateGraph()

Reads the current user-selected settings and emits them through the plotSettingsUpdated signal

update_controls()

Calls updateGraph and if animated is selected will begin the animation process.

animation()

While being animated this is called at each tick of the animation. Updates the plot's time step then calls updateGraph.

setMaxIndex(towhat)

This sets the maximum time step index of the plot controls to the integer **toWhat**.

reset()

Resets the plot controls to their default values when the application is opened

PhasesController.py

Class PhasesController(parent, initParams)

This class inherits from the Qt class QWidget. Paent specifies this widget's parent, and initParams specifies the parameters to use at first when this window is opened. The widget is structured as a vertical layout of many text input boxes which can be used by the user to pass in float values for parameters.

Signals

executed(dict)

Once the executed button has been clicked this signal is emitted containing all of the parameter values that have been specified by the user.

Methods

getParam(paramToGet)

Returns the float value of the parameter specified by the string **paramToGet**

getAllParams()

Returns a dictionary containing all of the parameters

setParam(param,toWhat)

Sets the parameter specified by the string **param** to the float **toWhat.**

setAllParams(params)

Copies all parameters form the dictionary **params** and places them in this widgets parameter input boxes.

PhasesCaller.py

This class is used as a simple Python wrapper for a C++ library which is used to do the number crunching within a simulation. Current Implementation only works on Windows machines.

Methods

Adda(a,b,c,d,e,f,g,h,i,j)

The only function required for passing data between Python and C++ by calling the PyAdda function from the C++ library. Passes values one at a time, specified by the arguments a,b,c,d,e,f,g,h,i,j. The function of each of these parameters can be seen by looking at the C++ source.

testAdda(a,b,c,d,e,f,g,h,i,j)

Does the same as Adda but prints some information to the Python Console for debugging purposes.

Mesh.ConditionEditor.MeshEditor.py

Class MeshEditor

This inherits from the Qt class QWidget. This is the entry point into the mesh editor functionality. Only this class needs to be placed inside the main window and the rest of the widgets will be contained inside of it. This widget contains two other widgets: the toolbar and the MeshDisplay. The toolbar contains four tools which are used to allow the user to edit initial and boundary conditions of the mesh. Two of these tools are used for selecting objects and the other two are used to edit the objects.

Attributes

Name	Type	Description	Documentation
canvas	MeshDisplay	The widget which displays the current mesh	Class MeshDisplay
mesh	Mesh	A reference to the current mesh	Mesh
singleSelect	Tools.SingleSelect	The tool for selecting single objects	Class SingleSelect
boxSelect	Tools.BoxSelect	The tool for selecting multiple objects quickly	Class BoxSelect
icTool	Tools.ICEditTool	The tool for editing initial conditions	Class ICEditTool
bcTool	Tools.BCEditTool	The tool for editing initial conditions	Class BCEditTool

Methods

setMesh(mesh)

Sets the current mesh to the given Mesh **mesh.**

Class MeshDisplay

The widget used for displaying the mesh within the MeshEditor widget.

Attributes

Name	Type	Description	Documentation
displayTool	DisplayTool	Either an ICEdittool or a BCEditTool depending on what the user has selected to use.	Class ICEditTool Class BCEditTool
selectTool	SelectTool	Either a SingleSelect or a BoxSelect tool depending on what the user has selected to use.	Class BoxSelect Class SingleSelect
currentObject	Element, Boundary or Node	An object that is calculated during every mouseMoveEvent within the widget. The display defines what the current object is based on the mpuse x and y coordinates.	Class Element Class Boundary Class Node

Signals

mouseMoved(str)
A signal that is emitted whenever the mouse is moved inside this widget. Contains a string which states the xy-coordinates of the current mouse position.

