ChemBox Project Documentation

Tom Schneider

Date: April 22, 2024

Center Number: 29065
Candidate Number: 7638

Ellesmere College

Contents

1 Analysis

1.1 Introduction

Technology gives us the benefits of saving time and doing work more efficiently. The use of software and technology in chemistry does not only help increase accuracy and decrease human error, but also reduces the time spent performing repetitive tasks by hand. ChemBox is a software project with the aim of creating an interactive, user-friendly and intuitive toolbox for automating and simplifying complex and repetitive tasks that come up on a daily basis for students, educators and professionals in the field of chemistry. The application features a range of different tools that should help chemists work more efficiently and also carry out their work more accurately. ChemBox is split into three distinct modules with different functionalities.

The first module is the "ChemCalculator". The aim of this part of the program is to help the user carry out calculations by filling in equations and formulae. Although substituting numbers into predefined equations is a rather trivial task, it leaves a lot of room for human error when it comes to things like converting between units or applying mathematical laws correctly. This module should help the user with the most important and at the same time most trivial tasks in chemistry.

The second module, "ChemBalancer", is for balancing chemical equations, as the name already suggests. Chemical equations come up in every lab experiment, calculation or research problem. While balancing short equations made up of very few different elements is arguably a rather easy task, it can get quite tricky when you have to work with a large number of different elements, complex ions or just very long equations. Making just a tiny mistake when balancing an important equation can cause a big set back as it can take long to find small errors like mixing up a 2 with a 3.

The third and last module is the "ChemEditor". Visualising molecular structures can play a vital role in understanding a substances chemical properties or understanding interactions with other substances. Drawing molecules out by hand is pretty straight forward. It is knowing when a bond is valid and which atoms bond together and which don't that is the tricky part. Having a tool that can help you make sure the chemical molecule you want to draw can even exist, can be a great help not only for beginner level chemists but also for more experienced chemists.

1.2 Prospective Users

Chembox will provide valuable tools to a diverse user base, spanning from students to professional chemists. The intuitive and straight forward design will allow users with varying backgrounds and degrees to use ChemBox for their own specific needs.
In the early stage, the main users of this system will be pupils and staff attending Ellesmere College, but it could be a goal to make the software available open source to anyone online.
Engaging with pupils at the college during the early stages allows for a valuable user feedback loop. This direct interaction with the user group will provide insights into the software's usability, identify potential improvements, and address any specific requirements that may arise within the college context.

1.2.1 Interview with Oliver Covill - Future User

To get an independent opinion on the topic when it comes to important features and future design of the system, I have interviewed Oliver Covill, a fellow A-Level chemistry student at Ellesmere College.

Q: What are the benefits of performing the tasks manually and by hand?

Oliver: Learning the process of the calculation and what is required for them and why we use them. Doing every step of the task by hand, be it calculating something or balancing an equation gives you a better feeling for it and teaches you the necessary foundations you need.

Q: What are the drawbacks of using the current system (by hand/without software)?

Oliver: Longer questions that require multiple equations and rearranging them can be quite tedious at times. Also, checking every calculation performed by hand can be very time consuming. Balancing long equations can sometimes be near impossible or might require multiple lines of working out, to get the right balanced equation, especially if you don't instantly notice a mistake you have made.

Q: What are the most common tasks in chemistry that could be computed and which features would be most important to you?

Oliver: Definitely a lot of the basic mole like calculations. Also having the ability to do PV=nRT calculations or having a Gibbs Free Energy calculator would be great. The most important Feature for me would be a chemical equation balancer, which is especially important for long and complex equations.

1.3 Specific Objectives

Through being an A-Level Chemistry student myself, I have learned a lot about using chemical equations and performing calculations as well as balancing chemical equations and visualising chemical substances and molecules. I was able to identify a number calculations that processes that come up on a regular basis and divide them into non-negotiable and nice-to-have objectives.

1.3.1 Chemical Equations and Calculations - ChemCalculator

The first module of the program is for performing calculations which are based on chemical formulae. Where appropriate the program should allow the user to choose from a range of different units for each calculation, so the user doesn't have to calculate the conversions like the one from cm^3 to dm^3 for example.

Required functionalities:

1. Standard moles calculation:

$$moles = \frac{mass}{molar\ mass}$$

2. Calculation to find the concentration:

$$concentration = \frac{moles}{volume}$$

3. Avogadro's number calculations. The user should be able to give a number of different inputs, including mass, moles, molecular weight and the number of atoms. After giving two independent inputs, the program should be able to calculate the rest of the values using Avogadro's number.
   The equation the calculator will be based on is:

$$number\ of\ atoms = Avogadro's\ number \times moles$$

   This should be paired with a mole calculator for the possibility use the following formula:

$$number\ of\ atoms = Avogadro's\ number \times \frac{mass}{molar\ mass}$$

4. Gibbs Free Energy calculation:

$$\Delta G = \Delta H - T\Delta S$$

   (ΔG = Gibbs Free Energy) (ΔH = enthalpy)
   (T = temperature) (ΔS = entropy)

5. Calculation for the Specific Heat Capacity and Enthalpy changes:

$$q = mc\Delta T$$

   (q = energy change) (m = mass) (c = specific heat capacity)
   (ΔT = temperature change)

6. Equilibrium Constant calculation for a reversible reaction:

$$a[A] + b[B] \rightleftharpoons c[C] + d[D]$$

$$K_C = \frac{[C]^c[D]^d}{[A]^a[B]^b}$$

   (K_C = Equilibrium Constant) (Upper case letter = Concentration) (Lower case letter = Moles in Equation)

7. Rate Equation and Rate constant calculation:

$$Rate = k[A]^m[B]^n$$

8. Ideal Gas Law calculation:

$$PV = nRT$$

   (P = Pressure, V = Volume, n = Number of moles, R = Gas constant, T = Temperature)

Non-essential objectives:

1. Acid calculations - pH and $[H^+]$

$$pH = -log[H^+]$$

$$[H^+] = 10^{-pH}$$

2. Acid dissociation constants K_a and pK_a

$$K_a = \frac{[H^+][A^-]}{[HA]}$$

$$pK_a = -logK_a$$

$$K_a = 10^{-pK_a}$$

3. Atom Economy calculation:

$$Atom\ Economy = \frac{Mr\ of\ desired\ product}{Sum\ of\ Mr\ of\ all\ reactants} \times 100$$

4. Percentage Yield calculation:

$$\%Yield = \frac{Actual\ yield}{Theoretical\ yield} \times 100$$

1.3.2 Chemical Equation Balancer - ChemBalancer

A substantial part of the project will be the ChemBalancer which will be the module that balances chemical equations. This system must be able to take complex unbalanced equations and convert them into a balanced version. It must be able to handle subscript numbers, brackets and complex ions.

1.3.3 Visualisation of Chemical Molecules - ChemEditor

The third part of the program will be the ChemEditor module, which can be used for visualising the structures of chemical molecules. This module will require a user-friendly and easy to use interface, with the main focus on the canvas. The user should have the option to choose from a range of different elements what he wants to add to the canvas. In a tool bar, the user should also be able to choose the bond order (single, double, triple) and the charge on each atom. When clicking on an atom, there should be an option to add a bond to another atom or delete the atom.
When the user constructs their molecules, ChemEditor will have to conduct real-time checks to ensure that atoms do not exceed their valence electrons and that it is chemically possible to have a molecule with the given structure. The required objectives for this module are:

1. Tool bar:
   In the tool bar on the top end of the application, there has to be a list of buttons for choosing the element, which must include the most common elements (Carbon, Hydrogen, Sulphur, Chlorine, etc.). There also has to be the option to choose the bond order (single, double or triple bond). Another essential option in form of buttons should be removing atoms and bonds as well as being able to safe the drawn structures as a document.
   A possible non-essential enhancement would be getting extra information about atoms upon highlighting as well as getting information like the molar mass and the empirical formula of a molecule after highlighting.

2. Canvas:
   The canvas is the area in which the user can draw their molecules. There are a number of essential features that must be included here.

   (a) The user must be able to draw atoms by clicking on the canvas.

(b) Upon selecting an existing atom on the canvas, depending on the chosen action type, the user should have different options:

i. When the chosen action type is "Draw", a number of greyed out atoms and bonds to those atoms should be drawn, out of which the user can choose where he wants to place his next atom.

ii. When the chosen action type is "Bond", the program should draw a bond from the selected atom to every existing atom on the canvas, with which a bond would be possible. The colour of those bonds needs to be different to the colour of the actual existing bonds, to avoid confusion.

1.4 Current and Proposed Systems

The current standard is to work out chemical equations or draw molecules on paper. This might make sense for simple equations or small molecules, but it gets less efficient and more difficult as complexity increases. Although there are some software solutions for very specific tasks, there isn't one intuitive and easy to use application that combines the different tasks in one place.

Naturally, drawing molecules with pen and paper feels best and is the preferred choice by most people. This project is not here to replace that, it should merely pose as a help for chemists when working certain things out.

2 Documented Design

2.1 Introduction to the Documented Design

In this section, I will outline the decisions, that have outlined the development of the ChemBox project and explain the programming techniques used to implement certain algorithms and structures.
The program is written in python, with the aim of using as little external frameworks and dependencies as possible, and therefore creating most of this project from scratch. For the GUI implementation, I chose to use the PyQt6 framework, which is a powerful tool for creating GUI applications in python.

2.2 Project Hierarchy

As mentioned in the analysis part of this document, the project is separated into three stand alone modules which are merged in the main class of the program, ChemBox.

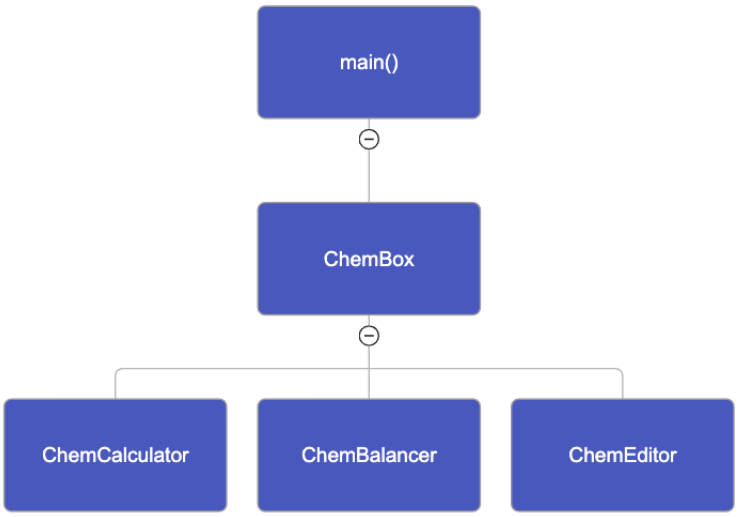


Figure 1: Program Hierarchy chart

I will explain the hierarchy and program flow of the system, beginning with the ChemCalculator module.

2.3 Structure of the GUI

The ChemBox class is the heart of the program, the point where all the different components are merged together to create one complete application. To give a better understanding of the system, I will explain how I composed the GUI and what each major component does. The ChemBox class contains only the constructor method, in which the layout of the graphical user interface is specified. The constructor defines the dimensions, the title and geometry of the window and then creates an instance of the TabBar class, which will act as the central widget of the program. Next, an instance of each of the three big components of the project, ChemCalculator, ChemBalancer and ChemEditor is created, and allocated to a separate tab of the tab bar (Listing 1: lines 18 - 25).

```
1  class ChemBox(QMainWindow):
2      def __init__(self):
3          super().__init__()
4
5          # set window properties
6          self.__left = 300
7          self.__top = 300
8          self.__width = 1280
9          self.__height = 720
10         self.__title = "ChemBox"
11         self.setWindowTitle(self.__title)
12         self.setGeometry(self.__left, self.__top, self.__width, self.
       __height)
13         self.setFixedSize(self.__width, self.__height)
14
15         self.tab_bar = TabBar()
16         self.setCentralWidget(self.tab_bar)
17
18         self.chem_calculator = ChemCalculator()
19         self.tab_bar.tab1.setLayout(self.chem_calculator.main_layout)
20
21         self.chem_balancer = ChemBalancer()
22         self.tab_bar.tab2.setLayout(self.chem_balancer.balancer_layout)
23
24         self.chem_editor = ChemEditor()
25         self.tab_bar.tab3.setLayout(self.chem_editor.editor_layout)
```

Listing 1: Code snippet of ChemBox class

The tab bar is used as the main widget of the program at all times, as it controls the navigation between modules. The goal was to achieve a design like the one illustrated in Figure 2, where 1, 2 and 3 in the small boxes at the top of the screen represent buttons in the tab bar for the three major modules of the ChemBox.

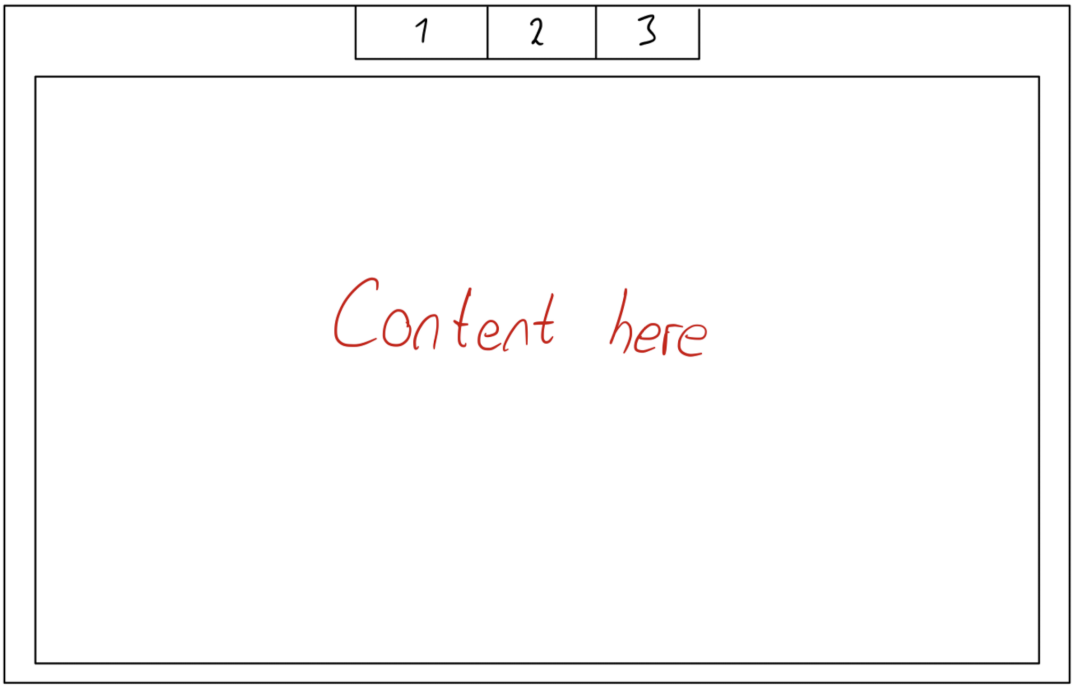


Figure 2: Mockup drawing of tab bar

The ChemCalculator module is the only part of the project with a special implementation of the user interface, which needs explanation.

When developing the ChemCalculator module, the first issue I encountered was how I would create a layout that would work for multiple separate sub-calculators. I had three different possible systems for displaying the module. The first option was creating a sidebar, where the user can choose the exact calculator they were looking for, which is then displayed on the screen. The second option was very similar, but with an additional tab bar at the top or bottom of the screen. This solution is not very aesthetically pleasing, and could cause confusion with the actual tab bar that allows the user to switch between the three main modules. The third option was putting every calculator on the same page, each in it's own area, clearly separated from the others, and make the window scrollable. I decided against this option,

as this design could have gotten very messy. Therefore, I decided to create a sidebar for the ChemCalculator.
As there is no built-in sidebar widget in PyQt6, I have designed my own way of creating one. I did so by using the QTabWidget, which I also used for the tab bar, but not display it. I was able to benefit from the already existing widget, for the switching between calculators, and use buttons placed on the left side of the screen as a replacement for an actual sidebar. The buttons connect to a method, which changes the the current index of the tab widget to the according number of the calculator, which updates the page that is displayed. This is illustrated in the following code snippet (Listing 2), which only includes the example on a single button, for better readability of this document.

```
class ChemCalculator(QWidget):
    def __init__(self):
        super(QWidget, self).__init__()

        self.side_bar_layout = QVBoxLayout()

        self.gibbs_calc = GibbsFreeEnergyCalculator()

        # Create buttons
        self.gibbs_free_energy_tab_button = QPushButton("Gibbs Free Energy
    Calculator")

        self.gibbs_free_energy_tab_button.clicked.connect(self.
    gibbs_free_energy_action)

        # Create tabs
        self.gibbs_free_energy_tab = QWidget()

        # Initialise gibbs free energy calculator
        self.gibbs_free_energy_tab.setLayout(self.gibbs_calc.layout)

        # Add buttons to sidebar layout
        self.side_bar_layout.addWidget(self.gibbs_free_energy_tab_button)

        self.side_bar_widget = QWidget()
        self.side_bar_widget.setLayout(self.side_bar_layout)

        self.page_widget = QTabWidget()

        self.page_widget.addTab(self.gibbs_free_energy_tab, "")

        self.page_widget.setCurrentIndex(0)
        self.page_widget.setStyleSheet('''QTabBar::tab{
        width: 0;
        height: 0;
        margin: 0;
        padding: 0;
        border: none;
        }''')

        self.main_layout = QHBoxLayout()
```

```
40          self.main_layout.addWidget(self.side_bar_widget)
41          self.main_layout.addWidget(self.page_widget)
42
43          self.main_widget = QWidget()
44          self.main_widget.setLayout(self.main_layout)
45
46      # Define actions for each button
47      def gibbs_free_energy_action(self):
48          self.page_widget.setCurrentIndex(5)
```

Listing 2: Example of ChemCalculator implementation

2.4 Algorithm Design for ChemCalculator

Essential for most individual calculators in the ChemCalculator module will be an algorithm to determine which of the input options the user left blank. In general, the user interface will always consist of a number of inputs, implemented as QLineEdit's, where the blank ones are the ones that our program will calculate.
The algorithm will have to take a list of inputs as a parameter, and use iteration to determine the blank one.

Algorithm 1 Algorithm to find empty input

$input\_list \leftarrow []$
$count \leftarrow 0$
$empty\_input \leftarrow NONE$
FOR $i \leftarrow 0$ *to* $Len(input\_list)$ **DO**
    IF $input\_list[i]$ *is empty* **THEN**
        $count \leftarrow count + 1$
        $empty\_input \leftarrow input\_list[i]$
    END IF
END FOR
IF $count = 1$ **THEN**
    $RETURN\ count$
END IF

2.5 Algorithm Design for ChemBalancer

Figure 3 includes my initial attempt at visualising how an equation balancer could work. My first version of this module worked on a very similar system, although I struggled with finding a method of consistently balancing the equations after splitting them into their smallest possible components.

