

Myanmar Institute of Information Technology
I Semester 2019-2020
CSE 5000 Capstone Project

**GUI enhancing for OpenModelica chemical process
simulator and Foundation work on PFD application**

2015-MIIT-CSE-055
Thiha Min Maung

Submitted in Partial Fulfilment of the Course CSE 5000 Capstone Project

and

Requirements of BE (Hons.) Degree in Computer Science & Engineering

at

FOSSEE, IIT Bombay
Powai, Mumbai, Maharashtra 400076, India.



Self-Declaration by the Student

I hereby certify that this Final Report submitted by me in partial fulfilment of the course CSE 5000 Capstone Project in I Semester 2019-2020 embodies work done by me at **FOSSEE, IIT Bombay** under the supervision of **Prof. Kannan Moudgalya, Principal Investigator**.

I certify that this report has been written by me, and all text, figures, tables, etc. that are not my original contributions have been appropriately acknowledged, and the sources cited clearly.

I also certify that the report submitted is free of any plagiarisms and/or violations of academic ethics. I further state that I shall be solely responsible for consequences if any transgressions of academic ethics are found in this report.



Thiha Min Maung
2015-MIIT-CSE-055

Certificate

This is to certify that this report **GUI enhancing for OpenModelica chemical process simulator and Foundation work on PFD application** submitted by **2015-MIIT-CSE-055, Thiha Min Maung** done under the supervision of **Prof. Kannan Moudgalya, Principal Investigator, FOSSEE, IIT Bombay** embodies work done by the student as part of the course CSE 5000 Capstone Project in I Semester 2019-2020, and towards fulfilment of the partial requirements of the BE (Hons.) Degree in Computer Science & Engineering.

(Signature of Supervisor)

Name Prof. Kannan Moudgalya
Title Principal Investigator
Organization FOSSEE, IIT Bombay
Date

(Signature of Rector)

Name Dr. Win Aye
Title Rector
Organization MIIT
Date

Table of Contents

ACKNOWLEDGEMENTS.....	6
INTRODUCTION	7
<i>TITLE OF PROJECT.....</i>	<i>7</i>
<i>DESCRIPTION OF PROJECT</i>	<i>7</i>
<i>PROJECT OBJECTIVES.....</i>	<i>8</i>
<i>SCOPE OF PROJECT</i>	<i>8</i>
<i>METHODOLOGY.....</i>	<i>9</i>
OPENMODELICA	9
PYTHON.....	10
PYQT5	10
GIT	10
SOFTWARE DEVELOPMENT LIFE CYCLE MODEL	10
CHAPTER 1 ENHANCEMENTS AND NEW FEATURES IMPLEMENTATIONS	11
1.1 <i>ENHANCEMENTS</i>	<i>12</i>
1.1.1 DECOUPLING GRAPHIC PARTS	12
1.1.2 OPTIMIZATION OF CODE STRUCTURES IN ACCORDANCE WITH OOP CONCEPTS AND SOLID PRINCIPLES	12
1.1.3 PARTIALLY INTERACTIVE NODE LINES	12
1.2 <i>NEW FEATURES.....</i>	<i>14</i>
1.2.1 SAVING THE CANVAS – “SAVE”	14
1.2.2 OPENING THE SAVED CANVAS – “OPEN”	14
1.2.3 DIRECTORY PATH WINDOWS FOR “SAVE” AND “OPEN”	14
CHAPTER 2 REVAMPED UNIT OPERATION CLASSES AND THEIR DOCK WIDGETS	16
2.1 <i>CLASS STRUCTURES MODIFICATION.....</i>	<i>16</i>
2.1.1 NEEDS FOR STORING FETCHED DATA FROM RESULT CSV	16
2.1.2 DESIGN STRUCTURE FOR UNIT OPERATION CLASS VARIABLES.....	16

2.2	<i>REVAMPED UI FOR INPUT DATA</i>	17
2.2.1	MERGING DOCKWIDGET AND RESDOCKWIDGETS INTO ONE.....	17
2.2.2	MODIFIED DOCKWIDGET UI – WITH MODE SELECTION	17
2.2.3	MODIFIED DOCKWIDGET UI – WITHOUT MODE SELECTION	19
2.3	<i>NEW UI FOR RESULTS</i>	21
2.3.1	FETCHED RESULT FOR WITH-MODES CLASSES	21
2.3.2	FETCHED RESULT FOR MATERIAL STREAM CLASS	22
2.4	<i>MISCELLANEOUS</i>	23
2.4.1	NODE ITEMS POSITIONING ERRORS FIXATION	23
2.4.2	DOCK WIDGETS POSITIONING ERRORS FIXATION	24
2.4.3	STANDARD NAMING CONVENTION FOR CLASS VARIABLES	25
CHAPTER 3	PROCESS FLOW DIAGRAMMING PFD APPLICATION AS A BY-PRODUCT	26
3.1	<i>INITIAL VERSION</i>	26
3.1.1	BASIC GUI	26
3.1.2	INTERACTIVE NODE LINE WITH ARROW HEAD	26
3.1.3	INTERACTIVE NODE SOCKETS	27
3.1.4	“EDITING” AND “PREVIEW” MODES.....	28
3.1.5	SAVING THE CANVAS AS AN IMAGE	28
3.1.6	MISCELLANEOUS	29
3.2	<i>FINAL VERSION</i>	30
3.2.1	USING SVG FOR BETTER RESOLUTION AND MANIPULATION OF NODE ITEMS.....	30
3.2.2	FULLY INTERACTIVE NODE LINES.....	30
3.2.3	EDITABLE NODE LINES USING NODE ANCHORS	31
3.2.4	NEW NODE SOCKETS IMPLEMENTATION.....	32
	CONCLUSION	343
	DIRECTIONS FOR FUTURE WORK	34
	REFERENCES / BIBLIOGRAPHY	35

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude towards Prof. Kannan Moudgalya for giving me a chance to do my final B.E. Capstone project at FOSSEE, IIT Bombay. It was such a wonderful once-in-a-lifetime experience to work at IIT Bombay and live in vibrant Mumbai city.

Secondly, I would like to thank Sir Pravin Dalve for mentoring and helping me throughout the whole time during this internship. I gained a lot of invaluable insights and motivational strength by working with him.

Thirdly, I also would like to thank Sir Priyam Nayak and ma'am Usha Viswanathan for all the administrative works and their hospitalities.

And last but not least, I would like to express my deepest appreciation to Prof. Raja Subramanian, Prof. Joe Tun Sein, Prof. Usha Subramanian and Ma Thae Thae Htwe from my mother univeristy MIIT for the arrangement of great internships to all of us.

INTRODUCTION

OpenModelica is an open source Modelica-based modeling and simulation environment intended for industrial and academic use. It is developed by a non-profit organization: the Open Source Modelica Consortium (OSMC). It is one of FOSSEE projects (Free/Libra Open Source Software for Education and Engineering). OpenModelica team of FOSSEE, IIT Bombay has developed a library package that can be used for the purpose of chemical process simulation in OpenModelica.

TITLE OF PROJECT

GUI enhancing for OpenModelica chemical process simulator and
Foundation work on PFD application

DESCRIPTION OF PROJECT

A typical process of creating chemical simulation requires the knowledge of how to use the OpenModelica and learn the syntax of writing code correctly that implements the library and creates flow sheet model. In other words, in order to create a flow sheet in OpenModelica, few lines of codes have to be written in OpenModelica following the correct syntax. In essence, the user needs to have a good knowledge of Modelica language and understanding of the chemical library in order to perform chemical process simulation in OpenModelica.

For the uninitiated, a typical example of chemical process simulation in creating a process flow sheet involves the following steps:

- Selection of compounds required for simulation from compound database
- Selection of thermodynamic package required for the simulation
- Selection of specific unit operations required to build the flow sheet
- Connect the unit operations in the required sequence to build the flow sheet
- Provide input variables to the unit operations to obtain desired result

The main essence of this project is to provide an easy-to-use drag and drop chemical simulation software which allows a user to do simulation without any computing background. This will eliminate the burden for learning extra coding lessons and save a lot of time for all chemical engineering students and users so that they can fully focus only on their simulations.

PROJECT OBJECTIVES

The previous fellow intern students have already implemented a basic working version of this chemical simulation software. But there are still plenty of rooms for improvement. The main problem with the existing implementation was that the codes were tightly coupled. The front-end and back-end parts were intermingling with each other. The graphic parts needed a lot of improvements and maintenance e.g. default node lines, dock widgets' positions, graphic items' positions when respective dock widget was invoked. Besides, new features which are necessary and vital for a software application have to be implemented.

To put it in a nutshell, the main objectives of this project are to enhance the overall structure and Graphical User Interface for the chemical simulation software OpenModelica and create additional features on top of existing ones.

SCOPE OF PROJECT

We aim to achieve a complete easy-to-use GUI for the simulation process with the help of OpenModelica API and at the same time, implement some important features. These include saving the current simulation and graphic canvas, reloading previous simulations, and the undo redo features etc. We aim to decouple the tightly bonded codes in accordance with OOP concepts so that everyone working in future will have a clear understanding quickly and make changes and/or improvements easily.

The python language has been used to create the whole system with PyQt5 (Qt framework for python). The existing implementation of the GUI is integrated with the python API to interact with the OpenModelica Chemical Simulator library made by the FOSSEE team. The very first approach toward this project is to understand the existing implementation and how the classes interact with one another. Then we will move to improve the overall structure step by step, make changes and implement new features.