Methods

showContextMenu(self,point)
Called whenever the right mouse button is pressed. Opens a right-click menu for the user that contains options that are specified within the display tool. These actions will then be used on currently selected objects.

paintEvent(event)
The method that controls what is displayed on the widget. This method will draw all of the nodes, elements, and boundaries that are specified within the mesh. Calls the method drawStuff from the selectTool which will then draw additional things onto the display depending on which objects are selected.

mouseMoveEvent(event)
Called whenever the mouse is moved within this widget. **event** contains information such as the mouse position. Calls the mouseMoveEvent methods of both the displayTool and the selectTool. DisplayTool sets the currentObject and the selectool may perform certain actions with it. Emits the mouseMoved signal.

mousePressEvent(event)
Caaled whenever a mouse button is pressed. If left mouse button the slectTool mousePressEvent is called.

mouseReleaseevent(event)
Called whenever a mouse button is released. Calls the selectTool mouseReleaseEvent

getRealXY(qx,qy)
Returns the real x and y coordinates based on the widgets internal coordinate system coordinates (given by **qx** and **qy**)

Mesh.ConditionEditor.Tools.py

Class BCEditTool

Methods

selectableObjects
Returns the mesh’s boundaries.

mouseMoveEvent(x,y)
Calcualtes whether or not the **xy** coordiantes are occupied by a boundary. If so it returns that boundary, if not returns None.

drawStuff(painter, boundary)
Paints the given **boundary** as yellow using the given QPainter.

contextMenuActions(menu,Boundaries)
Adds the option to edit boundary conditions on the given **boundaries** to the given QMenu **menu**

editBC(boundaries)
Opens a dialog which will change the boundary conditions of the **boundaries.**

Class ICEditTool

selectableObjects
Returns the mesh’s elements.

mouseMoveEvent(x,y)
Calculates whether or not the **xy** coordiantes are occupied by an element. If so it returns that element, if not returns None.

drawStuff(painter, element)
Paints the given **element** as transparent yellow using the given QPainter.

contextMenuActions(menu, slections)
Adds the option to edit initial conditions of the given **selections** to the given QMenu **menu.** Also adds the option to subdivide the given **selections.**

editIC(selections)
Opens a dialog which will change the initial conditions of the **selections.**

subdivide(elements)
Splits each given element in **elements** into four smaller pieces. Only works with rectangular elements currently.

Class SingleSelect

Methods

mousePressEvent(x,y,currentObject)
Selects the **currentObject** when called. If the Ctrl key is pressed appends the object to the current selections.

Class BoxSelect

Methods

mouseMoveEvent(event)
Called whenever the mouse is moved within this widget. While the left mouse button is pushed this method will calculate the box that will later be drawn by the drawStuff method.

mousePressEvent(event)
Stores this position as a corner of the rectangle to be drawn, clears the current selection list.

mouseReleaseEvent(event)
Causes this tool to stop drawing the box and saves the objects that were selected

selectionFilter(selectable)
Returns true if all the nodes within selectable are within the rectangle created by this tool. Any selectable object that meets these criteria should be selected while this tool is in use.

drawStuff(painter,*args,**kwargs)
While the mouse button is being pressed this method will draw a box onto the painter for visual feedback to the user.

Mesh.Generator.MeshGenerator.py

Class MeshGenerator

This class inherits from the Qt class QWidget. It is the main entry point into the mesh generation functionality. This widget will contain all the other widgets used, and so only this widget needs to be placed within the main window. The only functionality within this widget is to setup the layout and connect the data types of the GeneratorDisplay, NodeGenerator and EdgeGenerator classes. Eventually this class should be able to make a Mesh from scratch

Class GeneratorDisplay

This class inherits from the Qt class QWidget. It is used to provide visual feedback to the user while the mesh is being generated.

Attributes

Name	Type	Description	Documentation
nodes	List<Mesh.Nodes>	This is a reference to the list of nodes contained in the NodeGenerator class	Class Node Class NodeGenerator
edges	List<Edges>	This is a reference to the list of edges contained in the EdgeGenerator class	Class Edge Class EdgeGenerator

Signals

mouseMoved(str)

A signal that is emitted whenever the mouse is moved inside this widget. Contains a string which states the xy-coordinates of the current mouse position.