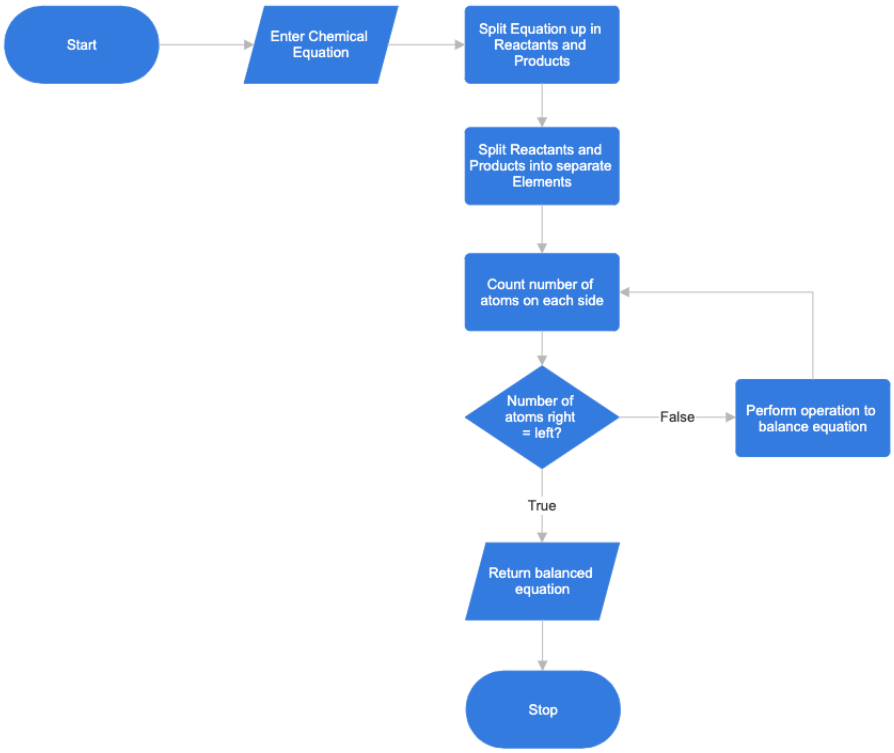


Figure 3: Initial flowchart for balancer

The following code blocks contains the code of my initial, failed, attempt at the balancer.

```
def splitEquation(self):
        self.equationSplit = self.equationInput.text().split(" = ")
        self.reactants = self.equationSplit[0]
        try:
            self.products = self.equationSplit[1]
        except IndexError:
            print("Wrong user input")
        self.reactantComponents = self.reactants.split(" + ")
        self.productComponents = self.products.split(" + ")
```

Listing 3: Method for splitting equation into components

The splitEquation method uses string manipulation to extract the individual molecules of the equation and store them in separate lists for reactants and products.
After finding the individual molecules in the equation, the parseComponent method is called to get the elements used in the molecules, and the amount of them. This is done through if statements, as there is only a limited amount of different possibilities. The longest element symbol consists of three letters, where the first letter of every atom is always capitalised, and the following ones are not. The subscript number (number of atoms or element in a given molecule) can be found easily, as numbers within molecules always belong to the preceding element (H2O - the 2 belongs to the hydrogen). Individual elements are found by checking for capital letters. An element starts with a capital letter and ends at the next capital letter, which is where the next element starts. The same techniques were used to find brackets.

```
def parseComponent(self, component, countsDict, totalDict):
        # Check for coefficient
        try:
            if component[0] in self.integers:
                try:
                    if component[0 + 1] in self.integers:
                        try:
                            if component[0 + 2] in self.integers:
                                coefficient = int(component[0: 0 + 3])
                            else:
                                coefficient = int(component[0: 0 + 2])
                        except IndexError:
                            coefficient = int(component[0: 0 + 2])
                    else:
                        coefficient = int(component[0])
                except IndexError:
                    coefficient = int(component[0])
            else:
                coefficient = 1
            for i in range(len(component)):
                try:
                    openBracket = component.find("(")
                    closedBracket = component.find(")")
                except IndexError:
                    continue
                try:
                    if component[i].isupper() and component[i - 1] != "(":
                        try:
                            if component[i + 1].islower():
                                try:
                                    if component[i + 2].islower():
                                        element = component[i:(i + 3)]
                                        if openBracket < i < closedBracket:
                                            subCoefficient = self.
    getSubCoefficient(component)
                                        else:
                                            subCoefficient = 1

```

```
                                    # Check for subscript
                                    try:
                                        if component[i + 3] in self.
integers:
                                            try:
                                                if component[i + 4] in
self.integers:
                                                    try:
                                                        if component[i +
 5] in self.integers:
                                                            subscript =
int(component[(i + 3): (i + 6)])
                                                        else:
                                                            subscript =
int(component[(i + 3): (i + 5)])
                                                    except IndexError:
                                                        subscript = int(
component[(i + 3): (i + 5)])
                                                else:
                                                    subscript = int(
component[i + 3])
                                            except IndexError:
                                                subscript = int(
component[i + 3])
                                        else:
                                            subscript = 1
                                    except IndexError:
                                        subscript = 1
                                else:
                                    element = component[i:(i + 2)]

                                    if openBracket < i < closedBracket:
                                        subCoefficient = self.
getSubCoefficient(component)
                                    else:
                                        subCoefficient = 1

                                    # Check for subscript
                                    try:
                                        if component[i + 2] in self.
integers:
                                            try:
                                                if component[i + 3] in
self.integers:
                                                    try:
                                                        if component[i +
 4] in self.integers:
                                                            subscript =
int(component[(i + 2): (i + 5)])
                                                    except IndexError:
                                                        subscript = int(
component[(i + 2): (i + 4)])
                                            except IndexError:
                                                subscript = int(
component[i + 2])
                                        else:
                                            subscript = 1
                                    except IndexError:
                                        subscript = 1
```

```
                                except IndexError:
                                    element = component[i:(i + 2)]
                                    if openBracket < i < closedBracket:
                                        subCoefficient = self.
getSubCoefficient(component)
                                    else:
                                        subCoefficient = 1
                            else:
                                element = component[i]
                                if openBracket < i < closedBracket:
                                    subCoefficient = self.getSubCoefficient(
component)
                                else:
                                    subCoefficient = 1
                                try:
                                    # Check for subscript
                                    if component[i + 1] in self.integers:
                                        try:
                                            if component[i + 2] in self.
integers:
                                                try:
                                                    if component[i + 3] in
self.integers:
                                                        subscript = int(
component[i + 1: i + 4])
                                                except IndexError:
                                                    subscript = int(
component[i + 1: i + 3])
                                            else:
                                                subscript = int(component[i
+ 1])
                                        except IndexError:
                                            subscript = int(component[i +
1])
                                    else:
                                        subscript = 1
                                except IndexError:
                                    subscript = 1
                        except IndexError:
                            element = component[i]

                            if openBracket < i < closedBracket:
                                subCoefficient = self.getSubCoefficient(
component)
                            else:
                                subCoefficient = 1

                            try:
                                # Check for subscript
                                if component[i + 1] in self.integers:
                                    try:
                                        if component[i + 2] in self.integers
:
                                            try:
                                                if component[i + 3] in self.
integers:
                                                    subscript = int(
component[i + 1: i + 4])
                                            except IndexError:
```

```
                                                subscript = int(component[i
    + 1: i + 3])
                                            else:
                                                subscript = int(component[i +
    1])
                                        except IndexError:
                                            subscript = int(component[i + 1])
                                    else:
                                        subscript = 1
                                except IndexError:
                                    subscript = 1
                            try:
                                if element in countsDict:
                                    countsDict[element] = int(countsDict[element
    ]) + subscript * coefficient * subCoefficient
                                else:
                                    countsDict[element] = subscript *
    coefficient * subCoefficient

                                if element in totalDict:
                                    totalDict[element] = int(totalDict[element])
     + subscript * coefficient * subCoefficient
                                else:
                                    totalDict[element] = subscript * coefficient
     * subCoefficient
                            except UnboundLocalError:
                                continue

                        # Check for brackets / complex ion in equation
                        elif component[i] == "(":
                            try:
                                if component[i + 2] == ")":
                                    element = component[i + 1]
                                    try:
                                        if component[i + 3] in self.integers:
                                            try:
                                                if component[i + 4] in self.
    integers:
                                                    try:
                                                        if component[i + 5] in
    self.integers:
                                                            subscript = int(
    component[(i + 3): (i + 6)])
                                                    else:
                                                        subscript = int(
    component[(i + 3): (i + 5)])
                                                except IndexError:
                                                    subscript = int(
    component[(i + 3): (i + 5)])
                                            else:
                                                subscript = int(component[(i
     + 3): (i + 4)])
                                        except IndexError:
                                            subscript = int(component[i +
    3])
                                    else:
                                        subscript = 1
                                except IndexError:
                                    subscript = 1
```

```
                            elif component[i + 1].isupper():
                                if component[i + 2].islower():
                                    try:
                                        if component[i + 3].islower():
                                            element = component[(i + 1): (i + 4)]

                                            # Check for subscript within brackets
                                            try:
                                                if component[i + 4] in self.integers:
                                                    try:
                                                        if component[i + 5] in self.integers:
                                                            try:
                                                                if component[i + 6] in self.integers:
                                                                    subscript = int(component[(i + 4): (i + 7)])
                                                                else:
                                                                    subscript = int(component[(i + 4): (i + 6)])
                                                            except IndexError:
                                                                subscript = int(component[(i + 4): (i + 6)])
                                                        else:
                                                            subscript = int(component[i + 4])
                                                    except IndexError:
                                                        subscript = int(component[i + 4])
                                                else:
                                                    subscript = 1
                                            except IndexError:
                                                subscript = 1

                                            # Find subscript coefficient of complex ion
                                            subCoefficient = self.getSubCoefficient(component)
                                        else:
                                            element = component[(i + 1): (i + 3)]

                                            # Check for subscript within brackets
                                            try:
                                                if component[i + 3] in self.integers:
                                                    try:
                                                        if component[i + 4] in self.integers:
                                                            try:
                                                                if component[i + 5] in self.integers:
                                                                    subscript = int(component[(i + 3): (i + 6)])
```

```
                                                                    else:
                    subscript = int(component[(i + 3): (i + 5)])
                                                                except
                    IndexError:
                                                                    subscript =
                    int(component[(i + 3): (i + 5)])
                                                            except IndexError:
                                                                subscript = int(
                    component[i + 3])
                                                        else:
                                                            subscript = 1
                                                    except IndexError:
                                                        subscript = 1

                                                    # Find subscript coefficient of
                    complex ion
                                                    subCoefficient = self.
                    getSubCoefficient(component)
                                            except IndexError:
                                                element = component[(i + 1): (i + 3)
                    ]

                                                # Check for subscript within
                    brackets
                                                try:
                                                    if component[i + 3] in self.
                    integers:
                                                        try:
                                                            if component[i + 4] in
                    self.integers:
                                                                try:
                                                                    if component[i +
                     5] in self.integers:
                                                                        subscript =
                    int(component[(i + 3): (i + 6)])
                                                                    else:
                                                                        subscript =
                    int(component[(i + 3): (i + 5)])
                                                                except IndexError:
                                                                    subscript = int(
                    component[(i + 3): (i + 5)])
                                                            except IndexError:
                                                                subscript = int(
                    component[i + 3])
                                                        else:
                                                            subscript = 1
                                                    except IndexError:
                                                        subscript = 1

                                                    # Find subscript coefficient of
                    complex ion
                                                    subCoefficient = self.
                    getSubCoefficient(component)
                                            else:
                                                element = component[i + 1]

                                                # Check for subscript within brackets
                                                try:
```

```
257                                         if component[i + 2] in self.integers
    :
258                                             try:
259                                                 if component[i + 3] in self.
    integers:
260                                                     try:
261                                                         if component[i + 4]
    in self.integers:
262                                                             subscript = int(
    component[(i + 2): (i + 5)])
263                                                         else:
264                                                             subscript = int(
    component[(i + 2): (i + 4)])
265                                                     except IndexError:
266                                                         subscript = int(
    component[(i + 2): (i + 4)])
267                                             except IndexError:
268                                                 subscript = int(component[i
    + 2])
269                                         else:
270                                             subscript = 1
271                                     except IndexError:
272                                         subscript = 1
273
274                                     # Find subscript coefficient of complex
    ion
275                                     subCoefficient = self.getSubCoefficient(
    component)
276                             except IndexError:
277                                 print("wrong user input ")
278                             try:
279                                 if element in countsDict:
280                                     countsDict[element] += subscript *
    subCoefficient * coefficient
281                                 else:
282                                     countsDict[element] = subscript *
    subCoefficient * coefficient
283                                 if element in totalDict:
284                                     totalDict[element] += subscript *
    subCoefficient * coefficient
285                                 else:
286                                     totalDict[element] = subscript *
    subCoefficient * coefficient
287                             except UnboundLocalError:
288                                 continue
289                         else:
290                             continue
291                     except IndexError:
292                         continue
293             except IndexError:
294                 None
```

Listing 4: parseComponent method for finding elements

Although this code is overly complicated, it worked most of the times for basic or intermediate level equations.
The main problem this version of the balancer ran into, was the actual bal-

ancing of the equations. The version uses a loop to attempt balancing, where copies of the left and right hand side components are created together with dictionaries to track the total count of elements. Each component then gets a randomly generated coefficient between 1 and 10, and the loop continues until the count on the left side equals the count on the right side. Once a balanced equation is found, the coefficients are normalised to their smallest integer values using the greatest common divisor (GCD) The balanced equation is then put back together and returned to the user.

```
def balance(self):
        if self.balanced:
            equation = str()
            for dictionary in self.left:
                compound = str()
                for element in dictionary:
                    compound += element
                    if dictionary[element] > 1:
                        compound += str(dictionary[element])
                equation += compound
                equation += " + "
            equation = equation[:len(equation) - 3] + " = "
            for dictionary in self.right:
                compound = str()
                for element in dictionary:
                    compound += element
                    if dictionary[element] > 1:
                        compound += str(dictionary[element])
                    else:
                        pass
                equation += compound
                equation += " + "
            equation = equation[:len(equation) - 2]
        else:
            while not self.balanced:
                tempLeft = list()
                tempRight = list()
                totalLeft = dict()
                totalRight = dict()

                for item in self.left:
                    newDict = dict()
                    for key in item:
                        newDict[key] = item[key]
                    tempLeft.append(newDict)

                for item in self.right:
                    newDict = dict()
                    for key in item:
                        newDict[key] = item[key]
                    tempRight.append(newDict)

                leftCoefficients = [randint(1, 10) for _ in range(len(
    tempLeft))]
                rightCoefficients = [randint(1, 10) for _ in range(len(
```

```
    tempRight))]
45
46                  for index in range(0, len(leftCoefficients)):
47                      for key in tempLeft[index]:
48                          tempLeft[index][key] *= leftCoefficients[index]
49                          if key not in totalLeft:
50                              totalLeft[key] = tempLeft[index][key]
51                          else:
52                              totalLeft[key] += tempLeft[index][key]
53                  for index in range(0, len(rightCoefficients)):
54                      for key in tempRight[index]:
55                          tempRight[index][key] *= rightCoefficients[index]
56                          if key not in totalRight:
57                              totalRight[key] = tempRight[index][key]
58                          else:
59                              totalRight[key] += tempRight[index][key]
60                  self.balanced = True
61                  for key in totalLeft:
62                      if totalLeft[key] != totalRight[key]:
63                          self.balanced = False
64                      else:
65                          continue
66
67              bigTup = tuple(leftCoefficients + rightCoefficients)
68              leftCoefficients = list(map(lambda x: int(x / reduce(gcd, bigTup
    )), leftCoefficients))
69              rightCoefficients = list(map(lambda x: int(x / reduce(gcd,
    bigTup)), rightCoefficients))
70
71              balancedEquation = str()
72              for index in range(0, len(self.left)):
73                  if leftCoefficients[index] != 1:
74                      compound = str(leftCoefficients[index])
75                  else:
76                      compound = str()
77                  for key in self.left[index]:
78                      compound += key
79                      if self.left[index][key] != 1:
80                          compound += str(self.left[index][key])
81                  balancedEquation += compound
82                  balancedEquation += " + "
83              balancedEquation = balancedEquation[:len(balancedEquation) - 3]
    + " = "
84              for index in range(0, len(self.right)):
85                  if rightCoefficients[index] != 1:
86                      compound = str(rightCoefficients[index])
87                  else:
88                      compound = str()
89                  for key in self.right[index]:
90                      compound += key
91                      if self.right[index][key] != 1:
92                          compound += str(self.right[index][key])
93                  balancedEquation += compound
94                  balancedEquation += " + "
95              balancedEquation = balancedEquation[:len(balancedEquation) - 2]
96              return self.balancedLabel.setText(f"{balancedEquation}")
```

Listing 5: balance method

This version had a few obvious limitations, it relied on randomly generated coefficients, which is not a very reliable system of finding the correct coefficients. Although the random generation usually provides coefficients, it may not always produce the most optimal or smallest possible coefficients. The normalisation step using the greatest common divisor ensures that the coefficients are integer multiples of each other, but it does not guarantee the smallest possible coefficients. If a balanced equation cannot be found with the randomly generated coefficients, there is also a possibility of infinite looping.

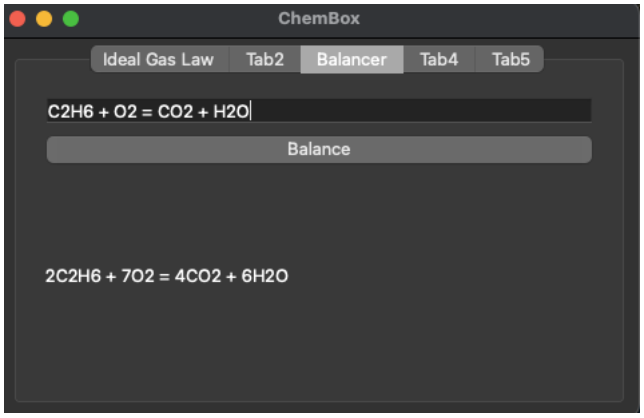


Figure 4: Initial balancer test

The test in Figure 4 shows, that the system works (at least for simple equations). Just the simple addition of two carbon atoms and four hydrogen atoms to get the equation $C2H6 + O2 = CO2 + H2O$ is already too much for the system, as this will end in an infinite loop. This loop can be prevented by increasing the range of possibilities for the randomly generated integers (for example from between 1 and 10 to between 1 and 100) but this means there is always a limit to what is possible for the balancer. Figure 5 shows the state of the program after trying to balance the equation with the original range of one to 10. The program in the figure is in the state of an infinite loop.

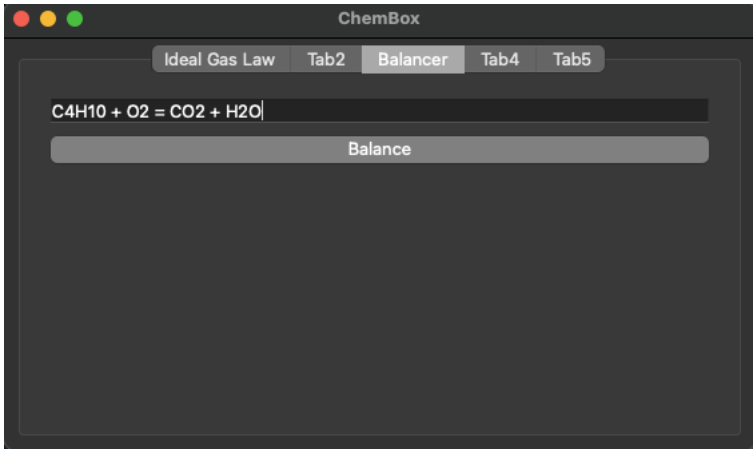


Figure 5: Initial balancer test No. 2

Whilst researching for better ways to implement a chemical equation balancer in python, I came across regular expressions, which around that same time, we also covered in our A-Level Computer Science lessons. Knowing about regular expressions helped me significantly improve the effectiveness of my balancer, when it comes to extracting individual elements and their amount from the equation. In my search for the best possible way to do so, I wrote a number of algorithms, which extend each other perfectly, to finish the job together.
This algorithm (Algorithm 2) takes the full equation as a parameter from the user input and removes any whites paces from it. It then splits it up into separate reactants and products, and lastly, it extracts the individual reagents within the reactants and products.

Algorithm 2 Algorithm to split equation

SUBROUTINE *split_equation*()
    $stripped\_equation \leftarrow USERINPUT$
    $stripped\_equation \leftarrow stripped\_equation.STRIP(\text{""})$
    $equation\_split \leftarrow stripped\_equation.SPLIT(\text{" = "})$
    $reactants \leftarrow equation\_split[0].SPLIT(\text{" + "})$
    $products \leftarrow equation\_split[1].SPLIT(\text{" + "})$
END SUBROUTINE

This second algorithm(Algorithm 3), which I designed for the task mentioned above, uses regular expressions, to identify separate reagents within a chemical compound by identifying brackets. The goal is to split up compounds like $CuNO3)2$ into ["Cu", "$(NO3)2$"], where Cu and $(NO3)2$ are

two separate entries in the list. The method then iterates over each reagent, and checks if it starts with an opening bracket. If it does, this indicates, that there is a chemical compound enclosed in the brackets. It extracts the inner compound and the subscript ($(OH)2$ indicates that the (OH) group exists twice), and then passes these two variables as parameters to the "find_elements" method, along with the "index" and "side" parameters.

Algorithm 3 Algorithm to find reagents

```
SUBROUTINE find_reagents(compound, index, side)
    reagents ← SPLIT compound INTO reagents USING REGEX PATTERN
    FOR reagent IN reagents DO
        IF reagent BEGINS WITH "(" THEN
            inner_compound ← SUBSTRING(1, LEN(reagent))
            bracket_subscript ← SPLIT reagent BY ")" AND GET SECOND PART
            IF bracket_subscript EXISTS THEN
                bracket_subscript ← INT(bracket_subscript)
            ELSE
                bracket_subscript ← 1
            END IF
            find_elements(inner_compound, index, bracket_subscript, side)
        ELSE
            bracket_subscript ← 1
            find_elements(reagent, index, bracket_subscript, side)
        END IF
    END FOR
END SUBROUTINE
```

The next algorithm (Algorithm 4) uses a regular expression to obtain the elements and associated subscript values. Each extracted element is then stored together with the correlated subscript value as a tuple, inside a list (For example: $Cr2O7 \rightarrow$ [("Cr", "2"), ("O", "7")]). The algorithm then iterates through each tuple (elements, subscript) in the element_counts list. With every iteration, a different subroutine named add_to_matrix is called, passing the current element, index, the product of the bracket_subscript and subscript, and the side argument.

Algorithm 4 Algorithm to find elements

```
SUBROUTINE find_reagents(reagent, index, bracket_subscript, side)
    element_counts ← SPLIT reagent INTO element_counts USING REGEX PATTERN
    FOR element, subscript IN element_counts DO
        IF subscript DOES NOT EXIST THEN
            subscript ← 1
        ELSE
            subscript ← INT(subscript)
        END IF
        add_to_matrix(element, index, bracket_subscript * subscript, side)
    END FOR
END SUBROUTINE
```

While I was conducting my research, I found a neat solution to balancing equations, matrix operations. As we had only covered the rudimentary principles of matrix operations, I chose to use the article [2] I had found on the internet to get a better understanding of the topic. After getting a grasp of the concept, and after trying out the code in the article, I had found a different way to tackle my problem. I decided not to reinvent the wheel, but to take the already existing algorithm and improve it, so it could be used for practically any tractable equation there is.
To address the problem, I decided to record a number of issues and possible improvements for the existing algorithm.

1. The first issue I had with the algorithm was, that it asked for separate inputs of reactants and products. Luckily, I had already solved this problem with the algorithms illustrated earlier.

2. Another issue the original algorithm had was the lack of structure and modularity. By organising the code in a class based structure, I ensured good readability for the code, as well as making it easier to maintain.

3. The original version had poor input validation and error handling. One common error I got with the early implementation of the algorithm was what I call a nullspace error. The nullspace is a linear subspace that contains a set of vectors that transform to the zeroth vector under a given linear transformation: multiplication with the matrix A, represented as $Ax = 0$.[1] In the context of chemical equations, the nullspace contains the coefficients, that balance the equation.
   Entering the unbalanced equation $K4[Fe(SCN)6]+K2Cr2O7+H2SO4 = Fe2(SO4)3+Cr2(SO4)3+CO2+H2O+K2SO4+KNO3$ resulted in a "nullspace error" (Figure 6).

```
Traceback (most recent call last):
  File "/Users/tom/Downloads/ChemBox-db54135db0ee918912ceb046cf7ef95920d8f1bf/main.py", line 345, in runBalancer
    num = self.elementMatrix.nullspace()[0]
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^
IndexError: list index out of range
```

Figure 6: Nullspace error

I managed to neutralise this error by having a more accurate and efficient method of separating out molecules, elements, etc. as shown in Algorithm 3 and Algorithm 4, before passing them on to the method that adds the items to the matrix.
I also made sure to use pythons built in exception handling to catch any expected and unexpected errors or exceptions.

Using the current working system, inputting the chemical equation mentioned before results in a perfectly balanced equation (Figure 7).