METHODOLOGY

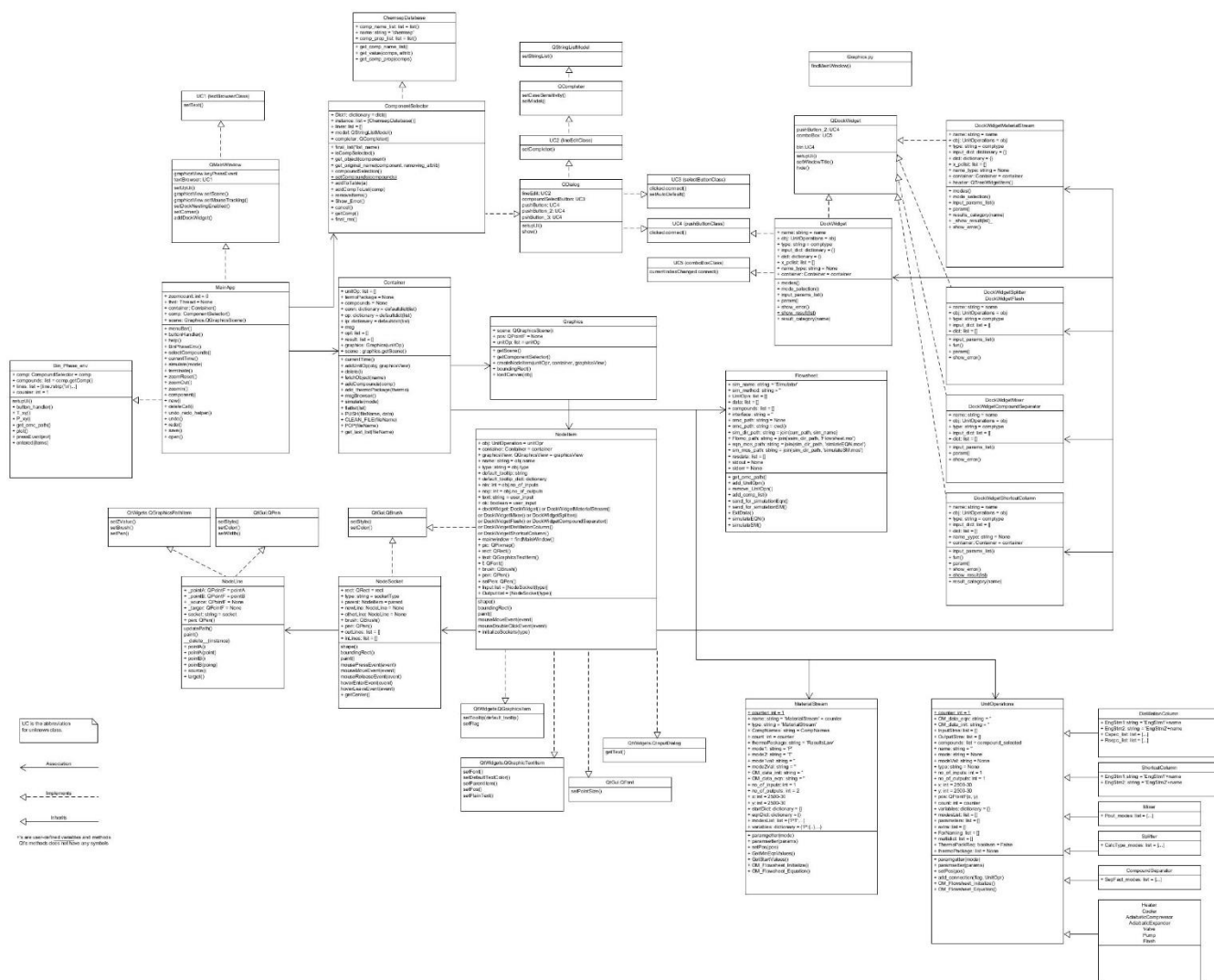


Figure: Class Diagram of OpenModelica Chemical Process Simulator
https://github.com/pravindalve/Chemical-Simulator-GUI/blob/master/class_diagram.pdf

Since this chemical simulator is an Open Source software, all technologies used in creating this software are also open source technologies. The various technologies that have been used for the development and the enhancement of the GUI includes:

OPENMODELICA

OpenModelica is a free and open source environment based on the Modelica modeling language for modeling, simulating, optimizing and analyzing complex dynamic systems. This software is actively developed by Open Source Modelica Consortium, a non-profit, non-government organization.

The various files required for performing the simulation has been coded using OpenModelica. The OpenModelica Compiler (OMC) has been used for compiling the *.mo* Flowsheet files generated by the GUI simulation. This internally does the calculations and generates the

desired output, which is then shown to the user using the GUI. Also, the *.mos* script file generated by the Python powered GUI is ran using the OpenModelica platform.

PYTHON

Python is an open source programming language which was made to be easy-to-read and powerful. Python is an interpreted, high-level, general-purpose programming language. Python allows faster development of projects in many fields ranging from web applications and machine learning to IoT devices. Also, as it is quite versatile, it has great community support which provides solutions to almost all known problems encountered during development. We have used python version 3 in this project along with various library packages.

PYQT5

PyQt is a Python binding of the cross-platform GUI toolkit Qt, implemented as a Python plugin. It is free software developed by the British firm Riverbank Computing.

PyQt version 5 is being used for the development of this project. The front-end created is powered by PyQt5 which connects its various actions and triggers the processing required at various stages. Many widgets of PyQt5 which are available as packages have also been actively used for this project.

GIT

Git, coupled with GitHub, has been used for version control in this project. This allows for tracking and identifying changes in the source code during the process of software development. It also helps in enabling many programmers to work together on the same piece of code collaboratively.

SOFTWARE DEVELOPMENT LIFE CYCLE MODEL

We have utilized Agile software development life cycle model in this project. According to Wikipedia, Agile software development comprises various approaches to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customers. It advocates adaptive planning, evolutionary development, early delivery, and continual improvement, and it encourages rapid and flexible response to change.

In the entire work duration, my supervisor Sir Pravin Dalve acted as both customer and team leader. We have at least two scrum meetings every day, one at the start of the day and the other at the end. We share our status and identify potential issues in these meetings. In morning, we discuss what to do in this day i.e. what are the requirement specifications from customer perspectives, explore the possible ways to tackle them and divide work among team members. At the end of the day, we talk about what have been completed today, review them and decide what to do in the next days.

CHAPTER 1 ENHANCEMENTS AND NEW FEATURES IMPLEMENTATIONS

In this chapter, all enhancements and new features which have been implemented will be discussed. There will be two parts in this chapter. While part one will be dealing with all the enhancements which try to improve overall existing code structure, part two will be focused on implementation of new features which are vital to an application.

Firstly, the main UI screen has been changed to the new look with better layout, font and color consistency in order to achieve visually appealing minimalist user interface.

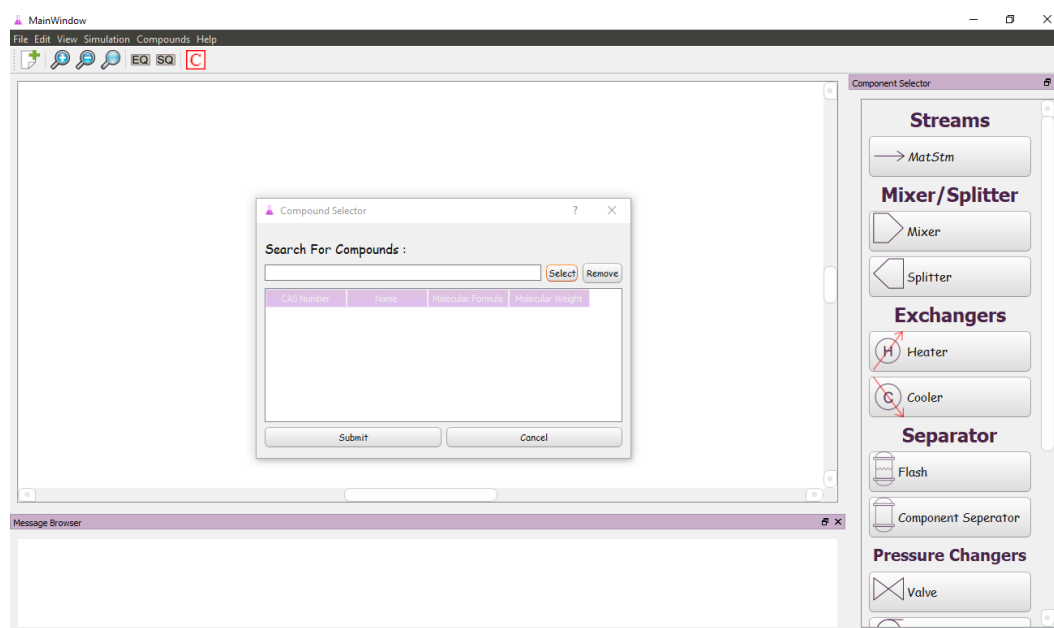


Figure 1(a): The old main screen UI

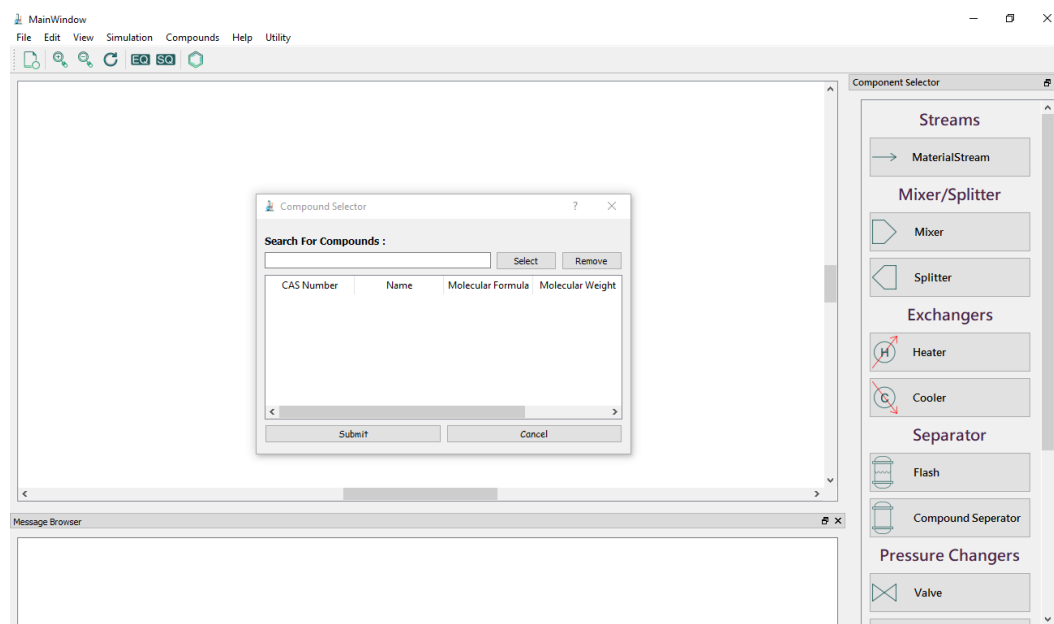


Figure 1(b): The new main screen UI

1.1 ENHANCEMENTS

1.1.1 DECOUPLING GRAPHIC PARTS

The existing implementation was in a way that the graphic parts have been intermingling with the backend parts which is not a desirable design architecture. Therefore all front end related classes have been taken out from the *mainApp.py* and gather in new module *Graphics.py* with a new “Graphics” controller class. This controller class helps in creating the graphic Node Items and add them to the main scene. The controller class is also responsible for recreation of Node Items when “Open” feature get called. This will later be explained in section 1.2.2.

1.1.2 OPTIMIZATION OF CODE STRUCTURES IN ACCORDANCE WITH OOP CONCEPTS AND SOLID

PRINCIPLES

In the previous GUI implementation, the system structure was tightly coupled and there were so much dependencies among classes. Therefore a small change would have made a huge impact to the entire structure and would either lead a program to crash or many consequence changes would have to be made for that tiny change. “Container” class is responsible for acting as an intermediate class between front end and back end classes. The new structure is in a way such that whenever “New” button get clicked, it will create a new flowsheet with a new container object. All other related objects and attributes will be stored and taken care only by that single container object. Similarly, it also applies for Deleting.

1.1.3 PARTIALLY INTERACTIVE NODE LINES

In previous existing implementation, the Node Line between any two Node Items was always in default path. Every Node Line appears as only in “Z” or “reverse Z” shape. It will not look good when Node Item positions get changed. Refer to the below diagram, the default path is when Node Item A is in the left side and Node Item B is in the right. When Node Item B is moved beyond to the left of Node Item A, the Node Line will looks like that it is coming from the input of Node A (instead of A output) and it is going inside the output of Node Item B (instead of B input).

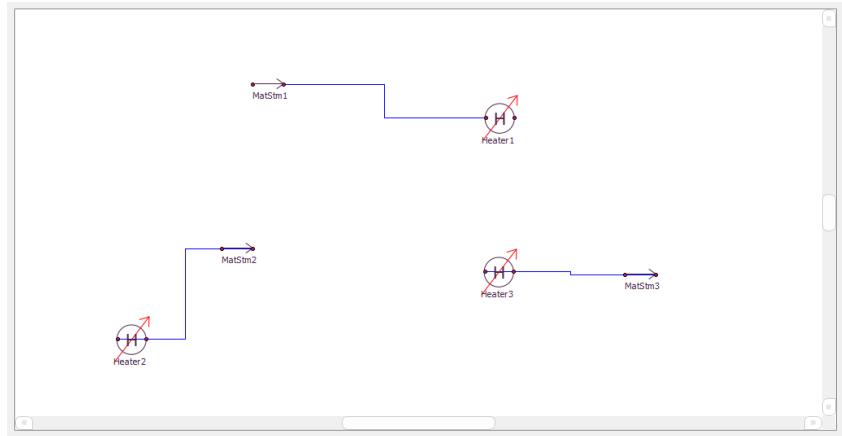


Figure 1.1.3(a): Before enhancing Node Line paths with issues

To overcome this problem, more Node Line paths have to be coded manually. When B is to the right of A, the default path will follow. When B is to the left of A, a new path (decimal shape of number 2) will follow. When B is to the left A beyond a certain distance, another new path (loop shape) will appear interactively based on the positions of two Node Items. The implementation fulfill the correct connection between inputs and outputs. (A Node Line which appears coming from a Node Item output could only goes inside the input of another Node Item.) The coding implementation can be found at “NodeLine” class method *updatePath(self)*. The commits of codes to the git can be found in References.