Methods

mouseMoveEvent(event)

Emits the mouseMoved signal whenever the mouse is moved.

nodeEdited(self)

Should be called whenever a node in the attribute nodes is edited somehow.

edgeEdited(self,edge)

Should be called whenever a edge in the attribute edges is edited somehow.

getRealXY(qx,qy)

Returns the real x and y coordinates based on the widgets internal coordinate system coordinates (given by **qx** and **qy**)

Mesh.Generator.NodeGenerator.py

Class NodeGenerator()

This class is used to allow the user to easily define nodes that will be used as the mesh's corners. Nodes implemented in the Mesh.py file are also used here. This class three main functionalities: adding nodes, deleting nodes, and reorganizing nodes.

Attributes

Name	Type	Description	Documentation
nodes	List<Mesh.Nodes>	A list of nodes to be used in the mesh.	Class Node

Signals

nodeEdited()

Emitted whenever a node in the nodes attribute is created, deleted, or changed in any way.

nodeDeleted()

Emitted whenever a node from nodes is deleted

Methods

addNode(x,y)

Adds a node to nodes at the location x, y, and makes the appropriate UI changes. Emites the nodeEdited signal.

delNode(index)

Deletes the node at the given **index** in nodes (with the first node having an index of 1). Emits both the nodeEdited and nodeDeleted signals.

swapNodes(index1, index2)

Swaps the indexes of the node at index1 and the node at index2. Emits the nodeEdited signal.

clearNodes()

Deletes all nodes in the attribute nodes.

Mesh.Generator.EdgeGenerator.py

Class EdgeGenerator

This class is used to add edges between nodes defined in NodeGenerator. These edges will act as the boundary of the mesh. This is class is designed to be able to add different types of edges such as straight lines or curves eventually. Currently only straight lines are implemented. This class adds the ability to add, delete, and edit existing edges.

Attributes

Name	Type	Description	Documentation
edges	List<Edges>	A list of edges to be used in the mesh.	Class Edge Class StraightLine
nodes	List<Mesh.Nodes>	This is a reference to the list of nodes contained in the NodeGenerator class	Class Node
edgeTypes	List<str>	String representations of the implemented edge types.	N/A

Signals

edgeEdited(Edge)

Emitted whenever an edge is edited, and contains the edge the was edited.

Methods

createEdge()

Creates the edge of the type specified in the UI using the nodes as specified in the UI.

edgeSelected(index)

Called whenever a new edge has been selected. Each edge has additional settings that can be used to change its properties and shape. This method will call the edge's settings method to retrieve a widget which will change the properties of the edge at the given **index**. Also takes signals returned by the settings method and connects them to the edgeEdited signal.

deleteEdge(index)

Deletes the edge at the given **index**.

saveEdges()

Saves data that has been specified in the edges' settings widget which will create new nodes and new edges. The protocol for this is specified in the Edge classes.

nodeDeleted(node)

should be called whenever a node is deleted. This method will delete any edges connected to that node

nodeEdited()

Should be called whenever a node is edited.

Class Edge
An abstract class that defines functionality that all edges must have.

Attributes

Name	Type	Description	Documentation
node1	Mesh.Node>	The first of two nodes that define an edge's endpoints.	Class Node
node2	Mesh.Node	The second of two nodes that define an edge's endpoints.	Class Node
scales	func	a function to retrieve scales used by the display widget to draw this edge properly	N/A
Current	Bool	Indicates if the edge is currently selected by the user.	N/A

Methods

scaleNodes()
Returns node1 and node2's coordinates using the display widget's internal coordinate system.

draw(painter)
Draws the edge using the QPainter **painter**

settings()
Returns a QWidget that can be used to further customize an edge, as well as a list of signals that the QWidget can emit which would require updates in the display panel.

lerp(x1,y1,x2,y2,t)
Linear interpretation between two points using the parameter t.

Class StraightLine
Currently the only class that inherits from the Edge class.

Attributes

Name	Type	Description	Documentation
detail	int	Additional nodes will be added along the edge. The amount of nodes is based on this value	N/A

Methods

draw(painter)
Draws a straight line between node1 and node2, with additional nodes being placed along the line based on the detail attribute. These nodes are temporary until split is called.

settings()
Returns a widget with a slider that controls the detail attribute.

angle()
Returns the angle in radians of this line relative to x-axis

split()
Creates new node where previously temporary nodes had been placed. Then places multiple edges connecting all of them.