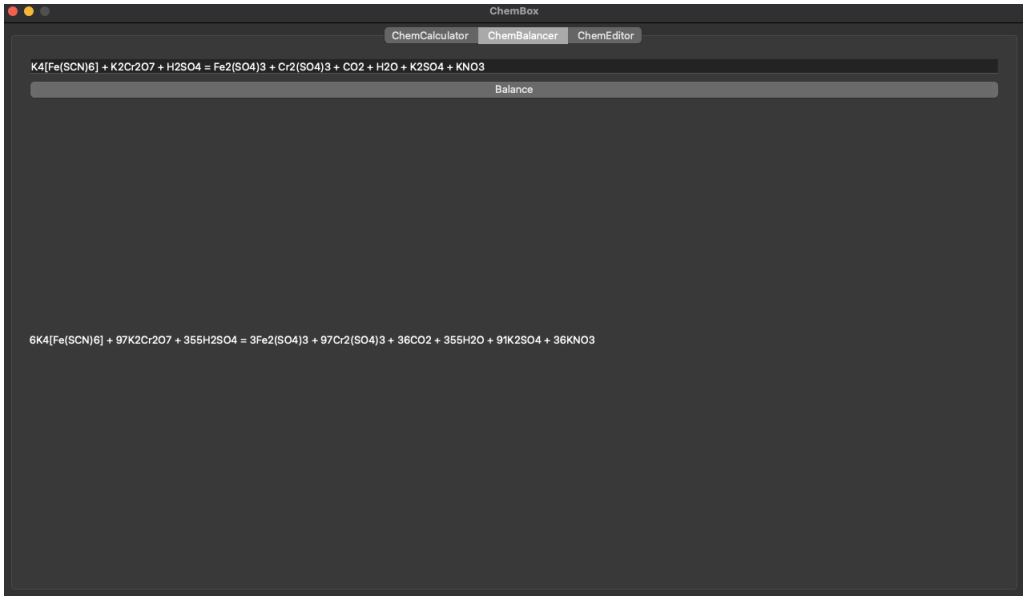


Figure 7: Balanced equation

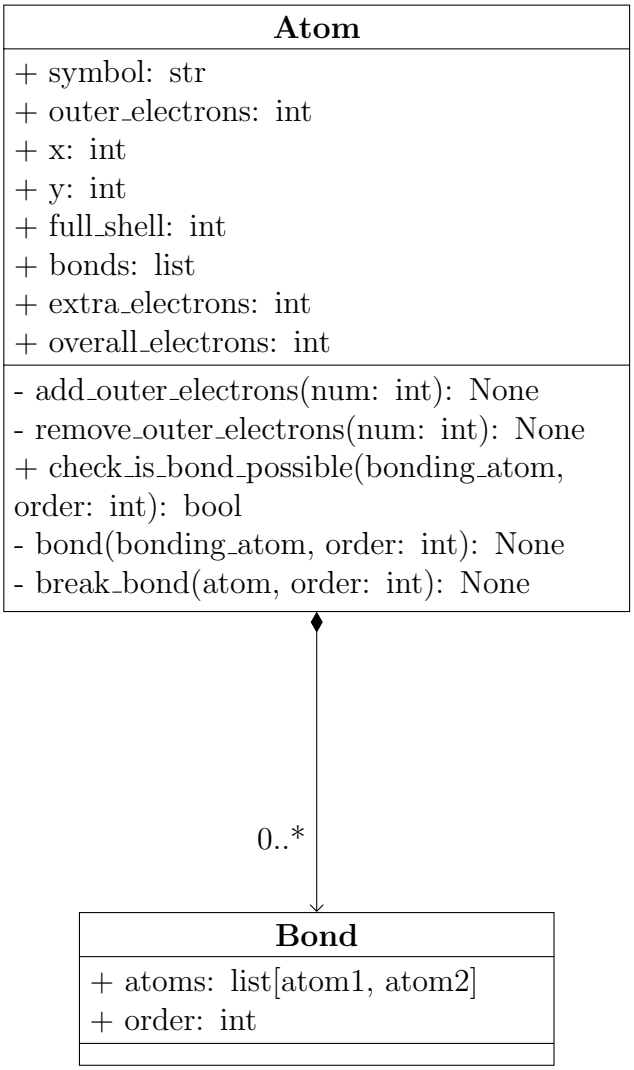
2.6 Algorithm Design for ChemEditor

My initial attempt at creating the ChemEditor was a mixture of following the plan made in the analysis stage and trying out different things to see what works best.

To start my work on this module, I decided to try to identify a number of structures I will have to implement. The ChemEditor can be broken down into a graphical user interface part and a logic part. The GUI section has to consist of two main classes, the "ChemEditor" class, which takes care of displaying all the buttons, the canvas and other parts of the GUI, and the "Canvas" class, which can be further broken down into two main structures, the "paintEvent", which takes care of all the drawing, and the "mousePressEvent", which defines what is supposed to happen after a users interaction with the interface. The segment taking care of the logic of this module must contain an "Atom" class and a "Bond" class which define certain actions and properties.

2.6.1 ChemEditor Logic

I first started my work on the logic of the module, where my initial action was to define the Atom class and the Bond class and their associated methods and variables. The Atom class is the blueprint used for every individual atom on the canvas. The Bond class is only used in the bond() method of the Atom class, where a new bond between two atoms is initiated. Each atom can have as many bonds as it has free spaces in its outer shell, while each instance of a bond has to always exist of exactly two atoms.



Algorithm 5 Algorithm to check whether a bond is possible

```
SUBROUTINE check_is_bond_possible(bonding_atom, order)
    IF (overall_electrons + order) > full_shell THEN
        RETURN False
    ELSE IF (bonding_atom.overall_electrons) > bonding_atom.full_shell THEN
        RETURN False
    ELSE
        RETURN True
    END IF
END SUBROUTINE
```

Algorithm 5 investigates whether or not the suggested bond is chemically possible by using selection statements. The first "if statement" tests if the

overall amount of (outer shell) electrons after addition of the extra electron, or electrons dependent on the chosen bond order, it would gain after bonding is greater than the allowed full shell capacity of the atom. If this applies, the subroutine returns the boolean value "False". The subsequent "else if statement" test the exact same thing for the other bonding atom, and returns False if it applies as well. If the program flow manages to get through both of these statements, the method returns "True", as the formation of a bond between the two atoms is possible.

Algorithm 6 Algorithm for bonding two atoms

```
SUBROUTINE bond(bonding_atom, order)
    IF check_is_bond_possible = False THEN
        RETURN
    END IF
    new_bond ← Bond(bonding_atom, order)
    bonds.append(new_bond)
    bonding_atom.bonds.append(new_bond)
    extra_electrons ← extra_electrons + order
    bonding_atom.extra_electrons ← bonding_atom.extra_electrons + order
    overall_electrons ← outer_electrons + extra_electrons
    bonding_atom.overall_electrons ← bonding_atom.outer_electrons +
bonding_atom.extra_electrons
END SUBROUTINE
```

The bonding algorithm (Algorithm 6) of the atom class is called when the user wants to bond two atoms together. The opening action of the method uses the subroutine shown in algorithm 5 to find out if the bond is possible, and returns back to the main program is this is not the case. If the check returned a boolean value of "True", a new instance of the class Bond with the correct bond order is created. The freshly formed bond instance is then appended to the list of bonds of each atom, and the number of extra electrons and outer electrons of both atoms is adjusted using the bond order, to correctly store the current amount of electrons each atom has in its outer shell.

2.6.2 ChemEditor GUI

One issue I encountered early on when working on the canvas was, that the bonds between atoms would start at the center of the atoms, and it could be complicated to correctly adjust the starting end endpoints of each bond (Figure 8).

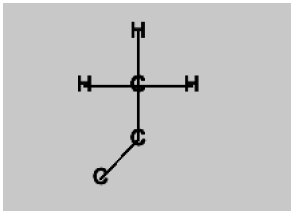


Figure 8: Overlap between atoms and bonds

To overcome this problem, I decided to create a method for drawing an invisible circle in the same colour as the background behind the symbol of each atom to hide the bonds (draw_atom_circle() in code). This method is called after the bonds are drawn, in order to make the overlap with the atoms invisible in the eyes of the user. After drawing the bonds and drawing the circle, the actual atom is drawn on top of it, without any visible overlap between atoms and bonds. Evidently, there is a clear hierarchy as to how the drawing of atoms, bonds and everything else is drawn. After applying the required hierarchy, this is what the bonding of atoms looks like in the current version (Figure 9).

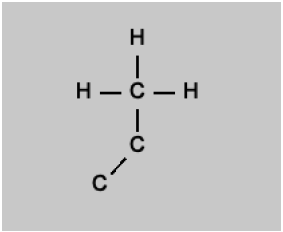


Figure 9: No overlap between atoms and bonds

One feature, which I thought would improve the user experience a lot was what I have called "drawing potential bonds". The goal of this is to draw a range of potential atoms and bonds in a circle around the selected atom in a different colour, so that the user only has to click on one of those potential atoms, and it gets drawn as an actual atom with a real bond to the selected atom. The "potential atoms" and the "potential bonds" are related to the element chosen by the user in the "periodic table" and to the

buttons suggesting the bond order.To get all the potential positions for the atoms, I created the "calc_potential_positions()" subroutine, which takes the atom instance selected by the user as a parameter and then accesses the atoms x and y coordinates and creates an empty list, which will later hold the positions of potential atoms. Using a for loop, the method calculates the x and y vector components using the given angle and magnitude (distance). This subroutine calculates the positions of potential atoms every 45 degrees, and therefore return a maximum of 8 different tuples holding the positions.

Algorithm 7 Algorithm for finding potential atom positions

```
SUBROUTINE calc_potential_positions(atom)
    x ← atom.x_coords
    y ← atom.y_coords
    distance ← 40
    coordinate_list ← NONE
    FOR angle_degrees ← 0 TO 360 STEP 45 DO
        angle_radians ← math.radians(angle_degrees)
        new_x ← x + distance * math.cos(angle_radians)
        new_y ← y + distance * math.sin(angle_radians)
        APPEND (new_x, new_y) TO coordinate_list
    END FOR
    RETURN coordinate_list
END SUBROUTINE
```

All the drawing happens in the "paintEvent" method inside the "Canvas" class. To get a better understanding of the following pseudocode for this method, I have put together a table containing all the methods used for it (Table 1).

Method Name	Parameters	Description
calc_potential_positions	Instance of atom class	Calculates and returns a list of positions for potential atoms in a circle around the atom.
check_atom_overlap	pos_x, pos_y	Iterates through list of atoms and checks for overlap, returns a boolean value.
draw_single_bond	atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter, pen, actual_bond: bool	Draws a bond line from one atom to another.
draw_double_bond	atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter, pen, actual_bond: bool	Draws two bond line next to each other from one atom to another.
draw_triple_bond	atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter, pen, actual_bond: bool	Draws three bond line next to each other from one atom to another.
__diagonal_bonds	atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter, offset: int, diag_offset: int	Handles the drawing of the diagonal bonds in double and triple bonds to avoid overlap of lines.
draw_atom_circle	atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter, pen	Draws a circle in the same colour as the background colour to prevent bonds from visually overlapping with atom.
draw_atom	atom1_x: int, atom1_y: int, symbol: str, painter, pen, potential: bool	Draws the atom symbol on the screen.

Table 1: Method Descriptions

The pseudocode of my "paintEvent" method shows the clear structure and hierarchy when it comes to drawing out all the atoms and bonds on the canvas. The method can be broken down in to two main parts, the first one just iterates through a list of atoms and draws the atom and the associated bonds. The second part is only called if an atom is currently selected, and it essentially draws out the possible bonds or potential atoms around it.

Algorithm 8 paintEvent Algorithm, first part

```
SUBROUTINE paintEvent(event)
    FOR atom IN atoms_list DO
        draw_atom(x, y, symbol, painter, pen, False)
        FOR bond IN atom.bonds DO
            IF bond.order = 2 THEN
                draw_double_bond(x1, y1, x2, y2, symbol, painter, True)
            ELSE IF bond.order = 3 THEN
                draw_triple_bond(x1, y1, x2, y2, symbol, painter, True)
            ELSE
                draw_single_bond(x1, y1, x2, y2, symbol, painter, True)
            END IF
            draw_atom_circle(x2, y2, x1, y1, painter, pen)
            draw_atom(x2, y2, symbol2, painter, pen, False)
            draw_atom(x1, y1, symbol1, painter, pen, False)
        END FOR
    END FOR
    SECOND PART HERE
END SUBROUTINE
```

For visualisation purposes, I have decided to split the pseudo code for the "paintEvent" method into the two parts mentioned before.
As explained in an earlier part, it is essential to follow a certain hierarchy structure for drawing out the canvas. Algorithm 8 (paintEvent Algorithm, first part) illustrated this procedure perfectly. The very first thing to happen every time the "paintEvent" is called, is that using an iterative for-loop structure, every atom in the atoms list is being drawn on the canvas. In the bigger picture, this is so that all the atoms that are not bonded and don't need anything extra are drawn. Next, all the bonds of the specific atom are displayed, and the "atom circle" is drawn. And lastly both atoms on either side of the bond are drawn and the first part of the "paintEvent" method is completed. As noted at the start of this sub-section 2.6, it is important that for bonded atoms, the bond is drawn first , then the "atom circle" and lastly the actual atoms.

Algorithm 9 paintEvent Algorithm, second part

```
IF selected = True THEN
    x1 ← selected_atom.x_coords
    y1 ← selected_atom.y_coords
    IF action_type! = "bond" THEN
        IF selected_atom IS NOT NONE THEN
            potential_positions ← calc_potential_positions(selected_atom)
            FOR pos IN potential_positions DO
                IF check_atom_overlap(pos[0], pos[1]) = False THEN
                    IF bond_order = 2 THEN
                        draw_double_bond(x1, y1, pos[0], pos[1], painter, pen, False)
                    ELSE IF bond_order = 3 THEN
                        draw_triple_bond(x1, y1, pos[0], pos[1], painter, pen, False)
                    ELSE
                        draw_single_bond(x1, y1, pos[0], pos[1], painter, pen, False)
                    END IF
                    draw_atom_circle(pos[0], pos[1], x1, y1, painter, pen)
                    draw_atom(pos[0], post[1], symbol, painter, pen, True)
                    draw_atom(x1, y1, symbol, painter, pen, False)
                END IF
            END FOR
        END IF
    END IF
END IF
```

This second part of the "paintEvent" (Algorithm 9), uses the "calc_potential_positions()" method discussed earlier and all in all just puts the whole functionality of drawing potential atoms and bonds around the selected atom together.

The "mousePressEvent" subroutine handles the users actions on the canvas. The pseudocode for this algorithm is divided into two separate blocks for visualisation purposes (Algorithm 10 and Algorithm 11). The method initially uses "if statements" to find out which action type the user has currently selected (draw, bond, remove). If the selected action type is "remove", the atom at the click position and if applicable all of the atoms bonds are removed.

Algorithm 10 mousePressEvent Algorithm, first part

```
SUBROUTINE mousePressEvent(event)
    IF user clicked on canvas THEN
        click_pos ← user click position
        IF action_type = "remove" THEN
            remove_bond(click_pos.x, click_pos.y)
            remove_atom(click_pos.x, click_pos.y)
        ELSE IF action_type = "bond" THEN
            FOR atominatoms_list DO
                atom_x ← atom.x_coords
                atom_y ← atom.y_coords
                IF atom position = click position THEN
                    selected ← True
                    selected_atom ← atom
                    temp_bond_list.append(atom)
                    IF LEN(temp_bond_list) = 2 THEN
                        IF temp_bond_list[0] = temp_bond_list[1] THEN
                            OUTPUT "Trying to bond to itself"
                            temp_bond_list.CLEAR()
                            RETURN
                        END IF
                        temp_bond_list[0].bond(temp_bond_list[1], bond_order)
                        temp_bond_list.CLEAR()
                        selected_atom ← NONE
                    END IF
                    RETURN
                END IF
                selected ← False
            END FOR
        ELSE
            SECOND PART HERE
        END IF
    END IF
END SUBROUTINE
```

If the user has selected "bond", the program iterates over the list of all atoms on the canvas, and compares each atoms position with the users click position. If the click position matches the coordinates of an atom, this atom is added to a temporary list, and gets bonded to the other atom in that list, as soon as the length of this list is equal to two.

Lastly, in the second part of the subroutine (Algorithm 11), which can be imagined to be placed in algorithm 10 after the last else statement where it says " SECOND PART HERE", if the user has selected the draw action, another selection is used to check whether an atom is currently selected. If this is the case, it means that the circle with potential atoms is currently displayed. This section now evaluates whether the user has clicked on one of the potential atoms and therefore allows this atom and bond to be permanently added to the canvas, and then creates those new atoms and bonds. Otherwise if the user has not clicked on an atom, this means that they have clicked on a blank area on the canvas, a new atom is created at the click position and added to the list of atoms on the canvas.

Algorithm 11 mousePressEvent Algorithm, second part

```
IF selected = True THEN
    IF selected_atom IS NOT NONE THEN
        potential_positions ← calc_potential_positions(selected_atom)
        IF check_atom_overlap(pos) = False THEN
            FOR pos IN potential_positions DO
                IF pos = clickposition THEN
                    new_atom ← Atom(element, pos)
                    IF new_atom.check_is_bond_possible(selected_atom, bond_order) =
True THEN
                        atoms_list.append(new_atom)
                        new_atom.bond(selected_atom, bond_order)
                    END IF
                END IF
            END FOR
            selected ← False
        END IF
    END IF
END IF
IF check_clicked_on_atom(pos) = True THEN
    RETURN
END IF
new_atom ← Atom(element, pos)
atoms_list.append(new_atom)
```

3 Testing

As usual with software projects, a lot of undocumented exploratory testing is happening during the production phase. A good example for this would be the tests shown in Figure 4, 5 and 7, which luckily got captured in the moment.

3.0.1 Tests for ChemCalculator

The following tables contain an overview of basic input and output tests on the ChemCalculator module to ensure that the expected outcomes align with the actual outcomes during system usage. As this module has many very similar features, I have decided to summarise those tests into fewer fields, to make this section less repetitive. The tests will of course still contain any outliers of the testing process.

Test No.	**Description**	**TEX (Typical, Erroneous, Extreme**	**Expected Outcome**	**Actual Outcome**
1	Calculate value for blank input	T: Fill 2 out of 3 input options with common values	Blank field calculated correctly.	As expected.
		E: Fill in all 3 input options with common values	Must leave exactly one input line empty for it to be calculated!	As expected.
		X: Use invalid characters	Only numerical values in the form of integers or decimals allowed!	As expected.
2	Test unit drop down	T: Change units	Value scales accordingly.	As expected.

Table 2: Universal Calculator tests

Table **??** shows the only outlier out of all the tests conducted on the ChemCalculator module. The Avogadro's calculator can with certain combi-

Test No.	**Description**	**TEX (Typical, Erroneous, Extreme**	**Expected Outcome**	**Actual Outcome**
3	Calculate value for blank input	T: Leave one input blank	Blank field calculated correctly.	As expected.
		E: Fill in all input options with common values	Must leave one input line empty for it to be calculated!	As expected.
		X: Use invalid characters	Only numerical values in the form of integers or decimals allowed!	As expected.
4	Calculate value for multiple blank inputs	T: Leave allowed combination of inputs blank	Blank fields calculated correctly.	Must leave one input line empty for it to be calculated!
		E: Leave two critical inputs blank (mass and molecular weight)	Must leave one input line empty for it to be calculated!	As expected.

Table 3: Avogadros constant calculator tests

nations of inputs also work with two blank inputs. The outlier can be found in the typical (T) row of test no. 4, where allowed combinations of empty inputs are tested. Allowed combinations in this case would be leaving the "mass" and the "number of atoms" lines blank, as the "number of atoms" input can be calculated from the combination of the moles and Avogadro's constant and calculating the mass only requires the molecular weight and the moles.
To correct this, I have used "if-statements" to find out which line have been left blank and which ones have not. After applying this in a correct sequence and implementing our error dialogue, the calculator now works in the expected way and produces the desired output.

3.0.2 Tests for ChemBalancer

To test the robustness and efficiency of the balancer, I chose to pick an independent list of chemical equations to find out if my program can handle every different type of equation listed. The following chemical equations represent a sample of the tests for robustness and efficiency applied on the balancer. This list was published as a part of an article on "Journal of High School Science". Note that the equations are not written with proper subscript formatting, as it is shown in the format, which the balancer takes as an input and produces as an output, with the only difference, that arrows are used here compared to equals signs in the program.

1. Unbalanced equation: C4H10 + O2 → CO2 + H2O
   Balanced equation: 2C4H10 + 13O2 → 8CO2 + 10H2O

2. Unbalanced equation: (NH4)2Cr2O7 → N2 + Cr2O3 + H2O
   Balanced equation: (NH4)2Cr2O7 → N2 + Cr2O3 + 4H2O

3. Unbalanced equation: C57H110O6 + O2 → CO2 + H2O
   Balanced equation: 2C57H110O6 + 163O2 → 114CO2 + 110H2O

4. Unbalanced equation: KNO3 + C12H22O11 → N2 + CO2 + H2O + K2CO3
   Balanced equation: 48KNO3 + 5C12H22O11 → 24N2 + 36CO2 + 55H2O + 24K2CO3

5. Unbalanced equation: Cu2S + HNO3 → Cu(NO3)2 + CuSO4 + NO2 + H2O
   Balanced equation: 1Cu2S + 12HNO3 → 1Cu(NO3)2 + 1CuSO4 + 10NO2 + 6H2O

6. Unbalanced equation: K4[Fe(SCN)6] + K2Cr2O7 + H2SO4 → Fe2(SO4)3 + Cr2(SO4)3 + CO2 + H2O + K2SO4 + KNO3
   Balanced equation: 6K4[Fe(SCN)6] + 97K2Cr2O7 + 355H2SO4 → 3Fe2(SO4)3 + 97Cr2(SO4)3 + 36CO2 + 355H2O + 91K2SO4 + 36KNO3

7. Unbalanced equation: Na2S2O4 + NaOH → Na2SO3 + Na2S + H2O
   Balanced equation: 3Na2S2O4 + 6NaOH → 5Na2SO3 + 1Na2S + 3H2O

8. Unbalanced equation: C6H8O7 + NaHCO3 → Na3C6H6O7 + CO2 + H2O
   Balanced equation: 19C6H8O7 + 54NaHCO3 → 18Na3C6H6O7 + 60CO2 + 49H2O

9. Unbalanced equation: P4O10 + H2O → H3PO4
   Balanced equation: 1P4O10 + 6H2O → 4H3PO4

While conducting these tests, an unexpected issue appeared, where if an invalid equation is given, with an arrow instead of an equals sign for example (I.e.: C4H10 + O2 → CO2 + H2O instead of C4H10 + O2 = CO2 + H2O),the program has first opened a dialogue stating the issue to the user and then after the arrow has been changed to an equals sign, the application crashed upon hitting the "balance" button. Figure 10 shows the crash report for this bug.