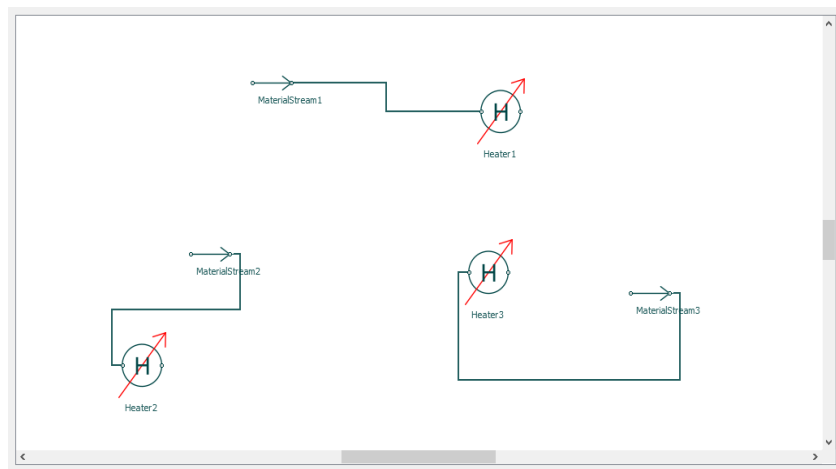


Figure 1.1.3(b): After enhancing Node Line paths to partially interactive lines

1.2 NEW FEATURES

1.2.1 SAVING THE CANVAS – “SAVE”

“Save” can be regarded as the most crucial feature of any application. In this case, it is to save the current simulation canvas as they are. For every created Node Item, there are two objects related to them. One is the graphic object which will be displayed on the canvas and the other one is the backend object which will store all the attributes related to that Node Item. Whenever any Node Item is moved to any place, its position will always be saved in its attribute in scene coordinates. The connections of Node Lines between Node Items will also keep track of in their respective attributes.

By saving, it also means storing the data values inside backend objects but not in the graphic objects. When a user hits the “Save” button or presses the short cut key, all Node Item backend objects and all selected compounds which are stored inside the list of “Container” object will be pickled as *.sim* files.

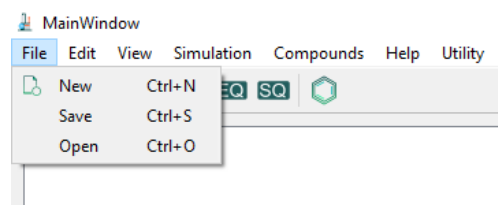


Figure 1.2.1: “Save” and “Open” action short cut keys

1.2.2 OPENING THE SAVED CANVAS – “OPEN”

Another ubiquitous feature of any software application is “Open”. This action will reload the previously saved canvas and recreate the whole canvas exactly as when it was saved. Since backend objects have been saved, it will iterate through those objects and instantiate respective graphic part of Node Item object based on the stored attributes’ values.

Recreating the Node Lines between Node Items is a bit tricky. A lot of effort have been put to make the Node Lines appears as they were and enable them to be interactive as before without failing other tiny subtle features. The coding implementations can be found at “MainApp” class method *open(self)* and “Graphics” class method *load_canvas(self, object, container)*.

1.2.3 DIRECTORY PATH WINDOWS FOR “SAVE” AND “OPEN”

When a user clicks “Save” or “Open” buttons or presses short cut keys, respective window which deals with file system will pop up. The file can be saved at any location of the whole computer system and also can be retrieved from any system path. It has also taken care of the fact that the program will not crush when “Cancel” button get clicked.

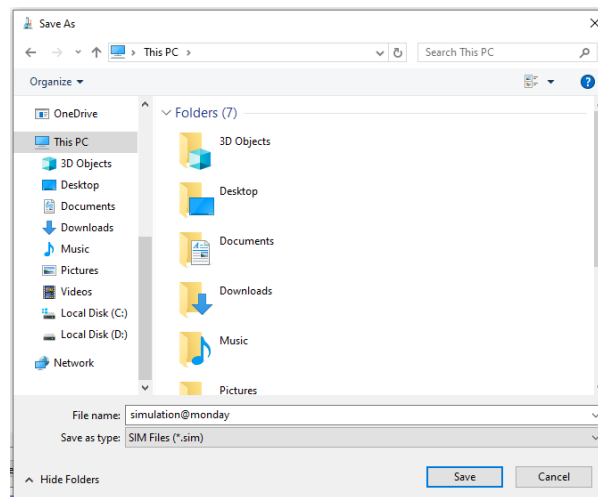


Figure 1.2.3(a): Directory path windows for saving files

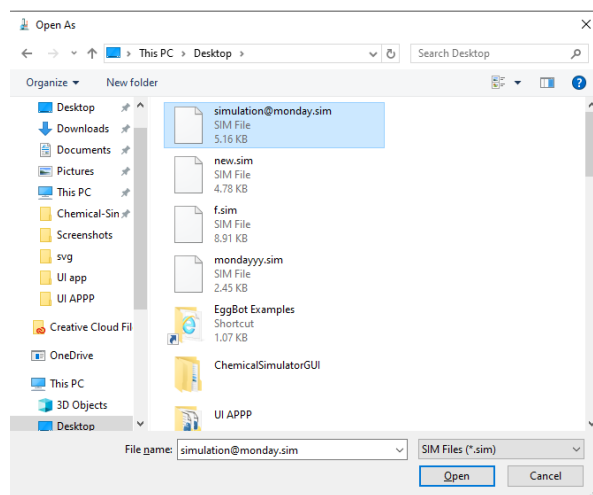


Figure 1.2.3(b): Directory path windows for opening files

CHAPTER 2 REVAMPED UNIT OPERATION CLASSES AND THEIR DOCK WIDGETS

In this chapter, the newly revamped graphical user interfaces of Unit Operation classes will be discussed. This chapter also contains two main sections. In the first section, it will explain the motivations of why new design structure is needed to change the existing implementation. The second section will deal with different kinds of unique user interfaces for different unit operation classes.

2.1 CLASS STRUCTURES MODIFICATION

2.1.1 NEEDS FOR STORING FETCHED DATA FROM RESULT CSV

Once a model is created in OpenModelica, the solution is achieved through Equation Oriented Mode only. This creates a problem in getting the final solution to converge where the system involves solving large numbers of nonlinear equations. After successful simulation, OpenModelica provides a tree structure of viewing the result variables with not a lot of explanation of what they might be or belong to. Chemical simulation of dynamic nature requires a Sequential Oriented Mode, wherein each unit operation must be simulated in isolation with respect to the other. To continue the simulation, the result values are required to be stored inside unit operation object. Refer to Section 2.3.1 for further explanation.

2.1.2 DESIGN STRUCTURE FOR UNIT OPERATION CLASS VARIABLES

In the existing implementation, the variables are declared literally as they were. In chemical simulation, all variables have full names, short name, default values and units. Unit conversion is also a feature which is yet to be implemented. Apart from values, all other attributes are new implementation to fulfill the requirements. All these can be stored with better readability by using dictionary data structure. There are altogether 13 unit operations with different variables. Design snippet of the sample unit operation Heater is shown below.

```
431 class Heater(UnitOperation):
432
433     def __init__(self, name='Heater'):
434         UnitOperation.__init__(self)
435         self.name = name + str(type(self).counter)
436         self.type = 'Heater'
437         self.no_of_inputs = 1
438         self.no_of_outputs = 1
439         self.modes_list = ['Q', 'Tout', 'xvapout', 'Tdel']
440         self.parameters = ['Pdel', 'Eff']
441         self.extra = None
442         self.for_naming = None
443         type(self).counter += 1
444
445         self.variables = {
446             'Pdel' : {'name':'Pressure Drop',          'value':0,      'unit':'Pa'},
447             'Eff'   : {'name':'Efficiency',             'value':1,      'unit':''},
448             'Tout'  : {'name':'Outlet Temperature',    'value':298.15, 'unit':'K'},
449             'Tdel'   : {'name':'Temperature Increase', 'value':0,      'unit':'K'},
450             'Q'      : {'name':'Heat Added',           'value':0,      'unit':'W'},
451             'xvapout': {'name':'Outlet Vapour',        'value':None,   'unit':''}
452         }
453
```

Figure 2.1.2: Design structure for heater class

2.2 REVAMPED UI FOR INPUT DATA

2.2.1 MERGING DOCKWIDGET AND RESDOCKWIDGETS INTO ONE

In new implementation, the old DockWidget (dock widget of accepting input values) and resDockWidgets (dock widget for storing the result values) are getting merged into only one widget called DockWidget. In the old design, all unit operations share the same generic UI which is not suitable for without-parameter unit operations since they have different attributes and natures. These will be explained in later sections.

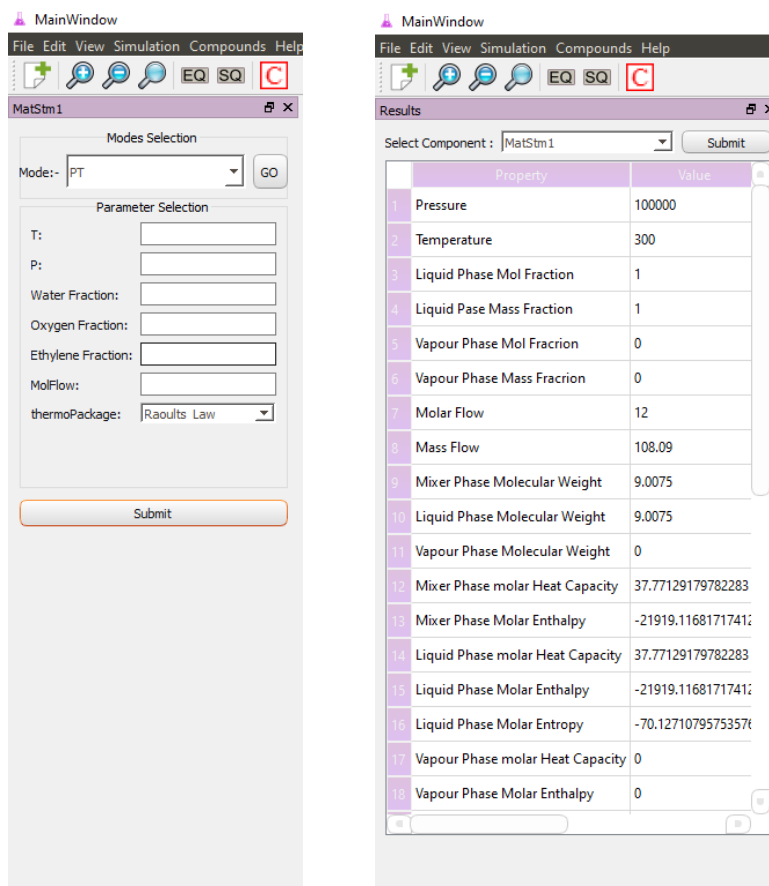


Figure 2.2.1: Previous DockWidget and resDockWidget with generic UI

2.2.2 MODIFIED DOCKWIDGET UI – WITH MODE SELECTION

Some unit operations have the so-called modes in which a user has to select one of the modes and continue simulation. Each unit operation has its own unique modes. Based on the selected modes, the parameters have to be filled. Initially, there will be default constant values occupied. If other values are desired, it has to be changed. For Material Stream class, there is an extra field called Mole Fractions. These are the mole fractions of the selected compounds. After filling all values, their sum will be normalized to 1.