Figure 10: Balancer Crash Report

I have resolved this issue using pythons exception handling. As the ChemBalancer module has many methods calling other methods, I have had to nest every relevant method in a try - except statement and raise this exception to the next higher method level. A smaller example version this solution is shown in Listing 6.

```python
def lowest_level_function():
  try:
      # Do something...
    except Exception:
      raise # Raise the exception to be caught by the next higher level
    function

def middle_level_function():
    try:
        lowest_level_function()
    except Exception:
        raise  # Re-raise the exception to be caught by the highest level
    function

def highest_level_function():
    try:
        middle_level_function()
    except Exception:
        return  # Stop further execution of highest level function
```

Listing 6: Example of exception structure

After eliminating the mentioned issues, I have tested the program with the entries of the list of equations again. The balancer has successfully produced the desired balanced equation every time. An example of a chemical equation balanced by the ChemBalancer is given in Figure 11. For this example I chose equation number 6 from the list, as this shows a very complex chemical equation that would be incredibly hard for a human to balance without any tools, which is exactly what the purpose of this project is.

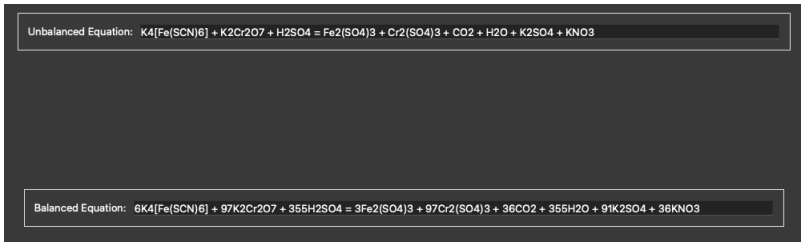


Figure 11: Working test of balancer

3.0.3 Tests for ChemEditor

As with the other modules, most features were tested by exploratory test during development. I have tried to summarise the tests for the ChemEditor in the following table.

Test No.	**Description**	**TEX (Typical, Erroneous, Extreme**	**Expected Outcome**	**Actual Outcome**
1	Draw atoms	T: Click on a blank are on the canvas	Atom gets drawn at click position.	As expected.
2	Bond two atoms together	T: Select bond action type and click on two different atoms	Bond between atoms appears.	As expected
		E: Click on at least one atom with full or near full outer shell with high bond order so that overall electrons would be greater than the full shell	Invalid user Action!	As expected.
		X: Select bond action type and click on the same atom twice	Invalid user Action!	As expected.
3	Remove Atom	T: Select remove action type and click on atom	Atom and connected bonds disappear	As expected

Table 4: ChemEditor tests

4 Evaluation

The following tables compare the finished system with the set objectives at the beginning of this document.

Colour keys for comparison:

Objective not met	Objective partially met	Objective met	Objective exceeded

4.0.1 Objective Comparison for ChemCalculator

Original Objective	**Completed System**
Standard moles calculator	Objective met.
Concentration calculator	Objective met.
Avogadro's number calculator with included moles calculator (mass/-molar mass)	Objective met.
Gibbs Free Energy calculator	Objective met.
Specific heat capacity calculator	Objective met.
Equilibrium constant calculator	Objective met.
Rate calculator calculator	Objective met.
Ideal Gas Law calculator	Objective met.

Table 5: ChemCalculator objective comparison

4.0.2 Objective Comparison for ChemBalancer

1. Handling Brackets: The balancer appropriately deals with chemical equations containing brackets. It uses regular expressions to recognise and process substances enclosed within brackets and correctly identifies separate reagents and elements.

2. Support for subscript numbers: Subscript numbers are identified correctly in the unbalanced equation and are handled in a way that the equation can be balanced properly.

3. Support for complex ions: Although the balancer can easily deal with equations containing square brackets ([ ]), it can not yet handle ionic charges. So the balancer can deal with chemical equations in the form of complex ions as long as charges are not included.

4. Handling complex unbalanced equations: The system successfully handles long and complex user inputs, and correctly identifies individual reagents, elements, brackets and subscripts.

5. Conversion to balanced equations: The ChemBalancer accurately converts unbalanced chemical equations into balanced versions, providing the user has followed the allowed "syntax" and rules for the module. It applies the necessary coefficients to each compound to ensure that the number of atoms of each element is equal on each side of the equation.

In summary, the ChemBalancer module successfully meets most of the set objectives, with the exception of complex ions. The system demonstrates its capability to balance most equations effectively, and it serves its purpose well.

4.0.3 Objective Comparison for ChemEditor

Original Objective	**Completed System**
Provide a user-friendly and easy to use interface.	The finished system has a very easy to use interface including pop-up dialogues for invalid user actions.
Have a range of commonly used elements for the user to choose from.	The ChemEditor provides the user with a near full periodic table including all the period 1, 2, 3, 4 and 5 elements.
Support ions and let the user choose different charges on atoms (ions)	This module does not yet support ions.
Allow for single, double and triple bonds	The user can choose between different bond orders.
Conduct real-time checks to ensure atoms do not exceed their valence electrons.	The program stores the elements valence electrons and checks if a new bond would lead to a higher number of electrons in the outer shell than allowed every time the user tries to bond two atoms together.

Table 6: ChemEditor objective comparison

Original Objective	Completed System
Tool bar containing buttons for the choice of element and bond order as well as options for choosing different action types like bond, draw or remove.	The tool contains the option for opening the periodic table, choose the preferred bond order, action types like bond, draw and remove, as well as giving the option to clear the canvas or save it as an image.
Draw atoms at click position on canvas	The system draws an atom of the chosen element at the user's click position.
Display a number of potential atoms and bonds when the user clicks on existing atom in draw mode.	System draws up to 8 potential atoms and bonds in a circle around the selected atom.
Provide the ability to bond atoms together	The system bonds the two latest selected atoms together

Table 7: ChemEditor objective comparison

4.0.4 Potential for Future Development

Thanks to the nature of the project, there are always new tools or features one could add. If the project were to be revisited in the future, the first task might be to include the non-essential objectives in the system. Also, two great features to add would be the accepting ionic charges in the balancer and being able to draw ions (with charges) on the canvas of the ChemEditor. As the field of chemistry is very broad, and even physicists could find use in a all-in-one toolbox, there is an unlimited range of possible future additions when it comes to new calculations or new drawing methods like drawing 3D molecules. As this program was created as an A-Level project, it doesn't even come close to the features required by an undergraduate or postgraduate student, so this could be picked up again at a later stage to add required features by those kinds of users.
A project like this would be a great open source software, where students and scientists from all over the world and every kind of level of education could add features they might require or know to be required by other people.

References

[1] Asqua Amir. *What is a Nullspace?* 2024. URL: https://www.educative.io/answers/what-is-a-null-space.

[2] Diego Chacon and Shireesh Apte. "A python code algorithm for balancing chemical equations as a system of simultaneous linear equations using matrix algebra." In: *Journal of High School Science* (2023).

Appendices

A Program Code

A.1 main.py File

```
import sys

from PyQt6.QtWidgets import QApplication, QMainWindow

from chem_editor_gui import ChemEditor

from chem_calculator import ChemCalculator

from chem_balancer import ChemBalancer
from gui_comps import TabBar


class ChemBox(QMainWindow):
    def __init__(self):
        super().__init__()

        # set window properties
        self.__left = 300
        self.__top = 300
        self.__width = 1280
        self.__height = 720
        self.__title = "ChemBox"
        self.setWindowTitle(self.__title)
        self.setGeometry(self.__left, self.__top, self.__width, self.
    __height)
        self.setFixedSize(self.__width, self.__height)

        self.tab_bar = TabBar()
        self.setCentralWidget(self.tab_bar)

        self.chem_calculator = ChemCalculator()

        self.tab_bar.tab1.setLayout(self.chem_calculator.main_layout)

        self.chem_balancer = ChemBalancer()
        self.tab_bar.tab2.setLayout(self.chem_balancer.balancer_layout)

        self.chem_editor = ChemEditor()
        self.tab_bar.tab3.setLayout(self.chem_editor.editor_layout)


def main():
    app = QApplication(sys.argv)

    # Load CSS file
    app.setStyleSheet(open('style.css').read())

    main_win = ChemBox()
    main_win.show()
```

```
    sys.exit(app.exec())


if __name__ == "__main__":
    main()
```

Listing 7: main.py File Program Code

A.2 GUI Components File

```
from PyQt6.QtWidgets import QTabWidget, QWidget, QHBoxLayout, QGridLayout,
    QFrame, QLabel, QLineEdit, QComboBox


class TabBar(QWidget):
    def __init__(self):
        super(QWidget, self).__init__()

        self.layout = QHBoxLayout(self)

        self.tabs = QTabWidget()
        self.tab1 = QWidget()
        self.tab2 = QWidget()
        self.tab3 = QWidget()
        self.tabs.addTab(self.tab1, "ChemCalculator")
        self.tabs.addTab(self.tab2, "ChemBalancer")
        self.tabs.addTab(self.tab3, "ChemEditor")

        self.layout.addWidget(self.tabs)
        self.setLayout(self.layout)


class RateBox(QWidget):
    """
    This class contains the gui components for molecules for the
    RateCalculator.

    ########################################
    # Order of reaction: _____________     #
    # Concentration: ____________________  #
    ########################################
    """

    def __init__(self, symbol):
        super(QWidget, self).__init__()

        self.layout = QGridLayout()

        self.box = QFrame(self)
        self.box.setFrameStyle(0x0001)
        self.box.setLineWidth(3)

        self.box_layout = QGridLayout()

        self.order_label = QLabel("Order of reaction: ")
        self.conc_label = QLabel(f"Concentration {symbol}: ")
```

```
45
46          self.order_input = QComboBox()
47
48          self.order_input.addItem("First")
49          self.order_input.addItem("Second")
50          self.order_input.setCurrentIndex(0)
51
52          self.conc_input = QLineEdit()
53
54          self.layout.addWidget(self.order_label, 0, 0)
55          self.layout.addWidget(self.order_input, 0, 1)
56          self.layout.addWidget(self.conc_label, 1, 0)
57          self.layout.addWidget(self.conc_input, 1, 1)
58
59          self.box.setLayout(self.layout)
60          self.box_layout.addWidget(self.box, 0, 0)
61
62
63  class RateResultBox(QWidget):
64      def __init__(self):
65          super(QWidget, self).__init__()
66
67          self.layout = QGridLayout()
68
69          self.box = QFrame(self)
70          self.box.setFrameStyle(0x0001)
71          self.box.setLineWidth(3)
72
73          self.box_layout = QGridLayout()
74
75          self.total_order_label = QLabel("Total order of reaction: ")
76          self.rate_constant_label = QLabel("Rate constant (k): ")
77          self.rate_label = QLabel("Rate of reaction: ")
78
79          self.total_order_input = QLineEdit()
80          self.rate_constant_input = QLineEdit()
81          self.rate_input = QLineEdit()
82
83          self.layout.addWidget(self.total_order_label, 0, 0)
84          self.layout.addWidget(self.total_order_input, 0, 1)
85          self.layout.addWidget(self.rate_constant_label, 1, 0)
86          self.layout.addWidget(self.rate_constant_input, 1, 1)
87          self.layout.addWidget(self.rate_label, 2, 0)
88          self.layout.addWidget(self.rate_input, 2, 1)
89
90          self.box.setLayout(self.layout)
91          self.box_layout.addWidget(self.box, 0, 0)
```

Listing 8: gui_comps.py File Program Code

A.3 ChemCalculator File

```
from PyQt6.QtGui import QFont
from PyQt6.QtWidgets import QGridLayout, QWidget, QPushButton, QLineEdit,
    QLabel, QComboBox, QHBoxLayout, QTabWidget, \
    QVBoxLayout, QMessageBox

from math import log

import re

from gui_comps import RateBox, RateResultBox


def is_numeric(user_input):
    if not user_input:
        return True
    # Regular expression to match integers or decimals
    pattern = r'^[-+]?[0-9]*\.?[0-9]+$'
    return bool(re.match(pattern, user_input))


def find_empty_input(input_list: list[QLineEdit]) -> QLineEdit | None:
    """
    An algorithm for finding the empty input out of a list of inputs.
    Only works when looking for a single empty input.
    """

    count = 0
    empty = None

    for i in range(len(input_list)):
        if not input_list[i].text().strip():
            count += 1
            empty = input_list[i]

    if count == 1:
        return empty


def check_invalid_symbol(input_list: list[QLineEdit]) -> bool:
    """
    An algorithm for checking for invalid symbols.
    """

    invalid = False

    for i in range(len(input_list)):
        if not is_numeric(input_list[i].text().strip()):
            invalid = True

    return invalid


def show_dialog(message):
    dlg = QMessageBox()
    dlg.setWindowTitle("Invalid Input!")
    dlg.setText(f"Invalid user input!\n {message}")
```

```
    dlg.setIcon(QMessageBox.Icon.Critical)
    button = dlg.exec()

    if button == QMessageBox.StandardButton.Ok:
        print("OK!")


class ChemCalculator(QWidget):
    def __init__(self):
        super(QWidget, self).__init__()

        self.side_bar_layout = QVBoxLayout()

        self.moles_calc = MolesCalculator()
        self.concentration_calc = ConcCalculator()
        self.avogadro_calc = AvogadroCalculator()
        self.ideal_gas_law_calc = IdealGasLawCalculator()
        self.equilibrium_calc = EquilibriumCalculator()
        self.gibbs_calc = GibbsFreeEnergyCalculator()
        self.specific_heat_calc = SpecificHeatCalculator()
        self.rate_calc = RateCalculator()

        # Create buttons
        self.moles_tab_button = QPushButton("Moles")
        self.conc_tab_button = QPushButton("Concentration")
        self.avogadro_tab_button = QPushButton("Avogadro's Calculator")
        self.ideal_gas_tab_button = QPushButton("Ideal Gas Equation")
        self.equilibrium_tab_button = QPushButton("Equilibrium Constant")
        self.gibbs_free_energy_tab_button = QPushButton("Gibbs Free Energy
    Calculator")
        self.specific_heat_tab_button = QPushButton("Specific Heat
    Calculator")
        self.rate_tab_button = QPushButton("Rate Constant")

        self.moles_tab_button.clicked.connect(self.moles_action)
        self.conc_tab_button.clicked.connect(self.conc_action)
        self.avogadro_tab_button.clicked.connect(self.avogadro_action)
        self.ideal_gas_tab_button.clicked.connect(self.ideal_gas_action)
        self.equilibrium_tab_button.clicked.connect(self.equilibrium_action)
        self.gibbs_free_energy_tab_button.clicked.connect(self.
    gibbs_free_energy_action)
        self.specific_heat_tab_button.clicked.connect(self.
    specific_heat_action)
        self.rate_tab_button.clicked.connect(self.rate_action)

        # Create tabs
        self.moles_tab = QWidget()
        self.conc_tab = QWidget()
        self.avogadro_tab = QWidget()
        self.ideal_gas_tab = QWidget()
        self.equilibrium_tab = QWidget()
        self.gibbs_free_energy_tab = QWidget()
        self.specific_heat_tab = QWidget()
        self.rate_tab = QWidget()

        # Initialise moles tab in sidebar
        self.moles_tab.setLayout(self.moles_calc.moles_layout)

        # Initialise concentration tab in sidebar
```

```
111         self.conc_tab.setLayout(self.concentration_calc.layout)
112 
113         # Initialise avogadro's calculator tab in sidebar
114         self.avogadro_tab.setLayout(self.avogadro_calc.layout)
115 
116         # Initialise igl tab in sidebar
117         self.ideal_gas_tab.setLayout(self.ideal_gas_law_calc.
    ideal_gas_layout)
118 
119         # Initialise equilibrium constant calculator tab
120         self.equilibrium_tab.setLayout(self.equilibrium_calc.layout)
121 
122         # Initialise gibbs free energy calculator
123         self.gibbs_free_energy_tab.setLayout(self.gibbs_calc.layout)
124 
125         # Initialise specific heat energy calculator
126         self.specific_heat_tab.setLayout(self.specific_heat_calc.layout)
127 
128         # Initialise rate constant calculator
129         self.rate_tab.setLayout(self.rate_calc.layout)
130 
131         # Add buttons to sidebar layout
132         self.side_bar_layout.addWidget(self.moles_tab_button)
133         self.side_bar_layout.addWidget(self.conc_tab_button)
134         self.side_bar_layout.addWidget(self.avogadro_tab_button)
135         self.side_bar_layout.addWidget(self.ideal_gas_tab_button)
136         self.side_bar_layout.addWidget(self.equilibrium_tab_button)
137         self.side_bar_layout.addWidget(self.gibbs_free_energy_tab_button)
138         self.side_bar_layout.addWidget(self.specific_heat_tab_button)
139         self.side_bar_layout.addWidget(self.rate_tab_button)
140 
141         self.side_bar_widget = QWidget()
142         self.side_bar_widget.setLayout(self.side_bar_layout)
143 
144         self.page_widget = QTabWidget()
145 
146         self.page_widget.addTab(self.moles_tab, "")
147         self.page_widget.addTab(self.conc_tab, "")
148         self.page_widget.addTab(self.avogadro_tab, "")
149         self.page_widget.addTab(self.ideal_gas_tab, "")
150         self.page_widget.addTab(self.equilibrium_tab, "")
151         self.page_widget.addTab(self.gibbs_free_energy_tab, "")
152         self.page_widget.addTab(self.specific_heat_tab, "")
153         self.page_widget.addTab(self.rate_tab, "")
154 
155         self.page_widget.setCurrentIndex(0)
156         self.page_widget.setStyleSheet('''QTabBar::tab{
157         width: 0;
158         height: 0;
159         margin: 0;
160         padding: 0;
161         border: none;
162         }''')
163 
164         self.main_layout = QHBoxLayout()
165         self.main_layout.addWidget(self.side_bar_widget)
166         self.main_layout.addWidget(self.page_widget)
167 
168         self.main_widget = QWidget()
```

```
169         self.main_widget.setLayout(self.main_layout)
170
171     # Define actions for each button
172
173     def moles_action(self):
174         self.page_widget.setCurrentIndex(0)
175
176     def conc_action(self):
177         self.page_widget.setCurrentIndex(1)
178
179     def avogadro_action(self):
180         self.page_widget.setCurrentIndex(2)
181
182     def ideal_gas_action(self):
183         self.page_widget.setCurrentIndex(3)
184
185     def equilibrium_action(self):
186         self.page_widget.setCurrentIndex(4)
187
188     def gibbs_free_energy_action(self):
189         self.page_widget.setCurrentIndex(5)
190
191     def specific_heat_action(self):
192         self.page_widget.setCurrentIndex(6)
193
194     def rate_action(self):
195         self.page_widget.setCurrentIndex(7)
196
197
198 class MolesCalculator(QWidget):
199     def __init__(self):
200         super(QWidget, self).__init__()
201         self.moles_layout = QGridLayout()
202
203         # Unit conversions
204         self.mass_conversions = {
205             "mg": 0.001,
206             "g": 1,
207             "kg": 1000,
208             "t": 1000000
209         }
210
211         self.mole_conversions = {
212             "µmol": 0.000001,
213             "mmol": 0.001,
214             "mol": 1,
215         }
216
217         self.volume_conversions = {
218             "cm3": 0.001,
219             "dm3": 1.0,
220             "m3": 1000.0,
221
222         }
223
224         # Initialise moles calculation Layout
225         self.moles_label = QLabel("Moles:")
226         self.mass_label = QLabel("Mass:")
227         self.mr_label = QLabel("Molecular weight:")
```

```
228
229         self.moles_input = QLineEdit()
230         self.mass_input = QLineEdit()
231         self.mr_input = QLineEdit()
232
233         self.input_list = [self.moles_input, self.mass_input, self.mr_input]
234
235         self.calculate_button = QPushButton("Calculate")
236         self.calculate_button.clicked.connect(self.calculate)
237
238         self.mass_unit_dropdown = QComboBox()
239         self.mass_unit_dropdown.addItem("mg")
240         self.mass_unit_dropdown.addItem("g")
241         self.mass_unit_dropdown.addItem("kg")
242         self.mass_unit_dropdown.addItem("t")
243
244         self.mass_unit_dropdown.setCurrentIndex(1)
245
246         self.moles_unit_dropdown = QComboBox()
247         self.moles_unit_dropdown.addItem("µmol")
248         self.moles_unit_dropdown.addItem("mmol")
249         self.moles_unit_dropdown.addItem("mol")
250
251         self.moles_unit_dropdown.setCurrentIndex(2)
252
253         self.get_moles_layout()
254
255     def calculate(self):
256         """
257         This function routes to the correct calculation, which is then
    performed.
258         """
259
260         if not find_empty_input(self.input_list.copy()):
261             show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
262             return
263         elif check_invalid_symbol(self.input_list.copy()):
264             show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
265             return
266
267         mass_unit = self.mass_conversions[self.mass_unit_dropdown.
    currentText()]
268         moles_unit = self.mole_conversions[self.moles_unit_dropdown.
    currentText()]
269
270         to_calc = find_empty_input(self.input_list.copy())
271
272         if to_calc is self.moles_input:
273             self.calculate_moles(mass_unit)
274         elif to_calc is self.mass_input:
275             self.calculate_mass(moles_unit)
276         elif to_calc is self.mr_input:
277             self.calculate_mr(mass_unit, moles_unit)
278         else:
279             return
280
281     def calculate_moles(self, mass_unit):
```

```
282         """
283         Calculates the moles and calls for an update of the gui input and
    adjusts the moles unit dropdown.
284         """
285 
286         moles = (float(self.mass_input.text()) * mass_unit) / float(self.
    mr_input.text())
287         self.update_input(moles)
288         self.moles_unit_dropdown.setCurrentIndex(2)
289 
290     def calculate_mass(self, moles_unit):
291         """
292         Calculates the mass and calls for an update of the gui input and
    adjusts the moles unit dropdown.
293         """
294 
295         mass = (float(self.moles_input.text()) * moles_unit) * float(
296             self.mr_input.text())
297         self.update_input(mass)
298         self.mass_unit_dropdown.setCurrentIndex(1)
299 
300     def calculate_mr(self, mass_unit, moles_unit):
301         """
302         Calculates the mr and calls for an update of the gui input.
303         """
304 
305         mr = (float(self.mass_input.text()) * mass_unit) / (
306                 float(self.moles_input.text()) * moles_unit)
307         self.update_input(mr)
308 
309     def update_input(self, result):
310         """
311         Uses the find_empty_input() function to find the empty input, and
    then updates it using the result parameter.
312         """
313 
314         find_empty_input(self.input_list.copy()).setText(str(result))
315 
316     def get_moles_layout(self):
317         f"""
318         This function adds all the essential widgets to the {self.
    moles_layout}
319         """
320 
321         self.moles_layout.addWidget(self.moles_label, 0, 0)
322         self.moles_layout.addWidget(self.mass_label, 1, 0)
323         self.moles_layout.addWidget(self.mr_label, 2, 0)
324 
325         self.moles_layout.addWidget(self.moles_input, 0, 1)
326         self.moles_layout.addWidget(self.mass_input, 1, 1)
327         self.moles_layout.addWidget(self.mr_input, 2, 1)
328 
329         self.moles_layout.addWidget(self.moles_unit_dropdown, 0, 2)
330         self.moles_layout.addWidget(self.mass_unit_dropdown, 1, 2)
331 
332         self.moles_layout.addWidget(self.calculate_button, 3, 1)
333 
334 
335 class ConcCalculator(QWidget):
```

```
336     def __init__(self):
337         super(QWidget, self).__init__()
338         self.layout = QGridLayout()
339 
340         # Unit conversions
341         self.mole_conversions = {
342             "µmol": 0.000001,
343             "mmol": 0.001,
344             "mol": 1,
345         }
346 
347         self.volume_conversions = {
348             "cm3": 0.001,
349             "dm3": 1.0,
350             "m3": 1000.0,
351 
352         }
353 
354         # Initialise concentration calculation Layout
355         self.conc_label = QLabel("Concentration:")
356         self.moles_label = QLabel("Moles:")
357         self.vol_label = QLabel("Volume:")
358 
359         self.conc_input = QLineEdit()
360         self.moles_input = QLineEdit()
361         self.vol_input = QLineEdit()
362 
363         self.input_list = [self.conc_input, self.moles_input, self.vol_input
    ]
364 
365         self.calculate_button = QPushButton("Calculate")
366         self.calculate_button.clicked.connect(self.calculate)
367 
368         self.moles_unit_dropdown = QComboBox()
369         self.moles_unit_dropdown.addItem("µmol")
370         self.moles_unit_dropdown.addItem("mmol")
371         self.moles_unit_dropdown.addItem("mol")
372 
373         self.moles_unit_dropdown.setCurrentIndex(2)
374 
375         self.vol_unit_drop_down = QComboBox()
376         self.vol_unit_drop_down.addItem("cm3")
377         self.vol_unit_drop_down.addItem("dm3")
378         self.vol_unit_drop_down.addItem("m3")
379 
380         self.vol_unit_drop_down.setCurrentIndex(1)
381 
382         self.get_conc_layout()
383 
384     def calculate(self):
385         """
386         This function routes to the correct calculation, which is then
    performed.
387         """
388 
389         if not find_empty_input(self.input_list.copy()):
390             show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
391             return
```

```
392         elif check_invalid_symbol(self.input_list.copy()):
393             show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
394             return
395 
396         moles_unit = self.mole_conversions[self.moles_unit_dropdown.
    currentText()]
397         vol_unit = self.volume_conversions[self.vol_unit_drop_down.
    currentText()]
398 
399         to_calc = find_empty_input(self.input_list.copy())
400 
401         if to_calc is self.moles_input:
402             self.calculate_moles(vol_unit)
403         elif to_calc is self.vol_input:
404             self.calculate_vol(moles_unit)
405         elif to_calc is self.conc_input:
406             self.calculate_conc(moles_unit, vol_unit)
407         else:
408             return
409 
410     def calculate_moles(self, vol_unit):
411         """
412         Calculates the moles and calls for an update of the gui input and
    adjusts the moles unit dropdown.
413         """
414 
415         moles = float(self.conc_input.text()) * (
416                 float(self.vol_input.text()) * vol_unit)
417         self.update_input(moles)
418         self.moles_unit_dropdown.setCurrentIndex(2)
419 
420     def calculate_vol(self, moles_unit):
421         """
422         Calculates the volume and calls for an update of the gui input.
423         """
424 
425         vol = (float(self.moles_input.text()) * moles_unit) / float(
426             self.conc_input.text())
427         self.update_input(vol)
428         self.vol_unit_drop_down.setCurrentIndex(1)
429 
430     def calculate_conc(self, moles_unit, vol_unit):
431         """
432         Calculates the concentration and calls for an update of the gui
    input.
433         """
434 
435         conc = (float(self.moles_input.text()) * moles_unit) / (
436                 float(self.vol_input.text()) * vol_unit)
437         self.update_input(conc)
438 
439     def update_input(self, result):
440         """
441         Uses the find_empty_input() function to find the empty input, and
    then updates it using the result parameter.
442         """
443 
444         find_empty_input(self.input_list).setText(str(result))
```

```
445
446     def get_conc_layout(self):
447         """
448            This function adds all the essential widgets to the layout.
449            """
450
451         self.layout.addWidget(self.conc_label, 0, 0)
452         self.layout.addWidget(self.moles_label, 1, 0)
453         self.layout.addWidget(self.vol_label, 2, 0)
454
455         self.layout.addWidget(self.conc_input, 0, 1)
456         self.layout.addWidget(self.moles_input, 1, 1)
457         self.layout.addWidget(self.vol_input, 2, 1)
458
459         self.layout.addWidget(self.moles_unit_dropdown, 1, 2)
460         self.layout.addWidget(self.vol_unit_drop_down, 2, 2)
461
462         self.layout.addWidget(self.calculate_button, 3, 1)
463
464
465 class AvogadroCalculator(QWidget):
466     def __init__(self):
467         super(QWidget, self).__init__()
468         self.layout = QGridLayout()
469
470         self.avogadros_constant = 6.02214076
471
472         # Unit conversions
473         self.mass_conversions = {
474             "mg": 0.001,
475             "g": 1,
476             "kg": 1000,
477             "t": 1000000
478         }
479
480         self.mole_conversions = {
481             "µmol": 0.000001,
482             "mmol": 0.001,
483             "mol": 1,
484         }
485
486         self.volume_conversions = {
487             "cm3": 0.001,
488             "dm3": 1.0,
489             "m3": 1000.0,
490
491         }
492
493         # Initialise Avogadro's calculator
494         self.mass_label = QLabel("Mass:")
495         self.moles_label = QLabel("Moles:")
496         self.molecular_weight_label = QLabel("Molecular weight:")
497         self.num_atoms_label = QLabel("Number of atoms:")
498
499         self.mass_input = QLineEdit()
500         self.moles_input = QLineEdit()
501         self.molecular_weight_input = QLineEdit()
502         self.num_atoms_input = QLineEdit()
503
```

```
504         self.input_list = [self.mass_input, self.moles_input, self.
    molecular_weight_input, self.num_atoms_input]
505 
506         self.calculate_button = QPushButton("Calculate")
507         self.calculate_button.clicked.connect(self.calculate)
508 
509         self.mass_unit_dropdown = QComboBox()
510         self.mass_unit_dropdown.addItem("mg")
511         self.mass_unit_dropdown.addItem("g")
512         self.mass_unit_dropdown.addItem("kg")
513         self.mass_unit_dropdown.addItem("t")
514 
515         self.mass_unit_dropdown.setCurrentIndex(1)
516 
517         # Initialise Atom Economy calculator
518 
519         self.get_layout()
520 
521     def get_layout(self):
522         f"""
523         This function adds all the essential widgets to the {self.layout}
524         """
525 
526         self.layout.addWidget(self.mass_label, 0, 0)
527         self.layout.addWidget(self.moles_label, 1, 0)
528         self.layout.addWidget(self.molecular_weight_label, 2, 0)
529         self.layout.addWidget(self.num_atoms_label, 3, 0)
530 
531         self.layout.addWidget(self.mass_input, 0, 1)
532         self.layout.addWidget(self.moles_input, 1, 1)
533         self.layout.addWidget(self.molecular_weight_input, 2, 1)
534         self.layout.addWidget(self.num_atoms_input, 3, 1)
535 
536         self.layout.addWidget(self.mass_unit_dropdown, 0, 2)
537 
538         self.layout.addWidget(self.calculate_button, 4, 1)
539 
540     def calculate(self):
541         mass_unit = self.mass_unit_dropdown.currentText()
542 
543         if not find_empty_input(self.input_list.copy()):
544             if not self.mass_input.text():
545                 if self.moles_input.text() and self.molecular_weight_input.
    text():
546                     self.update_mass(str(self.calculate_mass()))
547             if not self.moles_input.text():
548                 if self.num_atoms_input.text():
549                     self.update_moles(str(self.calculate_moles(mass_unit)))
550                 elif self.mass_input.text() and self.molecular_weight_input.
    text():
551                     self.update_moles(str(self.calculate_moles(mass_unit)))
552             if not self.molecular_weight_input.text():
553                 if self.mass_input.text() and self.moles_input.text():
554                     self.update_mol_weight(str(self.calculate_mol_weight(
    mass_unit)))
555             if self.num_atoms_input.text() == "":
556                 if self.moles_input.text() != "":
557                     self.update_num_atoms(str(self.calculate_num_atoms()))
558                 else:
```

```
559                     show_dialog("Must leave one input line empty for it to
    be calculated!")
560                     return
561             else:
562                 show_dialog("Must leave one input line empty for it to be
    calculated!")
563                 return
564         elif check_invalid_symbol(self.input_list.copy()):
565             show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
566             return
567 
568         to_calc = find_empty_input(self.input_list.copy())
569 
570         if to_calc is self.mass_input:
571             self.update_mass(str(self.calculate_mass()))
572         if to_calc is self.moles_input:
573             self.update_moles(str(self.calculate_moles(mass_unit)))
574         if to_calc is self.molecular_weight_input:
575             self.update_mol_weight(str(self.calculate_mol_weight(mass_unit))
    )
576         if to_calc is self.num_atoms_input:
577             self.update_num_atoms(str(self.calculate_num_atoms()))
578 
579     def calculate_mass(self):
580         mass = float(self.moles_input.text()) * float(self.
    molecular_weight_input.text())
581         return mass
582 
583     def calculate_moles(self, mass_unit):
584         try:
585             moles = float(self.num_atoms_input.text()) / self.
    avogadros_constant
586         except ValueError:
587             try:
588                 moles = (float(self.mass_input.text()) * self.
    mass_conversions[mass_unit]) / float(
589                     self.molecular_weight_input.text())
590                 return moles
591             except ValueError:
592                 return ""
593         return moles
594 
595     def calculate_mol_weight(self, mass_unit):
596         mol_weight = (float(self.mass_input.text()) * self.mass_conversions[
    mass_unit]) / float(
597             self.moles_input.text())
598         return mol_weight
599 
600     def calculate_num_atoms(self):
601         num_atoms = float(self.moles_input.text()) * self.avogadros_constant
602         return num_atoms
603 
604     def update_mass(self, mass):
605         self.mass_input.setText(mass)
606 
607     def update_moles(self, moles):
608         self.moles_input.setText(moles)
609 
```

```
610     def update_mol_weight(self, mol_weight):
611         self.molecular_weight_input.setText(mol_weight)
612 
613     def update_num_atoms(self, num_atoms):
614         self.num_atoms_input.setText(num_atoms)
615 
616 
617 class IdealGasLawCalculator(QWidget):
618     def __init__(self):
619         super(QWidget, self).__init__()
620         self.ideal_gas_layout = QGridLayout()
621 
622         # Initialise Ideal Gas Law (IGL) properties
623         self.ideal_gas_constant = 8.314
624 
625         self.pressure_input = QLineEdit()
626         self.volume_input = QLineEdit()
627         self.temperature_input = QLineEdit()
628         self.moles_input = QLineEdit()
629 
630         self.input_list = [self.pressure_input, self.volume_input, self.
    temperature_input, self.moles_input]
631 
632         self.pressure_label_igl = QLabel("Pressure:")
633         self.volume_label_igl = QLabel("Volume:")
634         self.temperature_label_igl = QLabel("Temperature:")
635         self.moles_label_igl = QLabel("Amount of substance - moles:")
636         self.calculate_button_igl = QPushButton('Calculate')
637 
638         self.pressure_conversions = {
639             "Pa": 1.0,
640             "kPa": 1000.0,
641         }
642 
643         self.temperature_conversions = {
644             "C": 273.15,
645             "K": 0.0,
646         }
647 
648         self.volume_conversions = {
649             "cm3": 0.000001,
650             "dm3": 0.001,
651             "m3": 1.0,
652 
653         }
654 
655         self.pressure_drop_down_igl = QComboBox()
656         self.volume_drop_down_igl = QComboBox()
657         self.temperature_drop_down_igl = QComboBox()
658 
659         self.ideal_gas_layout.addWidget(self.pressure_label_igl, 0, 0)
660         self.ideal_gas_layout.addWidget(self.pressure_input, 0, 1)
661         self.ideal_gas_layout.addWidget(self.pressure_drop_down_igl, 0, 2)
662 
663         self.ideal_gas_layout.addWidget(self.volume_label_igl, 1, 0)
664         self.ideal_gas_layout.addWidget(self.volume_input, 1, 1)
665         self.ideal_gas_layout.addWidget(self.volume_drop_down_igl, 1, 2)
666 
667         self.ideal_gas_layout.addWidget(self.temperature_label_igl, 2, 0)
```

```
668         self.ideal_gas_layout.addWidget(self.temperature_input, 2, 1)
669         self.ideal_gas_layout.addWidget(self.temperature_drop_down_igl, 2,
    2)
670 
671         self.ideal_gas_layout.addWidget(self.moles_label_igl, 3, 0)
672         self.ideal_gas_layout.addWidget(self.moles_input, 3, 1)
673         self.ideal_gas_layout.addWidget(self.calculate_button_igl, 4, 0)
674 
675         self.pressure_drop_down_igl.addItem("Pa")
676         self.pressure_drop_down_igl.addItem("kPa")
677 
678         self.pressure_drop_down_igl.setCurrentIndex(0)
679 
680         self.volume_drop_down_igl.addItem("cm3")
681         self.volume_drop_down_igl.addItem("dm3")
682         self.volume_drop_down_igl.addItem("m3")
683 
684         self.volume_drop_down_igl.setCurrentIndex(2)
685 
686         self.temperature_drop_down_igl.addItem("C")
687         self.temperature_drop_down_igl.addItem("K")
688 
689         self.temperature_drop_down_igl.setCurrentIndex(1)
690 
691         self.calculate_button_igl.clicked.connect(self.
    calculate_ideal_gas_law)
692 
693     def calculate_ideal_gas_law(self):
694         pressure_unit = self.pressure_drop_down_igl.currentText()
695         temperature_unit = self.temperature_drop_down_igl.currentText()
696         volume_unit = self.volume_drop_down_igl.currentText()
697 
698         if not find_empty_input(self.input_list.copy()):
699             show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
700             return
701         elif check_invalid_symbol(self.input_list.copy()):
702             show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
703             return
704 
705         to_calc = find_empty_input(self.input_list.copy())
706 
707         if to_calc is self.pressure_input:
708             self.pressure_input.setText(str(self.calculate_pressure(
    temperature_unit, volume_unit)))
709         elif to_calc is self.volume_input:
710             self.volume_input.setText(str(self.calculate_volume(
    temperature_unit, pressure_unit)))
711         elif to_calc is self.temperature_input:
712             self.temperature_input.setText(str(self.calculate_temperature(
    pressure_unit, volume_unit)))
713         elif to_calc is self.moles_input:
714             self.moles_input.setText(str(self.calculate_moles(
    temperature_unit, volume_unit, pressure_unit)))
715 
716     def calculate_pressure(self, temperature_unit, volume_unit):
717         pressure = (float(self.moles_input.text()) * self.ideal_gas_constant
     * (float(
```

```
718             self.temperature_input.text()) + self.temperature_conversions[
    temperature_unit])) / (
719                             float(self.volume_input.text()) * self.
    volume_conversions[volume_unit])
720         return pressure
721
722     def calculate_volume(self, temperature_unit, pressure_unit):
723         volume = (float(self.moles_input.text()) * self.ideal_gas_constant *
     (float(
724             self.temperature_input.text()) + self.temperature_conversions[
    temperature_unit])) / (
725                           float(self.pressure_input.text()) * self.
    pressure_conversions[pressure_unit])
726         return volume
727
728     def calculate_temperature(self, pressure_unit, volume_unit):
729         temperature = ((float(self.pressure_input.text()) * self.
    pressure_conversions[pressure_unit]) * (
730                 float(
731                     self.volume_input.text()) * self.volume_conversions[
    volume_unit])) / (
732                                float(self.moles_input.text()) * self.
    ideal_gas_constant)
733         return temperature
734
735     def calculate_moles(self, temperature_unit, volume_unit, pressure_unit):
736         moles = ((float(self.pressure_input.text()) * self.
    pressure_conversions[pressure_unit]) * (
737                 float(self.volume_input.text()) * self.volume_conversions[
    volume_unit])) / (
738                           self.ideal_gas_constant * (float(self.
    temperature_input.text()) +
739                                                      self.
    temperature_conversions[temperature_unit]))
740         return moles
741
742
743 class EquilibriumCalculator(QWidget):
744     def __init__(self):
745         super(QWidget, self).__init__()
746
747         self.layout = QGridLayout()
748
749         # Set up line edits and labels
750         self.equation_label = QLabel("a[A] + b[B] = c[C] + d[D]")
751         self.equation_label.setFont(QFont("SansSerif", 22))
752         self.equation_label.setStyleSheet("color: darkGray;")
753
754         self.calc_button = QPushButton("Calculate")
755         self.calc_button.clicked.connect(self.check_empty_input)
756
757         self.conc_a_label = QLabel("Concentration [A]:")
758         self.conc_a = QLineEdit()
759
760         self.conc_b_label = QLabel("Concentration [B]:")
761         self.conc_b = QLineEdit()
762
763         self.conc_c_label = QLabel("Concentration [C]:")
764         self.conc_c = QLineEdit()
```

```
765
766         self.conc_d_label = QLabel("Concentration [D]:")
767         self.conc_d = QLineEdit()
768
769         self.coeff_a_label = QLabel("Coefficient a:")
770         self.coeff_a = QLineEdit()
771
772         self.coeff_b_label = QLabel("Coefficient b:")
773         self.coeff_b = QLineEdit()
774
775         self.coeff_c_label = QLabel("Coefficient c:")
776         self.coeff_c = QLineEdit()
777
778         self.coeff_d_label = QLabel("Coefficient d:")
779         self.coeff_d = QLineEdit()
780
781         self.equilibrium_constant_label = QLabel("Equilibrium Constant k:")
782         self.equilibrium_constant = QLineEdit()
783
784         self.input_list = [self.conc_a, self.conc_b, self.conc_c, self.
    conc_d, self.coeff_a, self.coeff_b, self.coeff_c,
785                            self.coeff_d, self.equilibrium_constant]
786         self.concentration_list = [self.conc_a, self.conc_b, self.conc_c,
    self.conc_d]
787         self.coeff_list = [self.coeff_a, self.coeff_b, self.coeff_c, self.
    coeff_d]
788
789         self.calculated_value = None
790
791         self.layout.addWidget(self.equation_label, 0, 0)
792         self.layout.addWidget(self.calc_button, 10, 0, 2, 0)
793
794         self.layout.addWidget(self.conc_a_label, 1, 0)
795         self.layout.addWidget(self.conc_a, 1, 1)
796         self.layout.addWidget(self.coeff_a_label, 2, 0)
797         self.layout.addWidget(self.coeff_a, 2, 1)
798         self.layout.addWidget(self.conc_b_label, 3, 0)
799         self.layout.addWidget(self.conc_b, 3, 1)
800         self.layout.addWidget(self.coeff_b_label, 4, 0)
801         self.layout.addWidget(self.coeff_b, 4, 1)
802         self.layout.addWidget(self.conc_c_label, 5, 0)
803         self.layout.addWidget(self.conc_c, 5, 1)
804         self.layout.addWidget(self.coeff_c_label, 6, 0)
805         self.layout.addWidget(self.coeff_c, 6, 1)
806         self.layout.addWidget(self.conc_d_label, 7, 0)
807         self.layout.addWidget(self.conc_d, 7, 1)
808         self.layout.addWidget(self.coeff_d_label, 8, 0)
809         self.layout.addWidget(self.coeff_d, 8, 1)
810         self.layout.addWidget(self.equilibrium_constant_label, 9, 0)
811         self.layout.addWidget(self.equilibrium_constant, 9, 1)
812
813     def check_empty_input(self):
814         empty_count = 0
815         empty_input = None
816
817         if not find_empty_input(self.input_list.copy()):
818             show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
819             return
```

```
820         elif check_invalid_symbol(self.input_list.copy()):
821             show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
822             return
823 
824         for item in self.input_list:
825             if item.text().strip() == "":
826                 empty_count += 1
827                 empty_input = item
828 
829         if empty_input in self.concentration_list and empty_count == 1 or
    self.calculated_value in self.concentration_list:
830             self.calculate_concentration(empty_input)
831         elif empty_input in self.coeff_list and empty_count == 1 or self.
    calculated_value in self.coeff_list:
832             self.calculate_coefficient(empty_input)
833         elif empty_input == self.equilibrium_constant and empty_count == 1
    or self.calculated_value == self.equilibrium_constant:
834             self.calculate_constant()
835         else:
836             show_dialog("")
837             return
838 
839     def calculate_constant(self):
840         try:
841             k = ((float(self.conc_c.text()) ** float(self.coeff_c.text())) *
     (
842                     float(self.conc_d.text()) ** float(self.coeff_d.text()))
    ) / (
843                         (float(self.conc_a.text()) ** float(self.coeff_a.
    text())) *
844                         (float(self.conc_b.text()) ** float(self.coeff_b.
    text()))))
845         except OverflowError:
846             print("Overflow Error, inputted numbers are too large")
847             show_dialog("Overflow Error, inputted numbers are too large!")
848             return
849         self.calculated_value = self.equilibrium_constant
850         self.update_gui(self.equilibrium_constant, k)
851 
852     def calculate_concentration(self, to_find):
853         if to_find == self.conc_a or to_find == self.calculated_value:
854             try:
855                 conc = ((float(self.conc_c.text()) ** float(self.coeff_c.
    text())) * (
856                         float(self.conc_d.text()) ** float(self.coeff_d.text
    ())) / (
857                                 float(self.equilibrium_constant.text()) * (
858                                 float(self.conc_b.text()) ** float(self.
    coeff_b.text()))))
859                 if float(self.coeff_a.text()) > 1:
860                     conc = conc ** (1 / float(self.coeff_a.text()))
861             except OverflowError:
862                 print("Overflow Error, inputted numbers too large")
863                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
864                 return
865             self.calculated_value = to_find
866             self.update_gui(to_find, conc)
```

```
867             return
868 
869         if to_find == self.conc_b or to_find == self.calculated_value:
870             try:
871                 conc = ((float(self.conc_c.text()) ** float(self.coeff_c.
    text())) * (
872                         float(self.conc_d.text()) ** float(self.coeff_d.text
    ())) / (
873                                 float(self.equilibrium_constant.text()) * (
874                                 float(self.conc_a.text()) ** float(self.
    coeff_a.text())))
875                 if float(self.coeff_b.text()) > 1:
876                     conc = conc ** (1 / float(self.coeff_b.text()))
877             except OverflowError:
878                 print("Overflow Error, inputted numbers too large")
879                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
880                 return
881             self.calculated_value = to_find
882             self.update_gui(to_find, conc)
883             return
884 
885         if to_find == self.conc_c or to_find == self.calculated_value:
886             try:
887                 conc = ((float(self.conc_a.text()) ** float(self.coeff_a.
    text())) * (
888                         float(self.conc_b.text()) ** float(self.coeff_b.text
    ())) * (
889                             float(self.equilibrium_constant.text())) / (
890                                 float(self.conc_d.text()) ** float(self.
    coeff_d.text())))
891                 if float(self.coeff_c.text()) > 1:
892                     conc = conc ** (1 / float(self.coeff_c.text()))
893             except OverflowError:
894                 print("Overflow Error, inputted numbers too large")
895                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
896                 return
897             self.calculated_value = to_find
898             self.update_gui(to_find, conc)
899             return
900 
901         if to_find == self.conc_d or to_find == self.calculated_value:
902             try:
903                 conc = ((float(self.conc_a.text()) ** float(self.coeff_a.
    text())) * (
904                         float(self.conc_b.text()) ** float(self.coeff_b.text
    ())) * (
905                             float(self.equilibrium_constant.text())) / (
906                                 float(self.conc_c.text()) ** float(self.
    coeff_c.text())))
907                 if float(self.coeff_d.text()) > 1:
908                     conc = conc ** (1 / float(self.coeff_d.text()))
909             except OverflowError:
910                 print("Overflow Error, inputted numbers too large")
911                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
912                 return
913             self.update_gui(to_find, conc)
```

```
914             return
915
916     def calculate_coefficient(self, to_find):
917         if to_find == self.coeff_a or to_find == self.calculated_value:
918             try:
919                 coeff = ((float(self.conc_c.text()) ** float(self.coeff_c.
    text())) * (
920                         float(self.conc_d.text()) ** float(self.coeff_d.text
    ()))) / (
921                                 float(self.equilibrium_constant.text()) * (
922                                 float(self.conc_b.text()) ** float(self.
    coeff_b.text())))
923                 coeff = int(log(coeff, float(self.conc_a.text())))
924             except OverflowError:
925                 print("Overflow Error, inputted numbers too large")
926                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
927                 return
928             self.calculated_value = to_find
929             self.update_gui(to_find, coeff)
930             return
931
932         if to_find == self.coeff_b or to_find == self.calculated_value:
933             try:
934                 coeff = ((float(self.conc_c.text()) ** float(self.coeff_c.
    text())) * (
935                         float(self.conc_d.text()) ** float(self.coeff_d.text
    ()))) / (
936                                 float(self.equilibrium_constant.text()) * (
937                                 float(self.conc_a.text()) ** float(self.
    coeff_a.text())))
938                 coeff = int(log(coeff, float(self.conc_b.text())))
939             except OverflowError:
940                 print("Overflow Error, inputted numbers too large")
941                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
942                 return
943             self.calculated_value = to_find
944             self.update_gui(to_find, coeff)
945             return
946
947         if to_find == self.coeff_c or to_find == self.calculated_value:
948             try:
949                 coeff = ((float(self.conc_a.text()) ** float(self.coeff_a.
    text())) * (
950                         float(self.conc_b.text()) ** float(self.coeff_b.text
    ())) * (
951                              float(self.equilibrium_constant.text())) / (
952                                  float(self.conc_d.text()) ** float(self.
    coeff_d.text())))
953                 coeff = int(log(coeff, float(self.conc_c.text())))
954             except OverflowError:
955                 print("Overflow Error, inputted numbers too large")
956                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
957                 return
958             self.calculated_value = to_find
959             self.update_gui(to_find, coeff)
960             return
```

```
961
962         if to_find == self.coeff_d or to_find == self.calculated_value:
963             try:
964                 coeff = ((float(self.conc_a.text()) ** float(self.coeff_a.
    text())) * (
965                         float(self.conc_b.text()) ** float(self.coeff_b.text
    ())) * (
966                             float(self.equilibrium_constant.text())) / (
967                                 float(self.conc_c.text()) ** float(self.
    coeff_c.text())))
968                 coeff = int(log(coeff, float(self.conc_d.text())))
969             except OverflowError:
970                 print("Overflow Error, inputted numbers too large")
971                 show_dialog("Overflow Error, inputted numbers are too large!
    ")
972                 return
973             self.calculated_value = to_find
974             self.update_gui(to_find, coeff)
975             return
976
977     def update_gui(self, empty_input, value):
978         if empty_input is self.calculated_value:
979             empty_input.setText(str(value))
980
981
982 class GibbsFreeEnergyCalculator(QWidget):
983     def __init__(self):
984         super(QWidget, self).__init__()
985
986         self.layout = QGridLayout()
987
988         self.temperature_conversions = {
989             "C": 273.15,
990             "K": 0.0,
991         }
992
993         self.general_energy_conversions = {
994             "kJ": 1.0,
995             "J": 0.001
996         }
997
998         self.gibbs_free_energy_label = QLabel("Gibbs Free Energy (delta G):"
    )
999         self.enthalpy_change_label = QLabel("Enthalpy Change (delta H):")
1000         self.temp_label = QLabel("Temperature:")
1001         self.entropy_change_label = QLabel("Entropy Change (delta S):")
1002
1003         self.gibbs_free_energy_input = QLineEdit()
1004         self.enthalpy_change_input = QLineEdit()
1005         self.temp_input = QLineEdit()
1006         self.entropy_change_input = QLineEdit()
1007
1008         self.input_list = [self.gibbs_free_energy_input, self.
    entropy_change_input, self.temp_input,
1009                            self.enthalpy_change_input]
1010
1011         self.gibbs_free_energy_unit_dropdown = QComboBox()
1012         self.gibbs_free_energy_unit_dropdown.addItem("kJ")
1013         self.gibbs_free_energy_unit_dropdown.addItem("J")
```

```
1014
1015            self.gibbs_free_energy_unit_dropdown.setCurrentIndex(0)
1016
1017            self.enthalpy_change_unit_dropdown = QComboBox()
1018            self.enthalpy_change_unit_dropdown.addItem("kJ")
1019            self.enthalpy_change_unit_dropdown.addItem("J")
1020
1021            self.enthalpy_change_unit_dropdown.setCurrentIndex(0)
1022
1023            self.temp_unit_dropdown = QComboBox()
1024            self.temp_unit_dropdown.addItem("C")
1025            self.temp_unit_dropdown.addItem("K")
1026
1027            self.temp_unit_dropdown.setCurrentIndex(1)
1028
1029            self.entropy_change_unit_dropdown = QComboBox()
1030            self.entropy_change_unit_dropdown.addItem("kJ")
1031            self.entropy_change_unit_dropdown.addItem("J")
1032
1033            self.entropy_change_unit_dropdown.setCurrentIndex(1)
1034
1035            self.calculate_button = QPushButton("Calculate")
1036            self.calculate_button.clicked.connect(self.perform_calculation)
1037
1038            self.layout.addWidget(self.gibbs_free_energy_label, 0, 0)
1039            self.layout.addWidget(self.gibbs_free_energy_input, 0, 1)
1040            self.layout.addWidget(self.gibbs_free_energy_unit_dropdown, 0, 2)
1041
1042            self.layout.addWidget(self.enthalpy_change_label, 1, 0)
1043            self.layout.addWidget(self.enthalpy_change_input, 1, 1)
1044            self.layout.addWidget(self.enthalpy_change_unit_dropdown, 1, 2)
1045
1046            self.layout.addWidget(self.temp_label, 2, 0)
1047            self.layout.addWidget(self.temp_input, 2, 1)
1048            self.layout.addWidget(self.temp_unit_dropdown, 2, 2)
1049
1050            self.layout.addWidget(self.entropy_change_label, 3, 0)
1051            self.layout.addWidget(self.entropy_change_input, 3, 1)
1052            self.layout.addWidget(self.entropy_change_unit_dropdown, 3, 2)
1053
1054            self.layout.addWidget(self.calculate_button, 4, 0, 1, 3)
1055
1056        def update_gibbs_free_energy(self, free_energy):
1057            self.gibbs_free_energy_input.setText(free_energy)
1058
1059        def update_enthalpy_change(self, enthalpy_change):
1060            self.enthalpy_change_input.setText(enthalpy_change)
1061
1062        def update_temperature(self, temp):
1063            self.temp_input.setText(temp)
1064
1065        def update_entropy_change(self, entropy_change):
1066            self.entropy_change_input.setText(entropy_change)
1067
1068        def perform_calculation(self):
1069            free_energy_unit = self.gibbs_free_energy_unit_dropdown.currentText
    ()
1070            enthalpy_unit = self.enthalpy_change_unit_dropdown.currentText()
1071            temp_unit = self.temp_unit_dropdown.currentText()
```

```
1072            entropy_unit = self.entropy_change_unit_dropdown.currentText()
1073
1074            if not find_empty_input(self.input_list.copy()):
1075                show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
1076                return
1077            elif check_invalid_symbol(self.input_list.copy()):
1078                show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
1079                return
1080
1081            to_calc = find_empty_input(self.input_list.copy())
1082
1083            if to_calc is self.gibbs_free_energy_input:
1084                free_energy = self.calculate_free_energy_change(free_energy_unit
    , enthalpy_unit, temp_unit, entropy_unit)
1085                if free_energy:
1086                    self.update_gibbs_free_energy(str(free_energy))
1087            elif to_calc is self.enthalpy_change_input:
1088                enthalpy_change = self.calculate_enthalpy_change(
    free_energy_unit, enthalpy_unit, temp_unit, entropy_unit)
1089                if enthalpy_change:
1090                    self.update_enthalpy_change(str(enthalpy_change))
1091            elif to_calc is self.temp_input:
1092                temperature = self.calculate_temperature(free_energy_unit,
    enthalpy_unit, temp_unit, entropy_unit)
1093                if temperature:
1094                    self.update_temperature(str(temperature))
1095            elif to_calc is self.entropy_change_input:
1096                entropy_change = self.calculate_entropy_change(free_energy_unit,
     enthalpy_unit, temp_unit, entropy_unit)
1097                if entropy_change:
1098                    self.update_entropy_change(str(entropy_change))
1099
1100        def calculate_free_energy_change(self, free_energy_unit, enthalpy_unit,
    temp_unit, entropy_unit):
1101            try:
1102                free_energy = (float(self.enthalpy_change_input.text()) * self.
    general_energy_conversions[
1103                    enthalpy_unit]) - ((float(self.temp_input.text()) + self.
    temperature_conversions[temp_unit]) * (
1104                        float(self.entropy_change_input.text()) * self.
    general_energy_conversions[entropy_unit]))
1105                free_energy = free_energy * self.general_energy_conversions[
    free_energy_unit]
1106                return free_energy
1107            except ValueError:
1108                print("Value Error")
1109                show_dialog("Value Error!")
1110                return
1111
1112        def calculate_enthalpy_change(self, free_energy_unit, enthalpy_unit,
    temp_unit, entropy_unit):
1113            try:
1114                enthalpy_change = (float(self.gibbs_free_energy_input.text()) *
    self.general_energy_conversions[
1115                    free_energy_unit]) + ((float(self.temp_input.text()) + self.
    temperature_conversions[temp_unit]) * (
```

```
                    float(self.entropy_change_input.text()) * self.
    general_energy_conversions[entropy_unit]))
            enthalpy_change = enthalpy_change * self.
    general_energy_conversions[enthalpy_unit]
            return enthalpy_change
        except ValueError:
            print("Value Error")
            show_dialog("Value Error!")
            return

    def calculate_temperature(self, free_energy_unit, enthalpy_unit,
    temp_unit, entropy_unit):
        try:
            temperature = ((float(self.enthalpy_change_input.text()) * self.
    general_energy_conversions[
                enthalpy_unit]) - (float(self.gibbs_free_energy_input.text()
    ) * self.general_energy_conversions[
                free_energy_unit])) / (
                                    float(self.entropy_change_input.text()) *
    self.general_energy_conversions[
                                entropy_unit])
            temperature = temperature + self.temperature_conversions[
    temp_unit]
            return temperature
        except ValueError:
            print("Value Error")
            show_dialog("Value Error!")
            return

    def calculate_entropy_change(self, free_energy_unit, enthalpy_unit,
    temp_unit, entropy_unit):
        try:
            entropy_change = ((float(self.enthalpy_change_input.text()) *
    self.general_energy_conversions[
                enthalpy_unit]) - (float(self.gibbs_free_energy_input.text()
    ) * self.general_energy_conversions[
                free_energy_unit])) / (
                                       float(self.temp_input.text()) + self.
    temperature_conversions[temp_unit])
            entropy_change = entropy_change / self.
    general_energy_conversions[entropy_unit]
            return entropy_change
        except ValueError:
            print("Value Error")
            show_dialog("Value Error!")
            return


class SpecificHeatCalculator(QWidget):
    def __init__(self):
        super(QWidget, self).__init__()

        self.layout = QGridLayout()

        self.mass_conversions = {
            "mg": 0.001,
            "g": 1,
            "kg": 1000,
            "t": 1000000
```

```
        }

        self.temperature_conversions = {
            "C": 273.15,
            "K": 0.0,
        }

        self.energy_conversions = {
            "kJ": 0.001,
            "J": 1.0
        }

        self.energy_label = QLabel("Heat Energy: ")
        self.mass_label = QLabel("Mass: ")
        self.heat_capacity_label = QLabel("Specific Heat Capacity: ")
        self.temperature_change_label = QLabel("Temperature Change: ")

        self.energy_input = QLineEdit()
        self.mass_input = QLineEdit()
        self.heat_capacity_input = QLineEdit()
        self.temperature_change_input = QLineEdit()

        self.input_list = [self.energy_input, self.mass_input, self.
    heat_capacity_input, self.temperature_change_input]

        self.energy_unit_dropdown = QComboBox()
        self.energy_unit_dropdown.addItem("J")
        self.energy_unit_dropdown.addItem("kJ")

        self.energy_unit_dropdown.setCurrentIndex(0)

        self.mass_unit_dropdown = QComboBox()
        self.mass_unit_dropdown.addItem("mg")
        self.mass_unit_dropdown.addItem("g")
        self.mass_unit_dropdown.addItem("kg")

        self.mass_unit_dropdown.setCurrentIndex(1)

        self.temp_unit_dropdown = QComboBox()
        self.temp_unit_dropdown.addItem("C")
        self.temp_unit_dropdown.addItem("K")

        self.temp_unit_dropdown.setCurrentIndex(1)

        self.calculate_button = QPushButton("Calculate")

        self.calculate_button.clicked.connect(self.calculate)

        self.layout.addWidget(self.energy_label, 0, 0)
        self.layout.addWidget(self.energy_input, 0, 1)
        self.layout.addWidget(self.energy_unit_dropdown, 0, 2)

        self.layout.addWidget(self.mass_label, 1, 0)
        self.layout.addWidget(self.mass_input, 1, 1)
        self.layout.addWidget(self.mass_unit_dropdown, 1, 2)

        self.layout.addWidget(self.heat_capacity_label, 2, 0)
        self.layout.addWidget(self.heat_capacity_input, 2, 1)