The unit operation classes which have mode selection are:

- (1) Material Stream
- (2) Heater
- (3) Cooler
- (4) Adiabatic Compressor
- (5) Adiabatic Condenser
- (6) Value
- (7) Pump

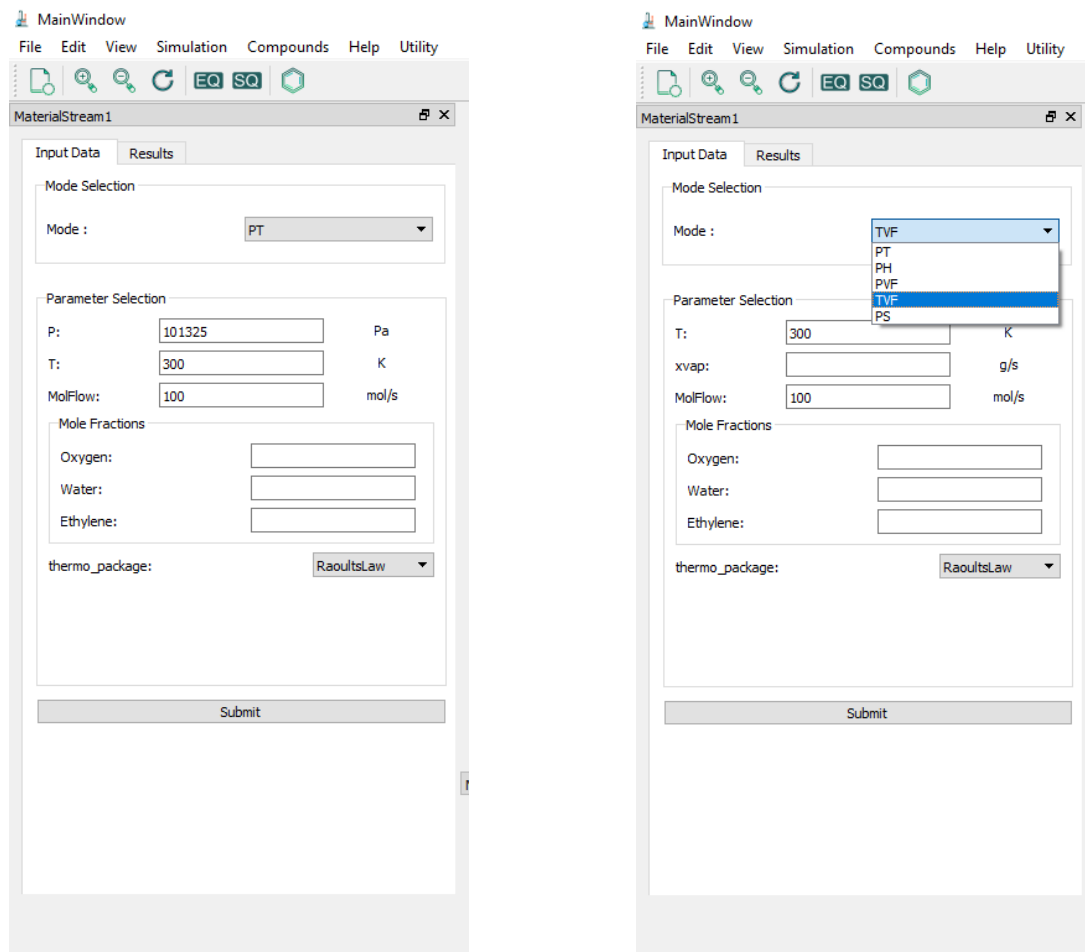


Figure 2.2.2(a): New DockWidget UI for Material Stream

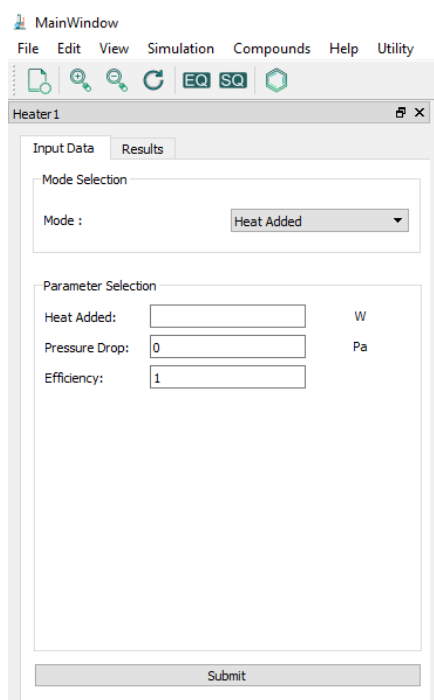
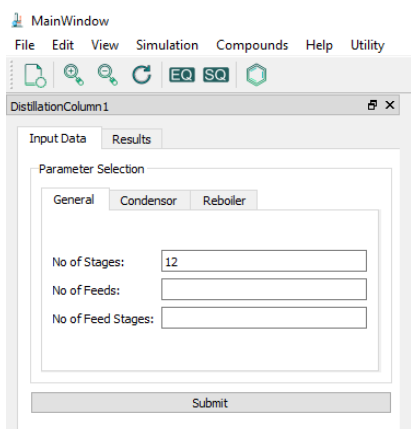


Figure 2.2.2(b): New DockWidget UI for Heater

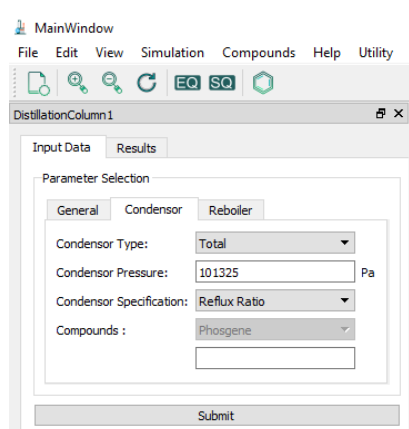
2.2.3 MODIFIED DOCKWIDGET UI – WITHOUT MODE SELECTION

There are unit operations which do not have mode selections. Their attributes are quite different from each other and therefore require unique UI. Generally, their unit operations may or may not have results. The result sections are also yet to be implemented in the future. The following figures represent two of these unit operations. The classes without mode selection are as follows.

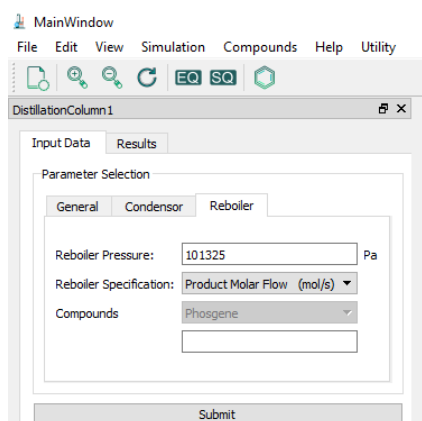
- (1) Mixer
- (2) Splitter
- (3) Flash
- (4) Compound Separator
- (5) Distillation Column
- (6) Separation Column



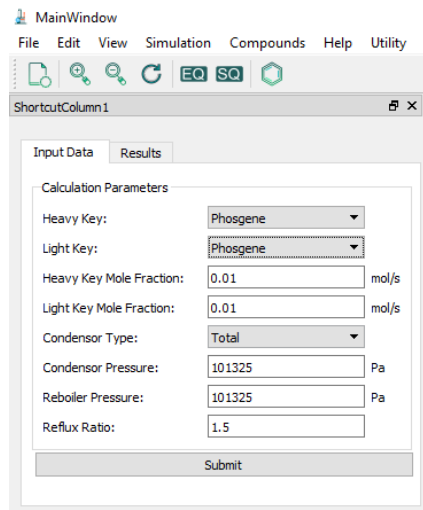
(a)



(b)



(c)



(d)

Figure 2.2.3 (a, b, c): New DockWidget UI for Distillation Column

Figure 2.2.3 (d): New DockWidget UI for Shortcut Column

2.3 NEW UI FOR RESULTS

2.3.1 FETCHED RESULT FOR WITH-MODES CLASSES

After simulation had successfully completed, OpenModelica yields a CSV file which contains the calculated result from simulation. The results are just a list of variables and values without the explanation of what they represent. That is why proper design structure is required to manipulate these values as mentioned in section 2.1.1. and 2.1.2.. Based on the selected unit operation, only its results should be displayed in the result DockWidget with proper names and units.

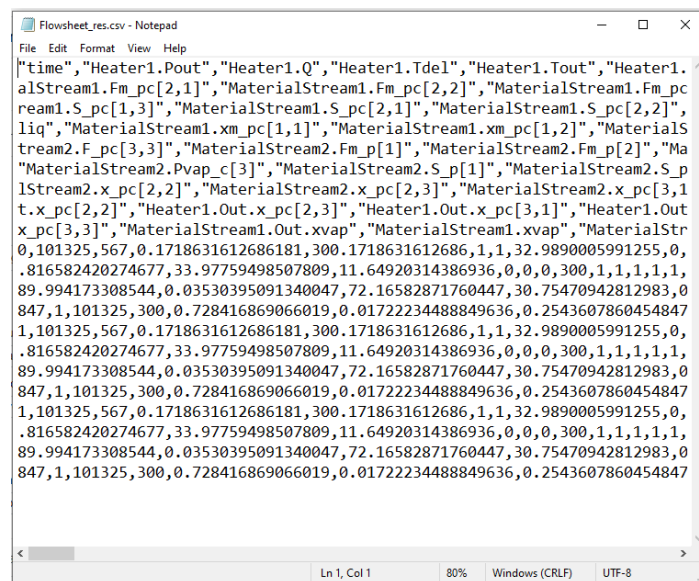


Figure 2.3.1(a): Result CSV file produced by OpenModelica

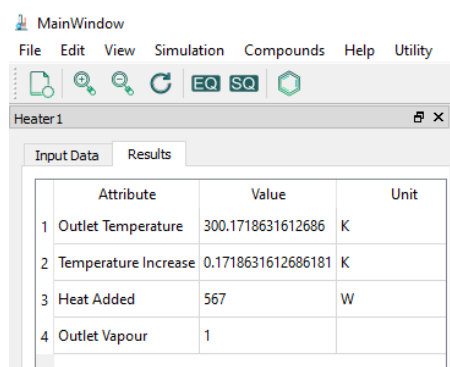


Figure 4.3.1.2(b): Result DockWidget UI for Heater

2.3.2 FETCHED RESULT FOR MATERIAL STREAM CLASS

The result of Material Stream is moderately different from those unit operation classes with mode selection. It has its own user interface with more intricate attributes than others. Three basic phases are Mixture, Liquid and Vapour. Each phase has its own amounts and phase properties. Each amount, in turn, has four parts namely Mole Fraction, Mass Fraction, Mole Flow and Mass Flow. In subsequence, there will be all selected compounds with their values and units in each part.