```

```
1221            self.layout.addWidget(self.temperature_change_label, 3, 0)
1222            self.layout.addWidget(self.temperature_change_input, 3, 1)
1223            self.layout.addWidget(self.temp_unit_dropdown, 3, 2)
1224
1225            self.layout.addWidget(self.calculate_button, 4, 1)
1226
1227        def update_energy(self, energy):
1228            self.energy_input.setText(str(energy))
1229
1230        def update_mass(self, mass):
1231            self.mass_input.setText(str(mass))
1232
1233        def update_heat_capacity(self, hc):
1234            self.heat_capacity_input.setText(str(hc))
1235
1236        def update_temp_change(self, temp):
1237            self.temperature_change_input.setText(str(temp))
1238
1239        def calculate(self):
1240            energy_unit = self.energy_conversions[self.energy_unit_dropdown.
    currentText()]
1241            mass_unit = self.mass_conversions[self.mass_unit_dropdown.
    currentText()]
1242            temp_unit = self.temperature_conversions[self.temp_unit_dropdown.
    currentText()]
1243
1244            if not find_empty_input(self.input_list.copy()):
1245                show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
1246                return
1247            elif check_invalid_symbol(self.input_list.copy()):
1248                show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
1249                return
1250
1251            to_calc = find_empty_input(self.input_list.copy())
1252
1253            if to_calc is self.energy_input:
1254                energy = self.__calculate_energy(energy_unit, mass_unit,
    temp_unit)
1255                if energy:
1256                    self.update_energy(energy)
1257            if to_calc is self.mass_input:
1258                mass = self.__calculate_mass(energy_unit, mass_unit, temp_unit)
1259                if mass:
1260                    self.update_mass(mass)
1261            if to_calc is self.heat_capacity_input:
1262                heat_capacity = self.__calculate_heat_capacity(energy_unit,
    mass_unit, temp_unit)
1263                if heat_capacity:
1264                    self.update_heat_capacity(heat_capacity)
1265            if to_calc is self.temperature_change_input:
1266                temp = self.__calculate_temp_change(energy_unit, mass_unit,
    temp_unit)
1267                if temp:
1268                    self.update_temp_change(temp)
1269
1270        def __calculate_energy(self, energy_unit, mass_unit, temp_unit):
1271            try:
```

```
            q = (float(self.mass_input.text()) * mass_unit) * float(self.
    heat_capacity_input.text()) * (
                    float(self.temperature_change_input.text()) + temp_unit)
            q = q * energy_unit

            return q
        except ValueError:
            print("ValueError")
            return

    def __calculate_mass(self, energy_unit, mass_unit, temp_unit):
        try:
            mass = (float(self.energy_input.text()) * energy_unit) / (float(
    self.heat_capacity_input.text()) * (
                    float(self.temperature_change_input.text()) + temp_unit)
    )
            mass = mass * mass_unit

            return mass
        except ValueError:
            print("ValueError")
            return

    def __calculate_heat_capacity(self, energy_unit, mass_unit, temp_unit):
        try:
            heat_capacity = (float(self.energy_input.text()) * energy_unit)
    / (
                    (float(self.mass_input.text()) * mass_unit) * (
                    float(self.temperature_change_input.text()) + temp_unit)
    )

            return heat_capacity
        except ValueError:
            print("ValueError")
            return

    def __calculate_temp_change(self, energy_unit, mass_unit, temp_unit):
        try:
            temp_change = (float(self.energy_input.text()) * energy_unit) /
    (
                    (float(self.mass_input.text()) * mass_unit) * (
                float(self.heat_capacity_input.text())))
            temp_change = temp_change + temp_unit

            return temp_change
        except ValueError:
            print("ValueError")
            return


class RateCalculator(QWidget):
    def __init__(self):
        super(QWidget, self).__init__()

        self.order_conversion = {
            "First": 1,
            "Second": 2
        }