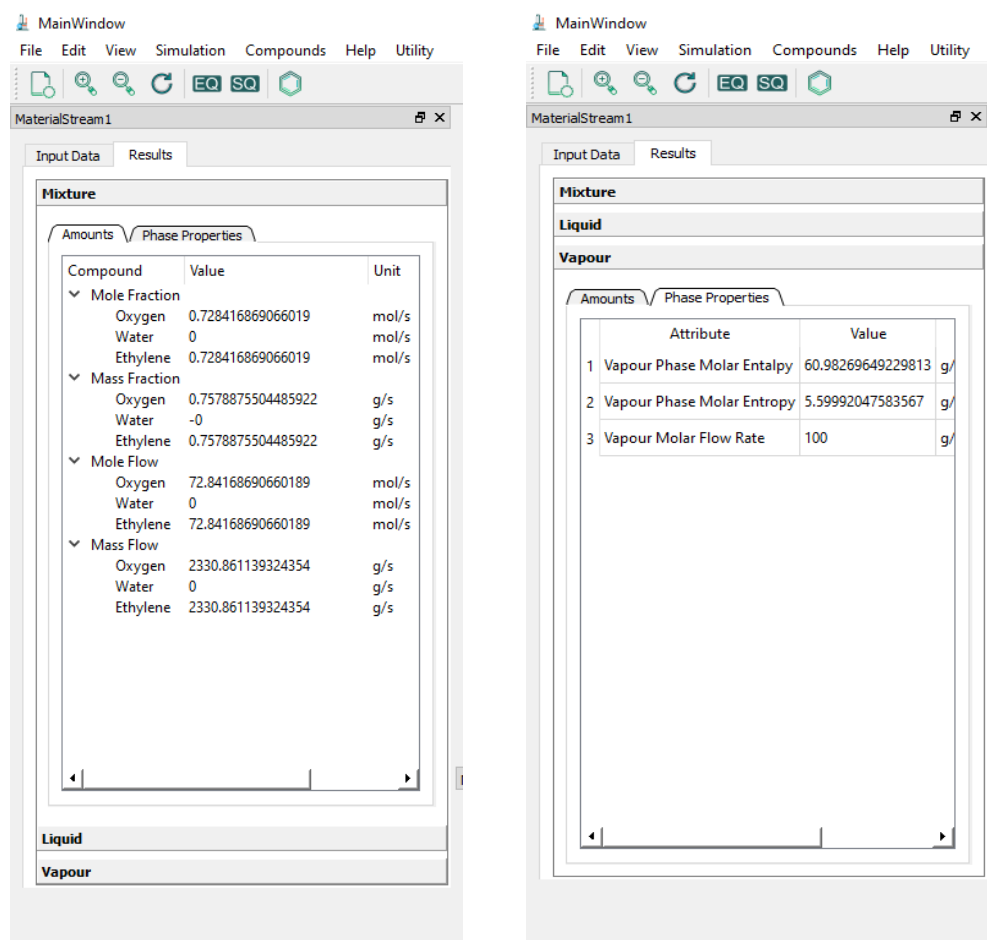


Figure 2.3.2: Result DockWidgets UI for Material Stream

2.4 MISCELLANEOUS

2.4.1 NODE ITEMS POSITIONING ERRORS FIXATION

There raises a problem when a Node Item is double-clicked. When a double-click action is triggered, a Dock Widget will be appeared at the left side of the canvas. It is accompanied by an unwanted issue in the graphic canvas. The issue is that the clicked Node Item and its Node Lines' geometry changes. It will then be resolved when a mouse-move event is done on that Node Item. This issue is completely tackled and no extra mouse-move event is ever required to restore the original geometry.

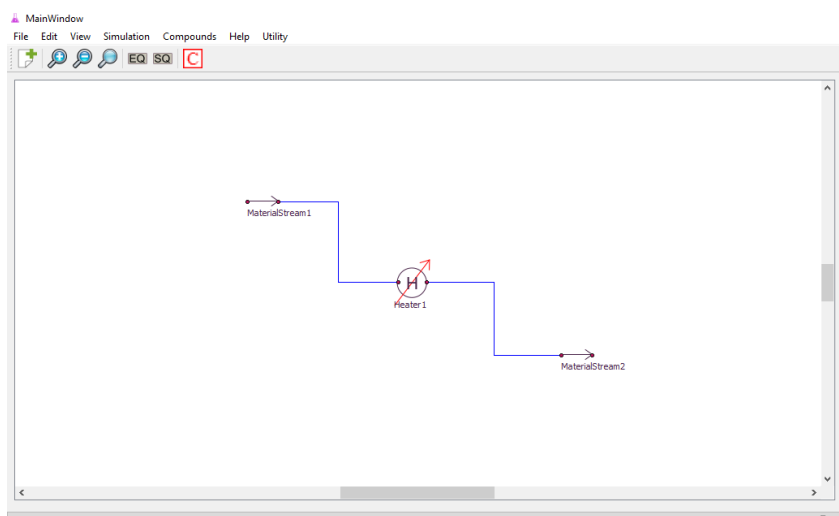


Figure 2.4.1(a): Original simulation graphic canvas

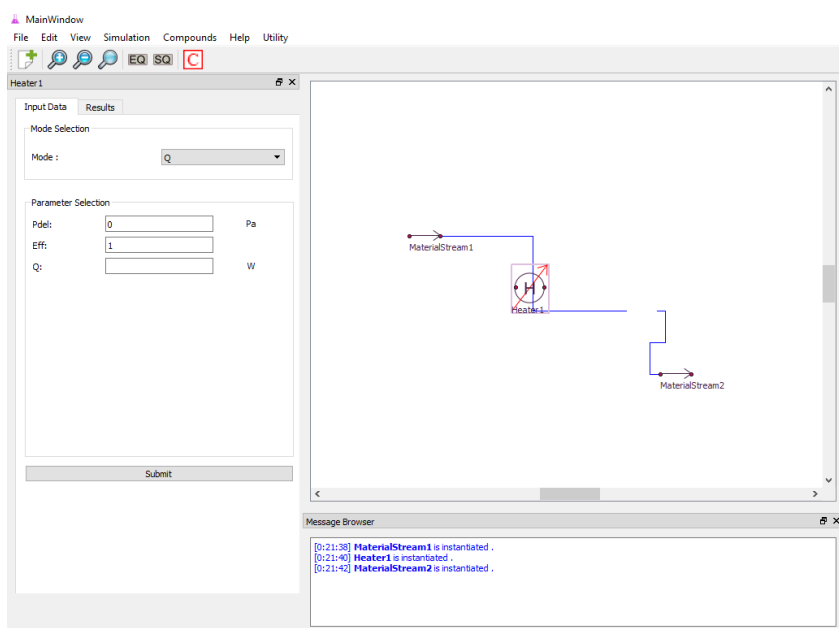


Figure 2.4.1(b): Node Items positioning error when a double-click action is triggered

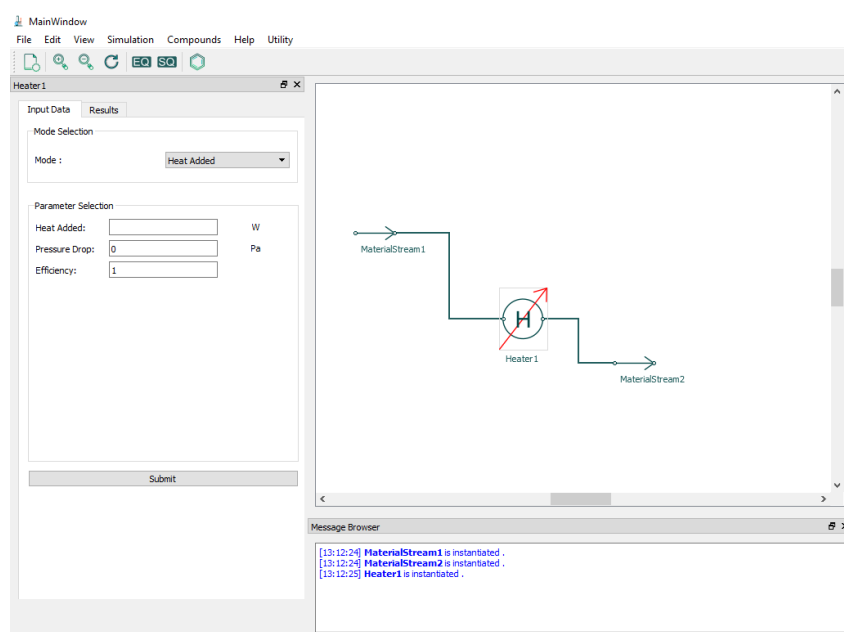


Figure 2.4.1(c): Fixed Node Items positioning error

2.4.2 DOCK WIDGETS POSITIONING ERRORS FIXATION

Another issue which is related to the above section is that when another Node Item is double-clicked, not only its graphic geometry but also the whole dock widgets' positions get changed as shown in below figure. This dock widgets positioning errors have already been solved in such a way that all dock widgets and graphic components will stay the same throughout the whole simulation. The Dock Widget will also get deleted whenever its graphic Node Item get removed from the scene.