```

```
1325            self.layout = QVBoxLayout()
1326
1327            self.calculate_button = QPushButton("Calculate")
1328            self.calculate_button.clicked.connect(self.calculate)
1329
1330            self.step_dropdown = QComboBox()
1331            self.step_dropdown.addItem("Unimolecular")
1332            self.step_dropdown.addItem("Bimolecular")
1333            self.step_dropdown.addItem("Trimolecular")
1334            self.step_dropdown.currentIndexChanged.connect(self.get_ui)
1335
1336            self.layout.addWidget(self.step_dropdown)
1337
1338            # Unimolecular Box
1339            self.unimol_box = RateBox("[A]")
1340            self.unimol_box_widget = QWidget()
1341            self.unimol_box_widget.setLayout(self.unimol_box.box_layout)
1342
1343            self.unimol_conc_input = self.unimol_box.conc_input
1344            self.unimol_order_input = self.unimol_box.order_input
1345
1346            self.unimol_order_input.currentTextChanged.connect(self.
    get_total_order)
1347
1348            # Bimolecular Box
1349            self.bimol_box = RateBox("[B]")
1350            self.bimol_box_widget = QWidget()
1351            self.bimol_box_widget.setLayout(self.bimol_box.box_layout)
1352
1353            self.bimol_conc_input = self.bimol_box.conc_input
1354            self.bimol_order_input = self.bimol_box.order_input
1355
1356            self.bimol_order_input.currentTextChanged.connect(self.
    get_total_order)
1357
1358            # Trimolecular Box
1359            self.trimol_box = RateBox("[C]")
1360            self.trimol_box_widget = QWidget()
1361            self.trimol_box_widget.setLayout(self.trimol_box.box_layout)
1362
1363            self.trimol_conc_input = self.trimol_box.conc_input
1364            self.trimol_order_input = self.trimol_box.order_input
1365
1366            self.trimol_order_input.currentTextChanged.connect(self.
    get_total_order)
1367
1368            # Result Box
1369            self.result_box = RateResultBox()
1370            self.result_box_widget = QWidget()
1371            self.result_box_widget.setLayout(self.result_box.box_layout)
1372
1373            self.rate_constant_input = self.result_box.rate_constant_input
1374            self.rate_input = self.result_box.rate_input
1375            self.total_order_input = self.result_box.total_order_input
1376            self.total_order_input.setReadOnly(True)
1377
1378            # List containing every line edit currently visible on screen
1379            self.input_list = [self.unimol_conc_input, self.rate_constant_input,
     self.rate_input]
```

```
1380        self.conc_input_list = [self.unimol_conc_input, self.
    bimol_conc_input, self.trimol_conc_input]
1381
1382        self.get_ui()
1383
1384    def get_ui(self) -> None:
1385        """
1386        This method adds the necessary widgets to the current layout and
    makes sure that widgets are removed or added
1387        when needed.
1388        """
1389
1390        self.update_input_list()
1391        self.get_total_order()
1392
1393        self.layout.addWidget(self.unimol_box_widget)
1394        self.layout.addWidget(self.bimol_box_widget)
1395        self.layout.addWidget(self.trimol_box_widget)
1396        self.layout.addWidget(self.result_box_widget)
1397        self.layout.addWidget(self.calculate_button)
1398
1399        # Hide the widgets to ensure correct order of display
1400        self.bimol_box_widget.hide()
1401        self.trimol_box_widget.hide()
1402        self.result_box_widget.hide()
1403        self.calculate_button.hide()
1404
1405        if self.step_dropdown.currentIndex() == 1:
1406            self.bimol_box_widget.show()
1407            self.result_box_widget.show()
1408            self.calculate_button.show()
1409        elif self.step_dropdown.currentIndex() == 2:
1410            self.bimol_box_widget.show()
1411            self.trimol_box_widget.show()
1412            self.result_box_widget.show()
1413            self.calculate_button.show()
1414        else:
1415            self.result_box_widget.show()
1416            self.calculate_button.show()
1417
1418    def update_input_list(self) -> None:
1419        """
1420        This function makes sure the self.input_list is always xup-to-date
    with the inputs displayed in the gui.
1421        """
1422
1423        if self.bimol_conc_input in self.input_list.copy():
1424            self.input_list.remove(self.bimol_conc_input)
1425        if self.trimol_conc_input in self.input_list.copy():
1426            self.input_list.remove(self.trimol_conc_input)
1427
1428        if self.step_dropdown.currentIndex() == 1:
1429            self.input_list.append(self.bimol_conc_input)
1430        elif self.step_dropdown.currentIndex() == 2:
1431            self.input_list.append(self.bimol_conc_input)
1432            self.input_list.append(self.trimol_conc_input)
1433
1434    def calculate(self) -> None:
1435        """
```

```
1436        This method checks which input has been left empty, and then calls
    the according method to calculate the
1437        missing value.
1438        """
1439
1440        if not find_empty_input(self.input_list.copy()):
1441            show_dialog("Must leave exactly one input line empty for it to
    be calculated!")
1442            return
1443        elif check_invalid_symbol(self.input_list.copy()):
1444            show_dialog("Only numerical values in the form of integers or
    decimals allowed!")
1445            return
1446
1447        unimol_order = self.order_conversion[self.unimol_order_input.
    currentText()]
1448        bimol_order = self.order_conversion[self.bimol_order_input.
    currentText()]
1449        trimol_order = self.order_conversion[self.trimol_order_input.
    currentText()]
1450
1451        to_find = find_empty_input(self.input_list)
1452
1453        if to_find is self.rate_input:
1454            self.calculate_rate(unimol_order, bimol_order, trimol_order)
1455        elif to_find is self.rate_constant_input:
1456            self.calculate_rate_constant(unimol_order, bimol_order,
    trimol_order)
1457        elif to_find in self.conc_input_list:
1458            self.calculate_concentration(to_find, unimol_order, bimol_order,
     trimol_order)
1459
1460    def get_total_order(self) -> None:
1461        """
1462        This method makes sure the total order is always displayed correctly
    .
1463        """
1464
1465        unimol_order = self.order_conversion[self.unimol_order_input.
    currentText()]
1466        bimol_order = self.order_conversion[self.bimol_order_input.
    currentText()]
1467        trimol_order = self.order_conversion[self.trimol_order_input.
    currentText()]
1468
1469        total_order = unimol_order
1470        if self.step_dropdown.currentIndex() == 1:
1471            total_order += bimol_order
1472        elif self.step_dropdown.currentIndex() == 2:
1473            total_order += bimol_order + trimol_order
1474
1475        self.update_total_order(total_order)
1476
1477    def calculate_rate(self, unimol_order: int, bimol_order: int,
    trimol_order: int) -> None:
1478        """
1479        Calculates the rate of the reaction based on the elementary step
1480        (unimolecular, bimolecular, trimolecular), and then calls the
    update_input() method to display the calculated
```

```
1481            result.
1482            """
1483
1484            rate = float(self.rate_constant_input.text()) * (float(self.
    unimol_conc_input.text()) ** unimol_order)
1485
1486            if self.step_dropdown.currentIndex() == 1:
1487                rate = rate * (float(self.bimol_conc_input.text()) **
    bimol_order)
1488            elif self.step_dropdown.currentIndex() == 2:
1489                rate = rate * (float(self.bimol_conc_input.text()) **
    bimol_order) * (
1490                        float(self.trimol_conc_input.text()) ** trimol_order)
1491
1492            self.update_input(rate)
1493
1494        def calculate_rate_constant(self, unimol_order: int, bimol_order: int,
    trimol_order: int) -> None:
1495            """
1496            Calculates the rate constant based on the elementary step
1497            (unimolecular, bimolecular, trimolecular), and calls the
    update_input method to display the calculated result.
1498            """
1499
1500            total_conc = (float(self.unimol_conc_input) ** unimol_order)
1501
1502            if self.step_dropdown.currentIndex() == 1:
1503                total_conc = total_conc * (float(self.bimol_conc_input.text())
    ** bimol_order)
1504            elif self.step_dropdown.currentIndex() == 2:
1505                total_conc = total_conc * (float(self.bimol_conc_input.text())
    ** bimol_order) * (
1506                        float(self.trimol_conc_input.text()) ** trimol_order)
1507
1508            rate_constant = float(self.rate_input) / total_conc
1509
1510            self.update_input(rate_constant)
1511
1512        def calculate_concentration(self, to_find: QLineEdit, unimol_order: int,
     bimol_order: int,
1513                                    trimol_order: int) -> None:
1514            """
1515            Calculates the concentration of the given empty concentration input,
     based on the elementary step
1516            (unimolecular, bimolecular, trimolecular), and then calls the
    update_input() method to display the calculated
1517            result.
1518            """
1519
1520            concentration = float(self.rate_input.text()) / float(self.
    rate_constant_input.text())
1521
1522            if to_find is self.unimol_conc_input:
1523                if self.step_dropdown.currentIndex() == 1:
1524                    concentration = concentration / (float(self.bimol_conc_input
    .text()) ** bimol_order)
1525                elif self.step_dropdown.currentIndex() == 2:
1526                    concentration = concentration / ((float(self.
    bimol_conc_input.text()) ** bimol_order) * (
```

```
1527                        float(self.trimol_conc_input.text()) ** trimol_order
    ))
1528            concentration = concentration ** (1 / unimol_order)
1529        elif to_find is self.bimol_conc_input:
1530            if self.step_dropdown.currentIndex() == 1:
1531                concentration = concentration / (float(self.
    unimol_conc_input.text()) ** unimol_order)
1532            elif self.step_dropdown.currentIndex() == 2:
1533                concentration = concentration / ((float(self.
    unimol_conc_input.text()) ** unimol_order) * (
1534                        float(self.trimol_conc_input.text()) ** trimol_order
    ))
1535            concentration = concentration ** (1 / bimol_order)
1536        elif to_find is self.trimol_conc_input:
1537            concentration = concentration / (
1538                    (float(self.unimol_conc_input.text()) ** unimol_order) *
     (
1539                    float(self.bimol_conc_input.text()) ** bimol_order))
1540            concentration = concentration ** (1 / trimol_order)
1541        else:
1542            return
1543        self.update_input(concentration)
1544
1545    def update_input(self, result: float) -> None:
1546        """
1547        Uses the find_empty_input() function to find the empty input, and
    then updates it using the result parameter.
1548        """
1549
1550        find_empty_input(self.input_list.copy()).setText(str(result))
1551
1552    def update_total_order(self, order: float) -> None:
1553        """
1554        Updates the total order input with the order parameter.
1555        """
1556
1557        self.total_order_input.setText(str(order))
```

Listing 9: chem_calculator.py File Program Code

A.4 ChemBalancer File

```
from PyQt6.QtWidgets import QWidget, QPushButton, QLineEdit, QLabel,
    QVBoxLayout, QFrame, QHBoxLayout, QMessageBox
from PyQt6.QtCore import Qt

import re
from sympy import Matrix, lcm


def show_dialog(message):
    dlg = QMessageBox()
    dlg.setWindowTitle("Invalid Input!")
    dlg.setText(f"Invalid user input!\n {message}")
    dlg.setIcon(QMessageBox.Icon.Critical)
    button = dlg.exec()

    if button == QMessageBox.StandardButton.Ok:
        print("OK!")


class ChemBalancer(QWidget):
    """
    This module is responsible for balancing chemical equations.
    It parses user-provided chemical equations, identifies reactants and
    products,
    and calculates the coefficients to achieve a balanced equation.
    The class utilizes SymPy for symbolic mathematics to find the null space
     and perform matrix operations,
    ensuring accurate and balanced chemical equations.
    """

    def __init__(self):
        super(QWidget, self).__init__()
        self.balancer_layout = QVBoxLayout()

        self.unbalanced_frame = QFrame()
        self.unbalanced_frame.setFrameShape(QFrame.Shape.Panel)

        self.balanced_frame = QFrame()
        self.balanced_frame.setFrameShape(QFrame.Shape.Panel)

        self.unbalanced_label = QLabel("Unbalanced Equation:")
        self.equation_input = QLineEdit()
        self.balanced_label = QLabel("Balanced Equation:")
        self.balanced_output = QLineEdit()
        self.balanced_output.setReadOnly(True)
        self.balance_button = QPushButton("Balance")

        self.equation_input.setMinimumWidth(800)
        self.balanced_output.setMinimumWidth(800)
        self.balance_button.setFixedWidth(300)

        self.unbalanced_frame_layout = QHBoxLayout()
        self.unbalanced_frame.setLayout(self.unbalanced_frame_layout)
        self.unbalanced_frame_layout.addWidget(self.unbalanced_label)
        self.unbalanced_frame_layout.addWidget(self.equation_input)

```

```
54          self.balanced_frame_layout = QHBoxLayout()
55          self.balanced_frame.setLayout(self.balanced_frame_layout)
56          self.balanced_frame_layout.addWidget(self.balanced_label)
57          self.balanced_frame_layout.addWidget(self.balanced_output)
58
59          # Add the widgets to the balancerLayout
60          self.balancer_layout.addWidget(self.unbalanced_frame, alignment=Qt.
    AlignmentFlag.AlignCenter)
61          self.balancer_layout.addWidget(self.balanced_frame, alignment=Qt.
    AlignmentFlag.AlignCenter)
62          self.balancer_layout.addWidget(self.balance_button, alignment=Qt.
    AlignmentFlag.AlignCenter)
63
64          self.balance_button.clicked.connect(self.run_balancer)
65
66          self.stripped_equation = str()
67          self.equation_split = list()
68
69          self.reactants = list()
70          self.products = list()
71
72          self.element_list = list()
73          self.element_matrix = list()
74
75          self.balanced_equation = str()
76
77      def clear_variables(self):
78          """
79          Clears all the used variables to avoid using data or values from
    previous calculations.
80          """
81
82          if len(self.equation_split) != 0:
83              self.equation_split.clear()
84          if len(self.reactants) != 0:
85              self.reactants.clear()
86          if len(self.products) != 0:
87              self.products.clear()
88          if len(self.element_list) != 0:
89              self.element_list.clear()
90          if len(self.element_matrix) != 0:
91              self.element_matrix = []
92
93          self.reactants = ""
94          self.products = ""
95          self.balanced_equation = ""
96
97      def split_equation(self):
98          f"""
99          Takes {self.equation_input}, strips it from all the whitespaces
100         and splits it up into separate reactants and products.
101         """
102
103         # Strip equation from any whitespaces
104         try:
105             self.stripped_equation = "".join(self.equation_input.text().
    split())
106         except IndexError:
107             return None
```

```
108         print(self.stripped_equation)
109
110         # Split equation into reactants (self.equationSplit[0]) and products
     (self.equationSplit[1])
111         try:
112             self.equation_split = self.stripped_equation.split("=")
113         except Exception:
114             raise
115
116         print(self.equation_split)
117
118         self.reactants = self.equation_split[0].split("+")
119
120         print(self.reactants)
121
122         self.products = self.equation_split[1].split("+")
123
124         print(self.products)
125
126     def find_reagents(self, compound, index, side):
127         f"""
128         This Function finds separate reagents by removing brackets from the
    compounds
129         and then calls {self.find_elements}.
130
131         :param compound: String of elements as compound (e.g. Ag3(Fe30)4).
132         :param index: Index position of row in matrix.
133         :param side: "1" for reactants, "-1" for products.
134         """
135
136         print("compound", compound)
137         # Split the compound by parentheses
138         reagents = re.split("(\([A-Za-z0-9]*\)[0-9]*)", compound)
139         print("Reagents", reagents)
140         for reagent in reagents:
141             if reagent.startswith("("):
142                 # Extract the element within parentheses
143                 inner_compound = reagent[1:-1]
144                 # Get the subscript outside the brackets
145                 bracket_subscript = reagent.split(")", 1)[-1]
146                 if bracket_subscript:
147                     bracket_subscript = int(bracket_subscript)
148                 else:
149                     bracket_subscript = 1
150                 # Recursively find elements within the inner compound
151                 try:
152                     self.find_elements(inner_compound, index,
    bracket_subscript, side)
153                 except Exception:
154                     raise
155             else:
156                 # No brackets, directly find elements
157                 bracket_subscript = 1
158                 try:
159                     self.find_elements(reagent, index, bracket_subscript,
    side)
160                 except Exception:
161                     raise
162
```

```
163     def find_elements(self, reagent, index, bracket_subscript, side):
164         f"""
165         Separates out elements and subscripts using a regex,
166         then loops through the elements and calls {self.add_to_matrix}.
167 
168         :param reagent: String of reagent (e.g. H2O).
169         :param index: Index position of row in matrix.
170         :param bracket_subscript: The subscript value outside the brackets.
    Equal to 1 if there are no brackets.
171         :param side: "1" for reactants, "-1" for products.
172         """
173 
174         print("reagent in find_el", reagent)
175         # Use regex to separate elements and subscripts
176         element_counts = re.findall("([A-Z][a-z]*)([0-9]*)", reagent)
177         print("element counts", element_counts)
178         try:
179             for element, subscript in element_counts:
180                 if not subscript:
181                     subscript = 1
182                 else:
183                     subscript = int(subscript)
184                 # Call addToMatrix for each element
185                 try:
186                     self.add_to_matrix(element, index, bracket_subscript *
    subscript, side)
187                 except Exception:
188                     raise
189         except Exception:
190             raise
191 
192     def add_to_matrix(self, element, index, count, side):
193         """
194         This function adds the provided element with a specified count to
    the matrix at the given index.
195         The 'side' parameter determines whether the element is part of the
    reactants (positive side)
196         or products (negative side) in the chemical equation.
197 
198         :param element: The element symbol as in the periodic table (e.g. Na
    ).
199         :param index: Index position of row in matrix.
200         :param count: Number of specific element to add to the matrix.
201         :param side: "1" for reactants, "-1" for products.
202         """
203 
204         print(element, index, count, side)
205         try:
206             if index == len(self.element_matrix):
207                 print(self.element_matrix)
208                 self.element_matrix.append([])
209                 print(self.element_matrix)
210                 for x in self.element_list:
211                     print(self.element_list)
212                     self.element_matrix[index].append(0)
213                     print(self.element_matrix)
214         except Exception:
215             raise
216         try:
```

```
            if element not in self.element_list:
                self.element_list.append(element)
                for i in range(len(self.element_matrix)):
                    self.element_matrix[i].append(0)
                    print(self.element_matrix)
        except Exception:
            raise

        column = self.element_list.index(element)
        self.element_matrix[index][column] += count * side
        print(self.element_list)
        print(self.element_matrix)

    def run_balancer(self):
        f"""
        This the core function of the {ChemBalancer} class,
        responsible for balancing chemical equations.
        It parses the user-provided equation, deciphers compounds,
        constructs a matrix, finds the null space for balancing coefficients
    ,
        and computes the balanced equation.
        This function leverages SymPy for mathematical operations, ensuring
    accurate chemical equation balancing.
        """

        # Clear variables, in case the program was run before
        self.clear_variables()

        try:
            self.split_equation()
        except Exception:
            show_dialog("")
            return

        for i in range(len(self.reactants)):
            try:
                self.find_reagents(self.reactants[i], i, 1)
            except Exception:
                show_dialog("")
                return
        for i in range(len(self.products)):
            try:
                self.find_reagents(self.products[i], i + len(self.reactants)
    , -1)
            except Exception:
                show_dialog("")
                return

        self.element_matrix = Matrix(self.element_matrix)
        self.element_matrix = self.element_matrix.transpose()

        try:
            num = self.element_matrix.nullspace()[0]
        except IndexError:
            show_dialog("Please provide a balanceable equation in the format
     X + Y = XY + Z")
            return

        print(num)
```

```
272         multiple = lcm([val.q for val in num])
273         num = multiple * num
274         print(num)
275 
276         coefficient = num.tolist()
277 
278         for i in range(len(self.reactants)):
279             if coefficient[i][0] != 1:
280                 self.balanced_equation += str(coefficient[i][0]) + self.
    reactants[i]
281             else:
282                 self.balanced_equation += self.reactants[i]
283             if i < len(self.reactants) - 1:
284                 self.balanced_equation += " + "
285         self.balanced_equation += " = "
286 
287         for i in range(len(self.products)):
288             if coefficient[i + len(self.reactants)][0] != 1:
289                 self.balanced_equation += str(coefficient[i + len(self.
    reactants)][0]) + self.products[i]
290             else:
291                 self.balanced_equation += self.products[i]
292             if i < len(self.products) - 1:
293                 self.balanced_equation += " + "
294         self.balanced_output.setText(f"{self.balanced_equation}")
```