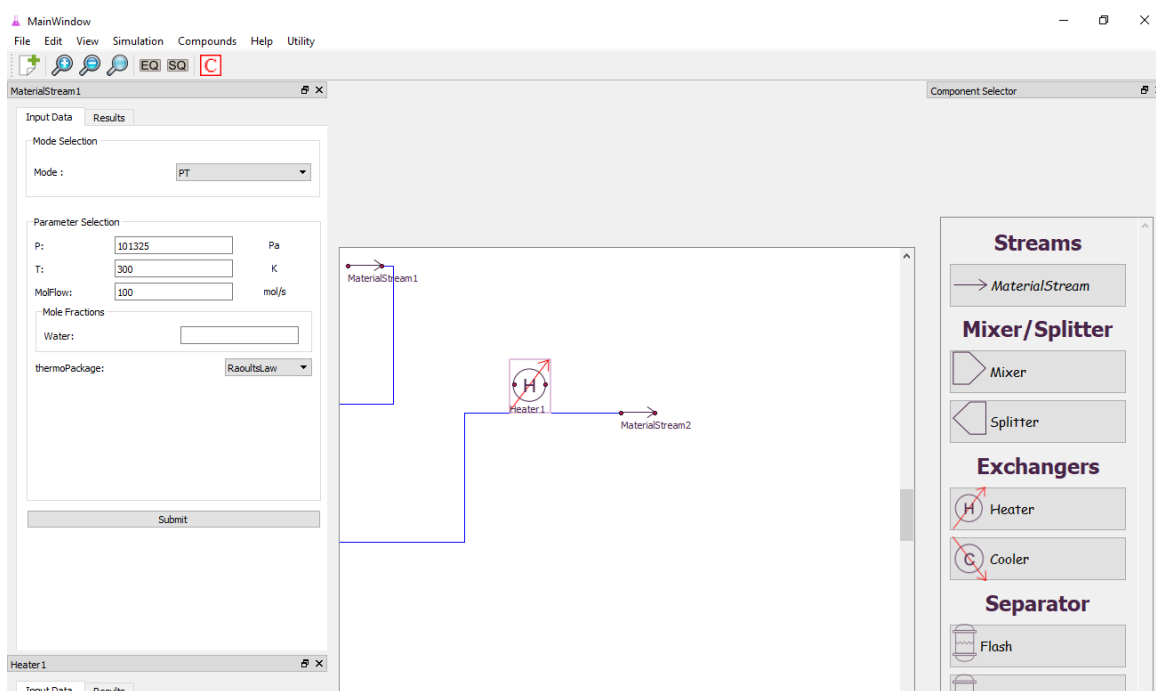


Figure 2.4.1(a): Dock Widgets positioning error

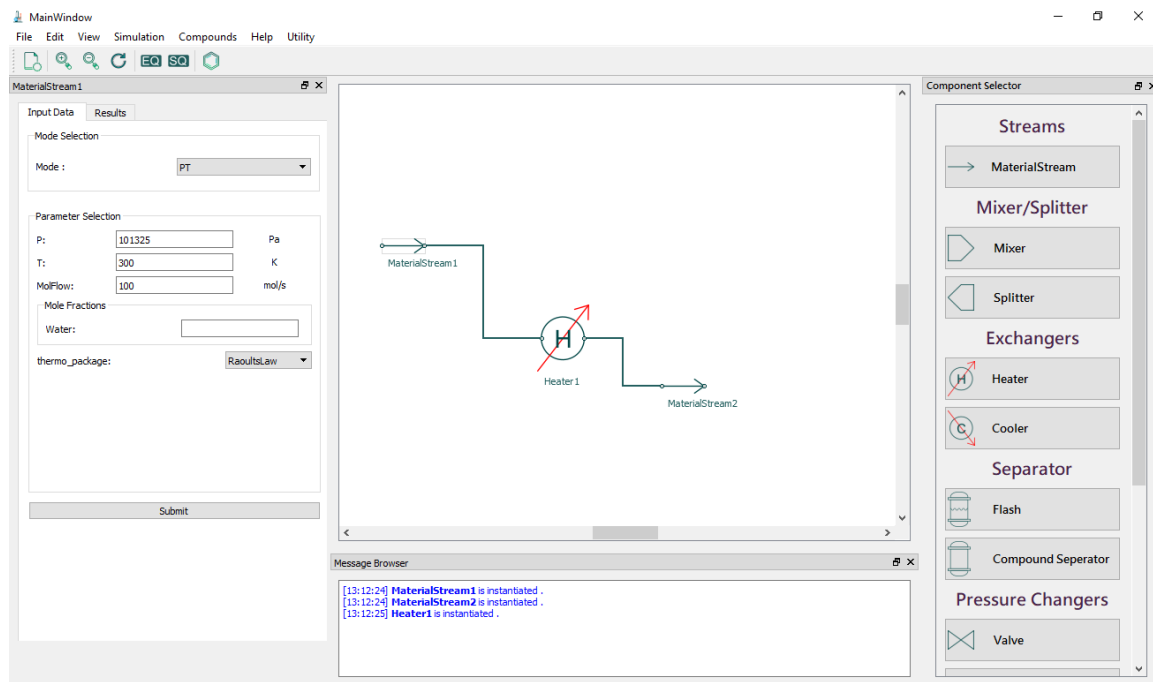


Figure 2.4.1(b): Fixed Dock Widgets positioning error

2.4.3 STANDARD NAMING CONVENTION FOR CLASS VARIABLES

Since this project was and is carried out by different fellow internship students, there is an issue in the variable names as different people refer using their own favorite style of naming convention. The whole program variables and method names have been changed to the standardized naming conventions. In future development, it is advised to use the python hyphen style convention for self-defined variables and methods along with C++ camelCase style for default PyQt5 built-in methods.

Also, the whole class diagram has been updated with new implementations and standardized variable and class names.

CHAPTER 3 PROCESS FLOW DIAGRAMMING PFD APPLICATION AS A BY-PRODUCT

In this section, the foundation work on Process Flow Diagramming PFD application will be discussed. Initially, this was not part of the original work plan. However, as per requirements, two types of diagramming applications were developed.

3.1 INITIAL VERSION

3.1.1 BASIC GUI

The very first version is to simply taking out the graphic parts of the chemical process simulator and make new independent diagramming application. The main objective of the new application is to help designing new chemical process more quickly and smoothly without worrying the simulation.

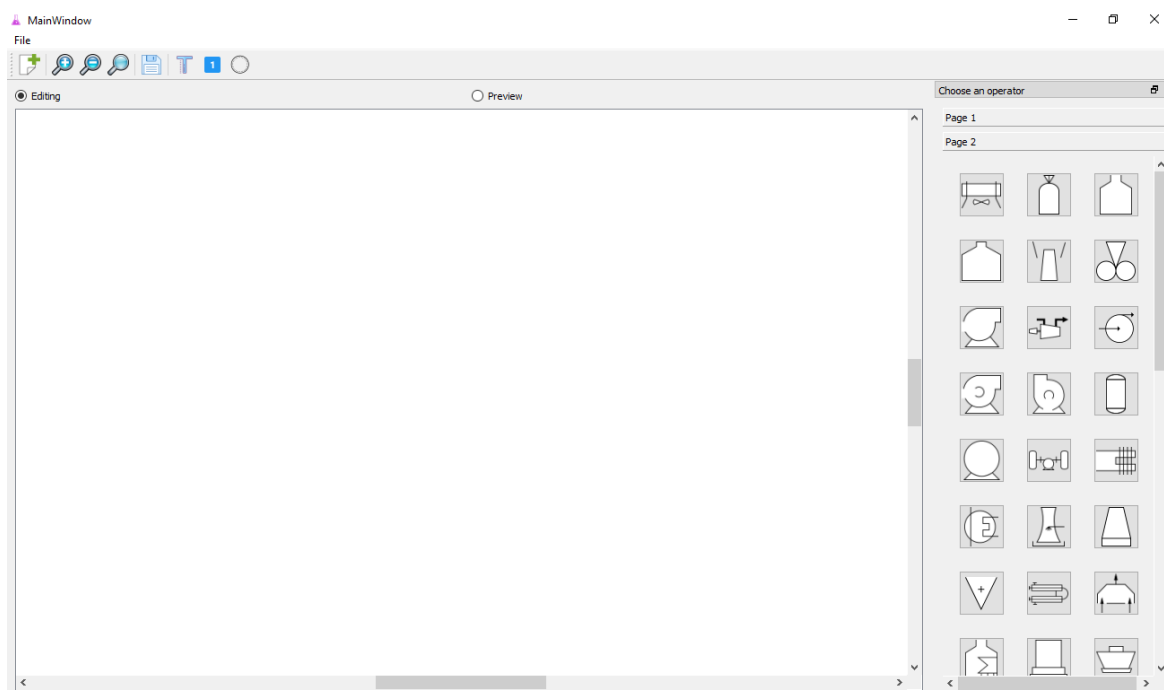


Figure 3.1.1: Basic GUI for initial PFD application

3.1.2 INTERACTIVE NODE LINE WITH ARROW HEAD

Node lines indicate the relation and connection between two chemical operators. Arrows are used to indicate the flow of chemical data from input sockets to output sockets. Likewise in the original simulator, the node lines are interactive enough to change its shape based on the positions of two operators.

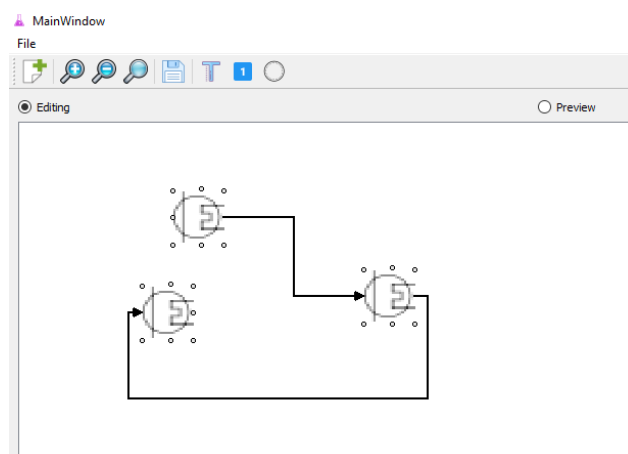


Figure 3.1.2: Two types of node lines based on the positions of operators

3.1.3 INTERACTIVE NODE SOCKETS

The main difference between chemical simulator and process flow diagramming application is the implementation of the node sockets. In chemical simulator, the node sockets are in their fixed positions, which is the correct state. As different operators has different types of sockets, for the sake of simplicity and uniformity, default eight node sockets are being implemented for all operators.

When a connection is established between two operations, a new node line will be drawn with its node sockets being hidden away. Vice versa, when a connection is deleted or removed, its node sockets will reappear again in order to show that new node lines can be formed from them.

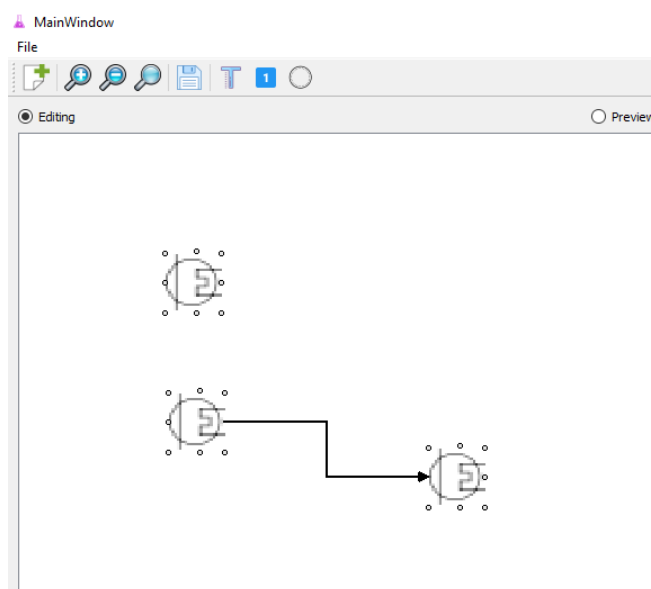


Figure 3.1.3: Default eight sockets of a operator and a sample Node Line with interactive node sockets

3.1.4 “EDITING” AND “PREVIEW” MODES

In editing mode, all the node sockets which do not form anything will appear as they are. While in preview mode, only the connection node lines and chemical operators will appear. One important feature of preview mode is that nothing new, such that creating new operators and new node lines, can be invoked in this mode. The desired mode can be switched using the radio buttons on the top of the graphic canvas.

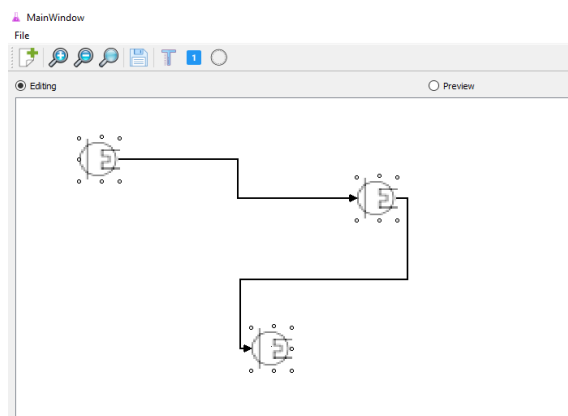


Figure 3.1.4 (a): Graphic canvas in editing mode

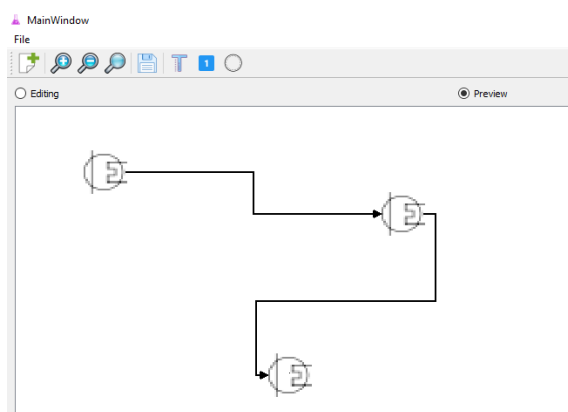


Figure 3.1.4 (b): Graphic canvas in preview mode

3.1.5 SAVING THE CANVAS AS AN IMAGE

After the completion of design process, the canvas can be saved as an image in computer file system. It will save the canvas of preview mode. This can be done by pressing “Save” icon or by clicking shortcut key “ctrl+S”. The ultimate goal is to save the canvas as it is and to be able to reopen saved graphic canvas and continue editing. This is also one of the directions of future works for next fellow interns.

3.1.6 MISCELLANEOUS

Below two figures are the sample reference PFD and the developed basic PFD. As in the reference, for each operator, it has own unique name. Moreover, there are labels and numbers spanning across the figure. The basic PFD application is implemented in such a way that it could render the image of design with the same features as the sample reference PFD. The implemented initial version could do the basic features such that naming the desired names for chemical operators, labelling the node lines with numbers, and labelling the operators with the extra texts.

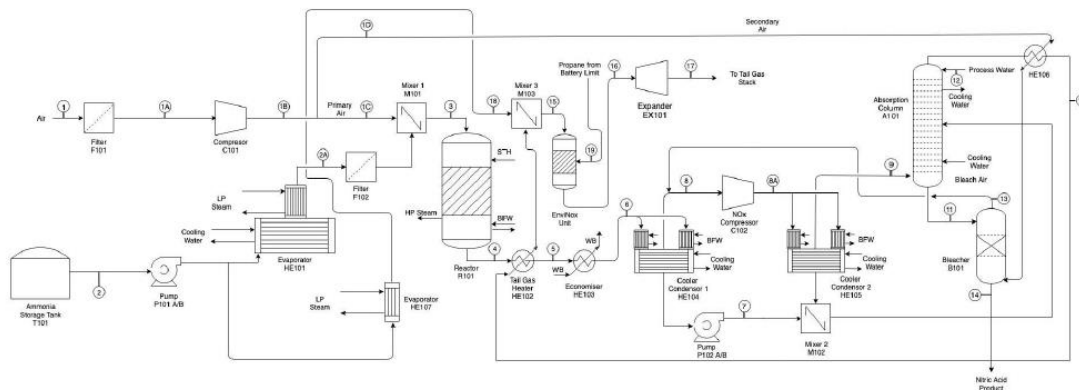


FIGURE 5.1: Process Flow Diagram for Dual Pressure Nitric Acid Process

Figure 3.1.6(a): Reference PFD sample

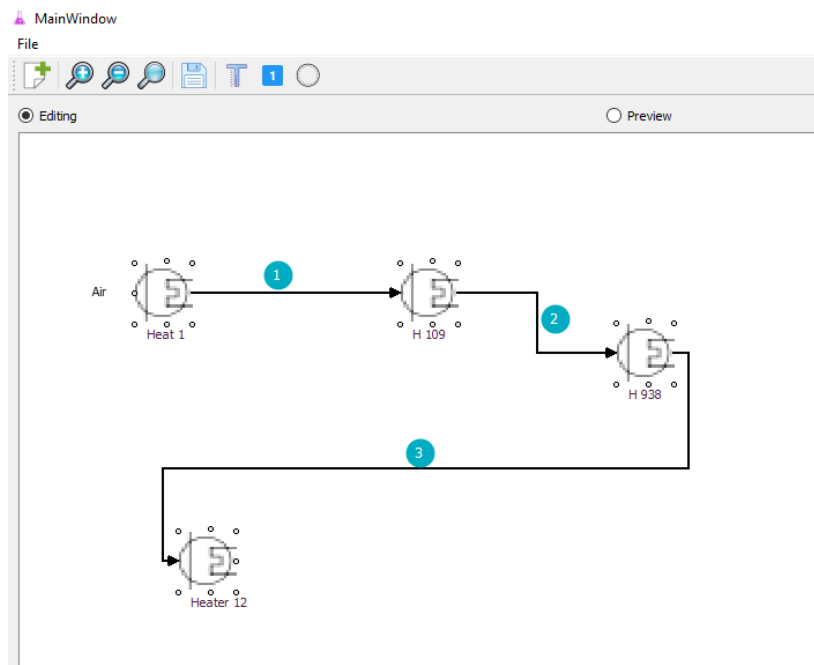


Figure 3.1.6(b): Developed basic PFD

3.2 FINAL VERSION

3.2.1 USING SVG FOR BETTER RESOLUTION AND MANIPULATION OF NODE ITEMS

In the basic version, the PNG images are used to render the chemical operators. The drawback of using png images is the poor resolution of the graphics on the canvas, either in normal state or when zooming in or zooming out. For better quality rendering and scaling, SVG are used to render the operators. Another advantage of using SVG is the ability to detect exact pixels for drawing connection lines between two operators.

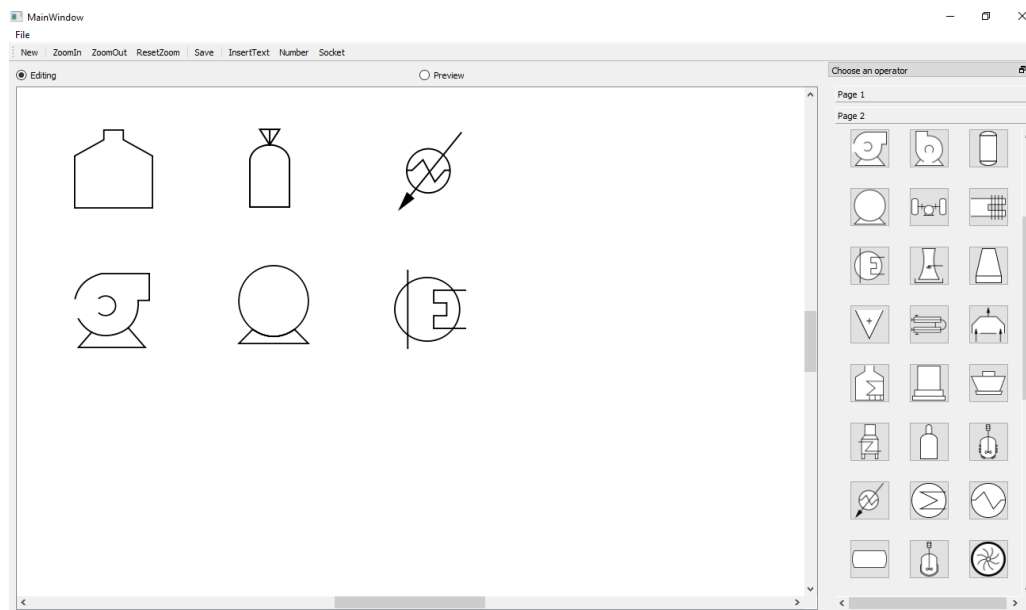


Figure 3.2.1: Sample rendered chemical operators using SVG

3.2.2 FULLY INTERACTIVE NODE LINES

In the previous chemical simulator and basic version of PFD application, the node lines are interactive enough only for basic design and simulation. For the complex design structures, they are very limited in terms of being interactive. Therefore, different ideas and implementations have been brainstormed and developed. The finalized conclusion is to have six different types of node lines which could be used as basic building blocks to draw almost all kinds of node lines. These lines are:

- (1) Left-right
- (2) Left-top
- (3) Top-right
- (4) Bottom-right
- (5) Left-bottom
- (6) Top-bottom

Each type of node line in turn has its counterpart shape based on the positions of the second operator. The details will be discussed in the coming section. This is literally the most complex, tedious and time consuming part of the whole application.

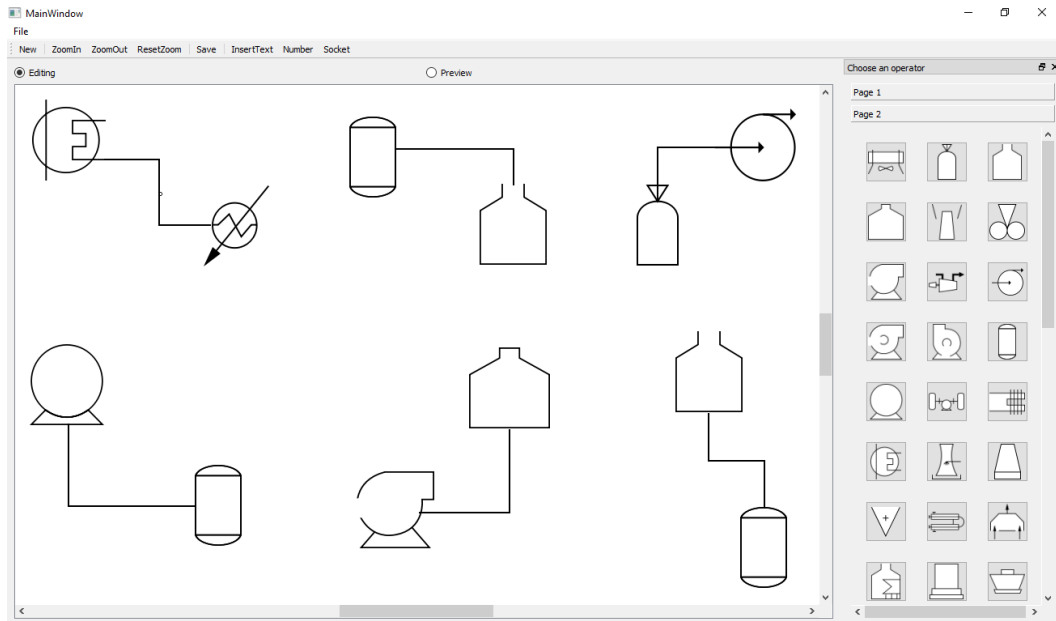


Figure 3.2.2: Six basic shapes of finalized Node Lines

3.2.3 EDITABLE NODE LINES USING NODE ANCHORS

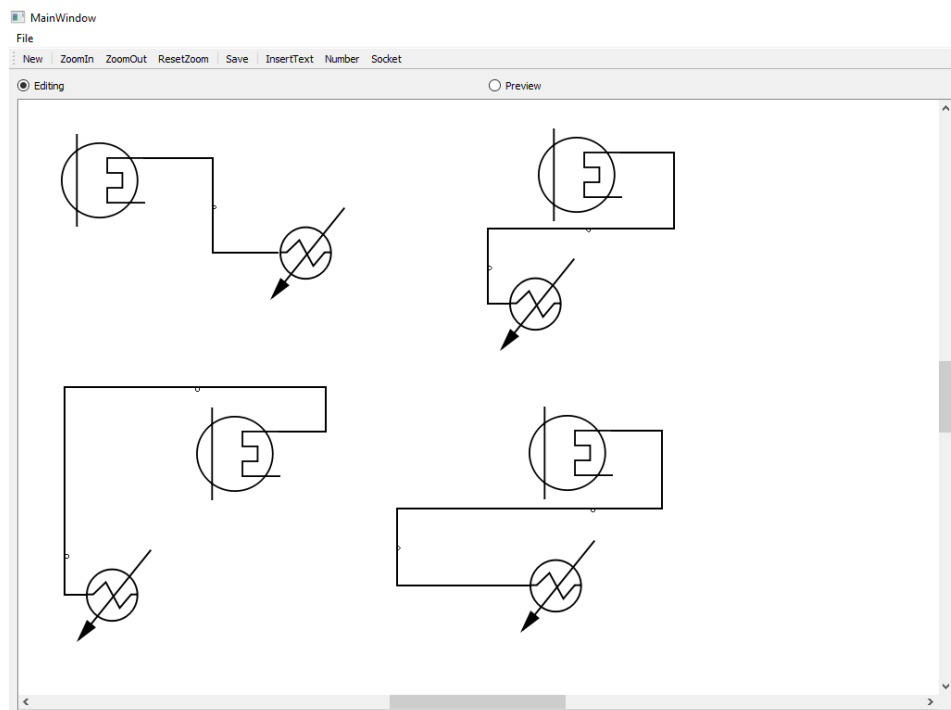


Figure 3.2.3: Demonstration of editable node lines using node anchors

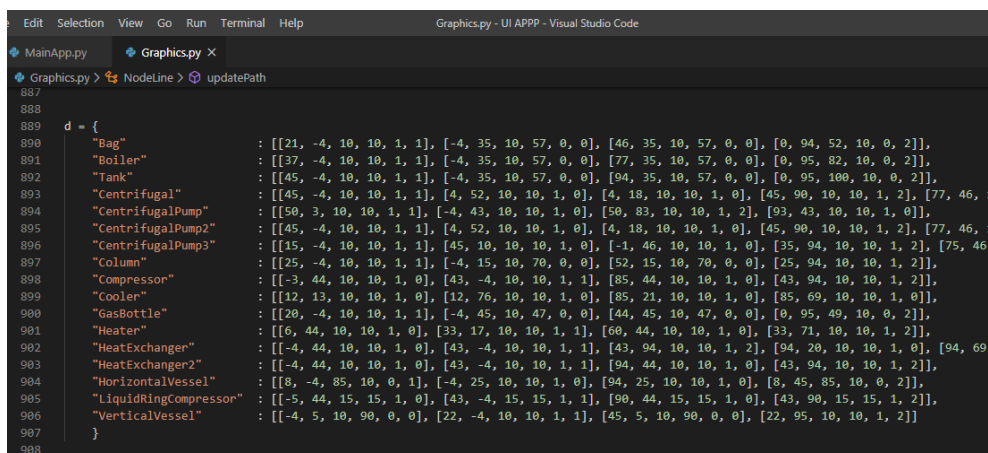
Interactive Node Lines means that these lines can be manipulated in every possible shapes. In other words, the node lines are said to be “Editable”. This feature is implemented by introducing new Node Anchor concept. Node Anchors are the anchor points which reside in the Node Lines. When a node anchor is pressed and dragged to a specific point, the respective

node lines also move along with that node anchor point. When new operators are added or removed, there will be changes in the canvas. The motivation behind editable node line is to help in the overall design structure. The above figure illustrate some of the shapes of Left-right node line in its manipulated states.

3.2.4 NEW NODE SOCKETS IMPLEMENTATION

The last-but-not-least part of PFD application is the new revamped Node Socket. Like Node Lines and Node Anchors, which have been discussed in previous sections, Node Sockets are also part of the Graphics class. They are the socket points where Node Lines begin and/or end. Unlike in chemical simulator, the sockets in PFD application is default in hidden mode. When a chemical operator is clicked, its Node Sockets are highlighted with purple rectangle boxes. When pressed and dragged, a node line will begin from that Node Socket.

The drawback of this implementation is having to do tedious and time consuming manual work for each chemical operator as different operators have different input and output sockets in their particular fixed positions.

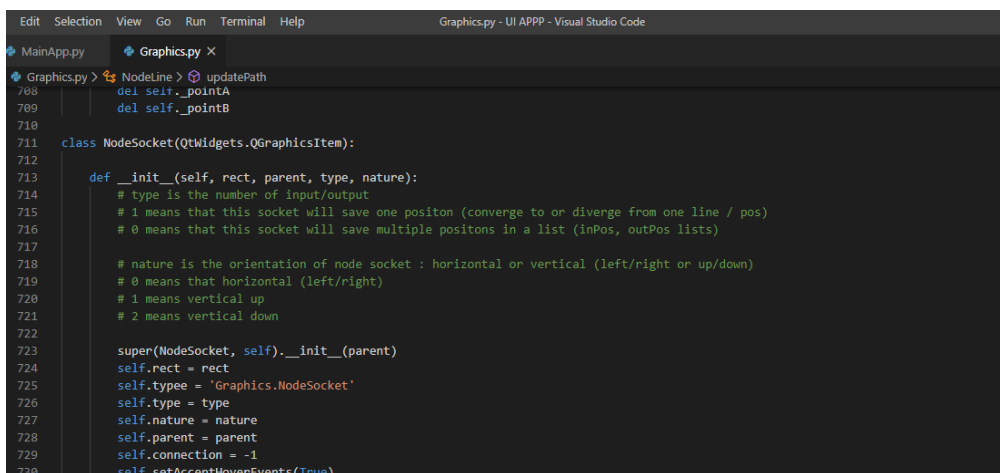


```

887
888
889 d = {
890     "Bag" : [[21, -4, 10, 10, 1, 1], [-4, 35, 10, 57, 0, 0], [46, 35, 10, 57, 0, 0], [0, 94, 52, 10, 0, 2]],
891     "Boiler" : [[37, -4, 10, 10, 1, 1], [-4, 35, 10, 57, 0, 0], [77, 35, 10, 57, 0, 0], [0, 95, 82, 10, 0, 2]],
892     "Tank" : [[45, -4, 10, 10, 1, 1], [-4, 35, 10, 57, 0, 0], [94, 35, 10, 57, 0, 0], [0, 95, 100, 10, 0, 2]],
893     "Centrifugal" : [[45, -4, 10, 10, 1, 1], [4, 52, 10, 10, 1, 0], [4, 18, 10, 10, 1, 0], [45, 90, 10, 10, 1, 2], [77, 46, 10, 10, 1, 2]],
894     "CentrifugalPump" : [[50, 3, 10, 10, 1, 1], [-4, 43, 10, 10, 1, 0], [50, 83, 10, 10, 1, 2], [93, 43, 10, 10, 1, 0]],
895     "CentrifugalPump2" : [[45, -4, 10, 10, 1, 1], [4, 52, 10, 10, 1, 0], [4, 18, 10, 10, 1, 0], [45, 90, 10, 10, 1, 2], [77, 46, 10, 10, 1, 2]],
896     "CentrifugalPump3" : [[15, -4, 10, 10, 1, 1], [45, 10, 10, 10, 1, 0], [-1, 46, 10, 10, 1, 0], [35, 94, 10, 10, 1, 2], [75, 46, 10, 10, 1, 2]],
897     "Column" : [[25, -4, 10, 10, 1, 1], [-4, 15, 10, 70, 0, 0], [52, 15, 10, 70, 0, 0], [25, 94, 10, 10, 1, 2]],
898     "Compressor" : [[-3, 44, 10, 10, 1, 0], [43, -4, 10, 10, 1, 1], [85, 44, 10, 10, 1, 0], [43, 94, 10, 10, 1, 2]],
899     "Cooler" : [[12, 13, 10, 10, 1, 0], [12, 76, 10, 10, 1, 0], [85, 21, 10, 10, 1, 0], [85, 69, 10, 10, 1, 0]],
900     "GasBottle" : [[20, -4, 10, 10, 1, 1], [-4, 45, 10, 47, 0, 0], [44, 45, 10, 47, 0, 0], [0, 95, 49, 10, 0, 2]],
901     "Heater" : [[6, 44, 10, 10, 1, 0], [33, 17, 10, 10, 1, 1], [60, 44, 10, 10, 1, 0], [33, 71, 10, 10, 1, 2]],
902     "HeatExchanger" : [[-4, 44, 10, 10, 1, 0], [43, -4, 10, 10, 1, 1], [43, 94, 10, 10, 1, 2], [94, 20, 10, 10, 1, 0], [94, 69, 10, 10, 1, 2]],
903     "HeatExchanger2" : [[-4, 44, 10, 10, 1, 0], [43, -4, 10, 10, 1, 1], [94, 44, 10, 10, 1, 0], [43, 94, 10, 10, 1, 2]],
904     "HorizontalVessel" : [[8, -4, 85, 10, 0, 1], [-4, 25, 10, 10, 1, 0], [94, 25, 10, 10, 1, 0], [8, 45, 85, 10, 0, 2]],
905     "LliquidRingCompressor" : [[-5, 44, 15, 15, 1, 0], [43, -4, 15, 15, 1, 1], [90, 44, 15, 15, 1, 0], [43, 90, 15, 15, 1, 2]],
906     "VerticalVessel" : [[-4, 5, 10, 90, 0, 0], [22, -4, 10, 10, 1, 1], [45, 5, 10, 90, 0, 0], [22, 95, 10, 10, 1, 2]]
907 }
908

```

Figure 3.2.4 (a): The drawback of node socket implementation



```

708
709
710
711 class NodeSocket(QtWidgets.QGraphicsItem):
712
713     def __init__(self, rect, parent, type, nature):
714         # type is the number of input/output
715         # 1 means that this socket will save one positon (converge to or diverge from one line / pos)
716         # 0 means that this socket will save multiple positons in a list (inPos, outPos lists)
717
718         # nature is the orientation of node socket : horizontal or vertical (left/right or up/down)
719         # 0 means that horizontal (left/right)
720         # 1 means vertical up
721         # 2 means vertical down
722
723         super(NodeSocket, self).__init__(parent)
724         self.rect = rect
725         self.typeee = 'Graphics.NodeSocket'
726         self.type = type
727         self.nature = nature
728         self.parent = parent
729         self.connection = -1
730         self.setAcceptHoverEvents(True)

```

Figure 3.2.4(b): Snippets of NodeSocket class implementation with comments on its complex constructors

CONCLUSIONS

This project started by studying its existing implementations and organizing the overall code design structure. New crucial features along with some major UI improvements have been developed and carried out. Besides, one particular part of chemical simulator has been extracted out and it became another independent software. This PFD application is still in foundation stage and needs a lot of features, improvements and enhancements to be completed by next fellow intern students.

To put it in a nutshell, in my opinion, I have completed my final capstone project successfully despite the inevitable pandemic crisis being happened at the very last two weeks of internship. Throughout the whole 4 months, I have learned many great things both for my future study and career paths.

DIRECTIONS FOR FUTURE WORK

Though I have completed my internship successfully, both OpenModelica chemical simulator and PFD application still have many rooms for new features and enhancements yet to be developed, implemented and evolved. Here are some of such examples.

TESTING

Basic testing has already been taken care while developing the applications. The remaining part is to test the various chemical operators according to the laws and phenomena of chemical engineering.

NEW FEATURES

Units play very crucial and vital role in chemical engineering. One of the features is to convert the units from one system to another system.

Another important feature is to be able to run multiple simulations at the same moment. This is supposed to be done by creating multiple graphic canvases with tabs.

The last thing is to re-render the dock widgets of the saved simulation chemical operators. The current “Open” will only re-render the graphic canvas. Since its chemical operator objects have been saved, the simulation results are also already saved. The remaining part is to print out the data again inside the respective result dock widgets.

IMPROVEMENTS

In the PFD application, there is few errors in node anchor implementation particularly the Left-right node line when the second operator is moved to left side of first operator.

There are hundreds of chemical operators with unique shapes and categories. Manual work for different node sockets positions is also one of the works yet to be done. Otherwise, another implementation has to be discussed and brainstormed.

Referring page no. 29 section 3.1.6, the labels and texts would have to be displayed properly and interactively as shown in the referenced PFD. This has not been developed in final version.

The very last requirement as per discussion is to include a table below of the canvas. This table would include all the chemical operators which have been used and some initial data or constant values.

REFERENCES / BIBLIOGRAPHY

<https://github.com/pravindalve/Chemical-Simulator-GUI>

<https://github.com/lucaszhao19/ChemicalSimulatorDiagrammingSoftwareVer0.1>

<https://github.com/lucaszhao19/Foundation-Work-on-PFD-Application/tree/master/PFD%20final>

<https://www.qt.io/qt-for-python>

<https://www.wikipedia.org/>

<https://om.fossee.in/>