Listing 10: chem_balancer.py File Program Code

A.5 ChemEditor GUI file

```
from PyQt6.QtCore import Qt, QPointF, pyqtSignal
from PyQt6.QtCore import QPoint
from PyQt6.QtWidgets import QWidget, QGridLayout, QPushButton, QLabel,
    QFileDialog, QMessageBox
from PyQt6.QtGui import QPixmap, QPainter, QPen, QColor, QFont, QBrush

import chem_editor_logic

import math
import json


def show_dialog(message):
    dlg = QMessageBox()
    dlg.setWindowTitle("Invalid Action!")
    dlg.setText(f"Invalid user action!\n {message}")
    dlg.setIcon(QMessageBox.Icon.Critical)
    button = dlg.exec()

    if button == QMessageBox.StandardButton.Ok:
        print("OK!")


class ChemEditor(QWidget):
    def __init__(self):
        super().__init__()

        self.editor_layout = QGridLayout()
        self.setLayout(self.editor_layout)

        self.draw_action_button = QPushButton("Draw")
        self.bond_action_button = QPushButton("Bond")

        self.remove_button = QPushButton("Remove")
        self.reset_button = QPushButton("Reset")

        self.save_button = QPushButton("Save")

        self.single_bond_button = QPushButton("Single")
        self.double_bond_button = QPushButton("Double")
        self.triple_bond_button = QPushButton("Triple")

        self.editor_layout.addWidget(self.draw_action_button, 0, 10)
        self.editor_layout.addWidget(self.bond_action_button, 0, 11)
        self.editor_layout.addWidget(self.remove_button, 0, 12, 1, 2)
        self.editor_layout.addWidget(self.reset_button, 0, 14)
        self.editor_layout.addWidget(self.save_button, 0, 15)
        self.editor_layout.addWidget(self.single_bond_button, 0, 22)
        self.editor_layout.addWidget(self.double_bond_button, 0, 23)
        self.editor_layout.addWidget(self.triple_bond_button, 0, 24)

        self.bond_action_button.clicked.connect(self.choose_bond_action)
        self.draw_action_button.clicked.connect(self.choose_draw_action)
        self.remove_button.clicked.connect(self.remove_action)
        self.reset_button.clicked.connect(self.reset_action)
        self.save_button.clicked.connect(self.save_action)
```

```
56          self.single_bond_button.clicked.connect(self.choose_first_order)
57          self.double_bond_button.clicked.connect(self.choose_second_order)
58          self.triple_bond_button.clicked.connect(self.choose_third_order)
59
60          self.canvas = Canvas()
61          self.editor_layout.addWidget(self.canvas, 1, 0, 25, 25)
62
63          self.periodic_table = PeriodicTable()
64
65          self.periodic_table.element_clicked.connect(self.set_element)
66
67          self.periodic_table_btn = QPushButton("Elements")
68          self.periodic_table_btn.clicked.connect(self.show_periodic_table)
69
70          self.editor_layout.addWidget(self.periodic_table_btn, 0, 0, 1, 2)
71
72          self.chem_logic = chem_editor_logic
73
74      def show_periodic_table(self):
75          self.periodic_table.show()
76
77      def set_element(self, data) -> None:
78          self.canvas.set_element(data)
79
80      def choose_draw_action(self) -> None:
81          self.canvas.set_action_type("draw")
82
83      def choose_bond_action(self) -> None:
84          self.canvas.set_action_type("bond")
85
86      def remove_action(self) -> None:
87          self.canvas.set_action_type("remove")
88
89      def reset_action(self) -> None:
90          self.canvas.reset_canvas()
91
92      def save_action(self) -> None:
93          self.canvas.save()
94
95      def choose_first_order(self) -> None:
96          self.canvas.set_bond_order(1)
97
98      def choose_second_order(self) -> None:
99          self.canvas.set_bond_order(2)
100
101     def choose_third_order(self) -> None:
102         self.canvas.set_bond_order(3)
103
104
105 class Canvas(QLabel):
106     # CONSTANTS
107     ATOM_RADIUS = 12
108
109     def __init__(self):
110         super().__init__()
111
112         self.pixmap = QPixmap(1280, 720)
113         self.pixmap.fill(QColor(200, 200, 200))
114         self.setPixmap(self.pixmap)
```

```
115
116         self.chem_logic = chem_editor_logic
117
118         # Set default element to carbon
119         self.element: dict = self.get_carbon()
120
121         # List containing all atoms on the Canvas
122         self.atoms: list[chem_editor_logic.Atom] = []
123
124         # Temporary list of atoms for bonding
125         self.temp_bond_list: list[chem_editor_logic.Atom] = []
126
127         # Set default action type to draw
128         self.action_type: str = "draw"
129
130         # Set default bond order to 1
131         # 1: single bond
132         # 2: double bond
133         # 3: triple bond
134         self.bond_order: int = 1
135
136         # Initially no atom is selected
137         self.selected: bool = False
138
139         # Initially no atom is selected
140         self.selected_atom = None
141
142     def get_carbon(self) -> dict:
143         elements = json.load(open("elements.json"))
144
145         for element, data in elements.items():
146             if element == "Carbon":
147                 return data
148
149     def save(self) -> None:
150         # Select file path
151         filePath, _ = QFileDialog.getSaveFileName(self, "Save Image", "",
152                                                   "PNG(*.png);;JPEG(*.jpg *.
    jpeg);;All Files(*.*) ")
153
154         # If file path is blank return back
155         if filePath == "":
156             return
157
158         # Save canvas at desired path
159         self.pixmap.save(filePath)
160
161     def set_element(self, new_element: dict) -> None:
162         self.element = new_element
163         self.update()
164
165     def set_action_type(self, action: str) -> None:
166         self.action_type = action
167         self.update()
168
169     def set_bond_order(self, order: int) -> None:
170         self.bond_order = order
171         self.update()
172
```

```
173     def remove(self) -> None:
174         if self.selected:
175             self.atoms.remove(self.selected_atom)
176         self.update()
177 
178     def reset_canvas(self) -> None:
179         self.atoms.clear()
180         self.temp_bond_list.clear()
181         self.set_action_type("draw")
182         self.set_bond_order(1)
183 
184     def paintEvent(self, event) -> None:
185 
186         self.pixmap.fill(QColor(200, 200, 200))
187         # Initialise painter
188         init_painter = QPainter(self)
189         init_painter.drawPixmap(0, 0, self.pixmap)
190 
191         # Initialise pixmap painter
192         pix_painter = QPainter(self.pixmap)
193         font = QFont("Arial", 16)
194         pix_painter.setFont(font)
195 
196         pen = QPen(QColor(0, 0, 0))
197         pen.setWidth(2)
198         pix_painter.setPen(pen)
199 
200         # Draw every atom in self.atoms list
201         for atom in self.atoms:
202             self.draw_atom(atom.x_coords, atom.y_coords, atom.symbol,
    pix_painter, pen, False)
203 
204             # For every bond of atom, draw the bond, and redraw the atoms
    again
205             for bond in atom.bonds:
206                 if bond.order == 2:
207                     self.draw_double_bond(bond.atoms[0].x_coords, bond.atoms
    [0].y_coords, bond.atoms[1].x_coords,
208                                           bond.atoms[1].y_coords,
    pix_painter, pen, True)
209                 elif bond.order == 3:
210                     self.draw_triple_bond(bond.atoms[0].x_coords, bond.atoms
    [0].y_coords, bond.atoms[1].x_coords,
211                                           bond.atoms[1].y_coords,
    pix_painter, pen, True)
212                 else:
213                     self.draw_single_bond(bond.atoms[0].x_coords, bond.atoms
    [0].y_coords, bond.atoms[1].x_coords,
214                                           bond.atoms[1].y_coords,
    pix_painter, pen, True)
215 
216                 self.draw_atom_circle(bond.atoms[1].x_coords, bond.atoms[1].
    y_coords, bond.atoms[0].x_coords,
217                                       bond.atoms[0].y_coords, pix_painter,
    pen)
218                 pen.setStyle(Qt.PenStyle.SolidLine)
219                 pix_painter.setPen(pen)
220                 self.draw_atom(bond.atoms[1].x_coords, bond.atoms[1].
    y_coords, bond.atoms[1].symbol, pix_painter, pen,
```

```
                                False)
                self.draw_atom(bond.atoms[0].x_coords, bond.atoms[0].
    y_coords, bond.atoms[0].symbol, pix_painter, pen,
                                False)

        # Check for selected atom and draw potential positions
        if self.selected:
            print("selected")
            if self.action_type != "bond":
                try:
                    # Calculate possible positions for new atoms in 360
    degrees around the selected atom
                    if self.selected_atom is not None:
                        potential_positions = self.calc_potential_positions(
    self.selected_atom)
                        for pos in potential_positions:
                            # If atoms at position don't overlap, draw the
    potential bonds and atoms in different colour
                            if not self.check_atom_overlap(pos[0], pos[1]):
                                if self.bond_order == 2:
                                    self.draw_double_bond(self.selected_atom
    .x_coords, self.selected_atom.y_coords,
                                                          pos[0],
                                                          pos[1],
    pix_painter, pen, False)
                                elif self.bond_order == 3:
                                    self.draw_triple_bond(self.selected_atom
    .x_coords, self.selected_atom.y_coords,
                                                          pos[0], pos[1],
    pix_painter, pen, False)
                                else:
                                    self.draw_single_bond(self.selected_atom
    .x_coords, self.selected_atom.y_coords,
                                                          pos[0], pos[1],
    pix_painter, pen, False)
                                self.draw_atom_circle(pos[0], pos[1], self.
    selected_atom.x_coords,
                                                      self.selected_atom.
    y_coords, pix_painter, pen)
                                pen.setStyle(Qt.PenStyle.SolidLine)
                                pix_painter.setPen(pen)

                                # Use of self.element["symbol"], as the
    whole elements json data is stored in self.element
                                # E.g. {'symbol': 'C',
                                # 'atomic_number': 6,
                                # 'group': 14,
                                # 'period': 2,
                                # 'outer_electrons': 4,
                                # 'full_shell': 8}

                                self.draw_atom(pos[0], pos[1], self.element[
    "symbol"], pix_painter, pen, True)
                                self.draw_atom(self.selected_atom.x_coords,
    self.selected_atom.y_coords,
                                               self.selected_atom.symbol,
    pix_painter, pen, False)
                        init_painter.drawPixmap(0, 0, self.pixmap)
                except AttributeError:
```

```
                    return
        init_painter.drawPixmap(0, 0, self.pixmap)
        init_painter.end()
        pix_painter.end()

    # Function to draw potential positions for atoms
    @staticmethod
    def calc_potential_positions(atom: chem_editor_logic.Atom) -> list[tuple[int, int]]:
        """

        :param atom:
        :return: list of tuples containing x and y coordinates of potential positions for new atoms
        """

        x = atom.x_coords
        y = atom.y_coords
        distance = 40

        # Calculate coordinates for angles in steps of 45 degrees from 0 to 360
        coordinates_list = []
        for angle_degrees in range(0, 360, 45):
            angle_radians = math.radians(angle_degrees)
            new_x = x + distance * math.cos(angle_radians)
            new_y = y + distance * math.sin(angle_radians)
            coordinates_list.append((int(new_x), int(new_y)))

        print(x, y)
        print(coordinates_list)
        return coordinates_list

    def check_atom_overlap(self, pos_x: int, pos_y: int) -> bool:
        atom_radius = Canvas.ATOM_RADIUS
        for atom in self.atoms:
            if (
                    atom.x_coords - atom_radius <= pos_x <= atom.x_coords + atom_radius and
                    atom.y_coords - atom_radius <= pos_y <= atom.y_coords + atom_radius
            ):
                return True

    def draw_single_bond(self, atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter: QPainter, pen: QPen,
                         actual_bond: bool = True) -> None:
        if not actual_bond:
            # Set colour red
            pen.setColor(QColor(255, 0, 0))
            painter.setPen(pen)

        painter.drawLine(QPoint(atom1_x, atom1_y), QPoint(atom2_x, atom2_y))

    def draw_double_bond(self, atom1_x: int, atom1_y: int, atom2_x: int, atom2_y: int, painter: QPainter, pen: QPen,
                         actual_bond: bool = True) -> None:

        if not actual_bond:
```

```
316             # Set colour red
317             pen.setColor(QColor(255, 0, 0))
318             painter.setPen(pen)
319 
320         offset = 2
321         diag_offset = 3
322 
323         self.__diagonal_bonds(atom1_x, atom1_y, atom2_x, atom2_y, painter,
    offset, diag_offset)
324 
325     def draw_triple_bond(self, atom1_x: int, atom1_y: int, atom2_x: int,
    atom2_y: int, painter: QPainter, pen: QPen,
326                          actual_bond: bool = True) -> None:
327 
328         if not actual_bond:
329             # Set colour red
330             pen.setColor(QColor(255, 0, 0))
331             painter.setPen(pen)
332 
333         offset = 4
334         diag_offset = 6
335 
336         painter.drawLine(QPoint(atom1_x, atom1_y),
337                          QPoint(atom2_x, atom2_y))
338         self.__diagonal_bonds(atom1_x, atom1_y, atom2_x, atom2_y, painter,
    offset, diag_offset)
339 
340     def __diagonal_bonds(self, atom1_x: int, atom1_y: int, atom2_x: int,
    atom2_y: int, painter: QPainter, offset: int,
341                          diag_offset: int) -> None:
342         # Top left diagonal
343         if atom2_x < atom1_x and atom2_y < atom1_y:
344             painter.drawLine(QPoint(atom1_x, atom1_y - diag_offset),
345                              QPoint(atom2_x + diag_offset, atom2_y))
346             painter.drawLine(QPoint(atom1_x - diag_offset, atom1_y),
347                              QPoint(atom2_x, atom2_y + diag_offset))
348         # Bottom left diagonal
349         elif atom2_x < atom1_x and atom2_y > atom1_y:
350             painter.drawLine(QPoint(atom1_x - diag_offset, atom1_y),
351                              QPoint(atom2_x, atom2_y - diag_offset))
352             painter.drawLine(QPoint(atom1_x, atom1_y + diag_offset),
353                              QPoint(atom2_x + diag_offset, atom2_y))
354         # Top right diagonal
355         elif atom2_x > atom1_x and atom2_y < atom1_y:
356             painter.drawLine(QPoint(atom1_x, atom1_y - diag_offset),
357                              QPoint(atom2_x - diag_offset, atom2_y))
358             painter.drawLine(QPoint(atom1_x + diag_offset, atom1_y),
359                              QPoint(atom2_x, atom2_y + diag_offset))
360         # Bottom right diagonal
361         elif atom2_x > atom1_x and atom2_y > atom1_y:
362             painter.drawLine(QPoint(atom1_x + diag_offset, atom1_y),
363                              QPoint(atom2_x, atom2_y - diag_offset))
364             painter.drawLine(QPoint(atom1_x, atom1_y + diag_offset),
365                              QPoint(atom2_x - diag_offset, atom2_y))
366         # Horizontal and Vertical lines
367         else:
368             painter.drawLine(QPoint(atom1_x - offset, atom1_y - offset),
369                              QPoint(atom2_x - offset, atom2_y - offset))
370             painter.drawLine(QPoint(atom1_x + offset, atom1_y + offset),
```

```
371                             QPoint(atom2_x + offset, atom2_y + offset))
372
373     def draw_atom_circle(self, atom1_x: int, atom1_y: int, atom2_x: int,
    atom2_y: int, painter: QPainter,
374                          pen: QPen) -> None:
375         """
376         This function draws a circle in the same colour as the background
    colour to prevent bonds from overlapping with
377         atom. This function must be called after drawing bonds in order to
    overwrite them, and before drawing atoms,
378         as this would make the atoms invisible.
379         """
380
381         # Set the brush color to match the background color
382         background_color = QColor(200, 200, 200)
383         brush = QBrush(background_color)
384         painter.setBrush(brush)
385         pen.setStyle(Qt.PenStyle.NoPen)
386         painter.setPen(pen)
387
388         # Draw a filled circle
389         circle_center = QPointF(atom2_x, atom2_y)
390         circle_radius = Canvas.ATOM_RADIUS
391         painter.drawEllipse(circle_center, circle_radius, circle_radius)
392
393         circle_center = QPointF(atom1_x, atom1_y)
394         painter.drawEllipse(circle_center, circle_radius, circle_radius)
395
396     def draw_atom(self, atom_x: int, atom_y: int, symbol: str, painter:
    QPainter, pen: QPen,
397                   potential: bool = False) -> None:
398
399         if potential:
400             pen.setColor(QColor(100, 100, 100))
401             painter.setPen(pen)
402
403         # Calculate position to draw atom in the center of the "atom circle"
404         letter_width = painter.fontMetrics().horizontalAdvance(symbol)
405         letter_height = painter.fontMetrics().height()
406         letter_x = atom_x - letter_width / 2
407         letter_y = atom_y + letter_height / 4
408         painter.drawText(int(letter_x), int(letter_y), symbol)
409
410     def check_clicked_on_atom(self, pos_x: int, pos_y: int) -> bool:
411         atom_radius = Canvas.ATOM_RADIUS
412         for atom in self.atoms:
413             if (
414                     atom.x_coords - atom_radius <= pos_x <= atom.x_coords +
    atom_radius and
415                     atom.y_coords - atom_radius <= pos_y <= atom.y_coords +
    atom_radius
416             ):
417                 self.selected = True
418                 self.selected_atom = atom
419                 self.update()
420                 return True
421
422     def remove_atom(self, pos_x: int, pos_y: int) -> None:
423         atom_radius = Canvas.ATOM_RADIUS
```

```
424
425         self.atoms = [atom for atom in self.atoms if not (
426                 atom.x_coords - atom_radius <= pos_x <= atom.x_coords +
    atom_radius and
427                 atom.y_coords - atom_radius <= pos_y <= atom.y_coords +
    atom_radius
428         )]
429         self.selected_atom = None
430         self.update()
431
432     def remove_bond(self, pos_x: int, pos_y: int) -> None:
433         atom_radius = Canvas.ATOM_RADIUS
434         for atom in self.atoms:
435             if (
436                     atom.x_coords - atom_radius <= pos_x <= atom.x_coords +
    atom_radius and
437                     atom.y_coords - atom_radius <= pos_y <= atom.y_coords +
    atom_radius
438             ):
439                 for bond in atom.bonds:
440                     if atom is bond.atoms[0]:
441                         atom.break_bond(bond.atoms[1], bond.order)
442                     elif atom is bond.atoms[1]:
443                         atom.break_bond(bond.atoms[0], bond.order)
444                 atom.bonds.clear()
445         self.update()
446
447     def mousePressEvent(self, event) -> None:
448         if event.button() == Qt.MouseButton.LeftButton:
449             click_position = event.pos()
450
451             if self.action_type == "remove":
452                 self.remove_bond(click_position.x(), click_position.y())
453                 self.remove_atom(click_position.x(), click_position.y())
454             elif self.action_type == "bond":
455                 self.selected_atom = None
456                 for atom in self.atoms:
457                     atom_x = atom.x_coords
458                     atom_y = atom.y_coords
459                     atom_radius = Canvas.ATOM_RADIUS
460
461                     if (
462                             atom_x - atom_radius <= click_position.x() <=
    atom_x + atom_radius and
463                             atom_y - atom_radius <= click_position.y() <=
    atom_y + atom_radius
464                     ):
465                         self.selected = True
466                         self.update()
467
468                         self.selected_atom = atom
469                         self.temp_bond_list.append(atom)
470                         if len(self.temp_bond_list) == 2:
471                             if self.temp_bond_list[0] is self.temp_bond_list
    [1]:
472                                 print("Trying to bond to itself")
473                                 show_dialog("You cannot bond an atom to
    itself!")
474                                 self.temp_bond_list.clear()
```

```
475                                 return
476                             self.temp_bond_list[0].bond(self.temp_bond_list
    [1], self.bond_order)
477                             self.temp_bond_list.clear()
478                             self.selected_atom = None
479                             self.update()
480                         return
481                     self.selected = False
482             else:
483                 if self.selected:
484                     potential_radius = Canvas.ATOM_RADIUS
485                     if self.selected_atom is not None:
486                         potential_positions = self.calc_potential_positions(
    self.selected_atom)
487                         for pos in potential_positions:
488                             if not self.check_atom_overlap(pos[0], pos[1]):
489                                 if (
490                                         pos[0] - potential_radius <=
    click_position.x() <= pos[0] + potential_radius and
491                                         pos[1] - potential_radius <=
    click_position.y() <= pos[1] + potential_radius
492                                 ):
493                                     new_atom = chem_editor_logic.Atom(self.
    element, [pos[0], pos[1]])
494                                 if new_atom.check_is_bond_possible(self.
    selected_atom, self.bond_order):
495                                     self.atoms.append(new_atom)
496                                     new_atom.bond(self.selected_atom,
    self.bond_order)
497                                     print("bonded")
498                                     print(self.selected_atom.symbol)
499                         self.selected = False
500                         self.update()
501                         return
502 
503                 # Check if there is an atom at clicked position
504                 if self.check_clicked_on_atom(click_position.x(),
    click_position.y()):
505                     print(f"{self.selected_atom=}")
506                     return
507 
508                 print(self.action_type)
509 
510                 print(self.element)
511                 new_atom = chem_editor_logic.Atom(self.element, [
    click_position.x(), click_position.y()])
512                 self.atoms.append(new_atom)
513                 self.update()
514 
515 
516 class PeriodicTable(QWidget):
517     element_clicked = pyqtSignal(dict)
518 
519     def __init__(self):
520         super(QWidget, self).__init__()
521 
522         self.layout = QGridLayout()
523         self.setLayout(self.layout)
524 
```

```
525         self.elements = None
526 
527         self.load_elements()
528 
529     def load_elements(self) -> None:
530         self.elements = json.load(open("elements.json"))
531 
532         for element, data in self.elements.items():
533             # TODO: add colour for element groups
534 
535             symbol = data["symbol"]
536             group = data["group"]
537             period = data["period"]
538             outer_el = data["outer_electrons"]
539 
540             button = QPushButton(f"{symbol}")
541             button.setFixedSize(40, 40)
542 
543             button.clicked.connect(self.button_clicked)
544 
545             button.setProperty("data", data)
546 
547             button.setToolTip(
548                 f"Element: {element}\nGroup: {group}\nPeriod: {period}\
    nOuter Electrons: {outer_el}")
549 
550             self.layout.addWidget(button, period, group)
551 
552     def button_clicked(self) -> None:
553         sender_button = self.sender()
554 
555         data = sender_button.property("data")
556         self.element_clicked.emit(data)
557 
558         self.hide()
```

Listing 11: chem_editor_gui.py File Program Code

A.6 ChemEditor Logic file

```
from PyQt6.QtWidgets import QMessageBox


def show_dialog(message):
    dlg = QMessageBox()
    dlg.setWindowTitle("Invalid Action!")
    dlg.setText(f"Invalid user action!\n {message}")
    dlg.setIcon(QMessageBox.Icon.Critical)
    button = dlg.exec()

    if button == QMessageBox.StandardButton.Ok:
        print("OK!")


class Atom:
    def __init__(self, element_data: dict, coordinates: list[int, int]):
        self.symbol: str = element_data["symbol"]
        self.outer_electrons: int = element_data["outer_electrons"]
        self.x_coords: int = coordinates[0]
        self.y_coords: int = coordinates[1]
        self.full_shell: int = element_data["full_shell"]
        self.bonds: list[Bond] = []
        self.extra_electrons: int = 0
        self.overall_electrons: int = (self.outer_electrons + self.
    extra_electrons)

    def __eq__(self, other) -> bool:
        return self.symbol == other.symbol and [self.x_coords, self.y_coords
    ] == [other.x_coords, other.y_coords]

    def __add_outer_electrons(self, num: int) -> None:
        self.outer_electrons += num

    def __remove_outer_electrons(self, num: int) -> None:
        self.outer_electrons -= num

    def check_is_bond_possible(self, bonding_atom, order: int = 1) -> bool:
        if (self.overall_electrons + order) > self.full_shell:
            print("Bonding unavailable, shell is full.")
            return False
        elif (bonding_atom.overall_electrons + order) > bonding_atom.
    full_shell:
            print("Bonding unavailable, bonding atoms shell is full.")
            return False
        else:
            return True

    def bond(self, bonding_atom, order: int = 1) -> None:
        if not self.check_is_bond_possible(bonding_atom, order):
            show_dialog("Bonding unavailable, shell is full.")
            return
        new_bond = Bond(self, bonding_atom, order)
        self.bonds.append(new_bond)
        bonding_atom.bonds.append(new_bond)

        self.extra_electrons += order
```

```
54          bonding_atom.extra_electrons += order
55          self.overall_electrons = (self.outer_electrons + self.
    extra_electrons)
56          bonding_atom.overall_electrons = (bonding_atom.outer_electrons +
    bonding_atom.extra_electrons)
57
58      def break_bond(self, atom, order: int = 1) -> None:
59          atom.bonds = [bond for bond in atom.bonds if not (self in bond.atoms
    )]
60          atom.__remove_outer_electrons(order)
61
62
63  class Bond:
64      def __init__(self, atom1: Atom, atom2: Atom, order: int):
65          self.atoms: list = [atom1, atom2]
66          self.order: int = order
```

Listing 12: chem_editor_logic.py File Program Code

A.7 Style.css file

```
1
2 QTabWidget::tab-bar {
3     alignment: center;
4 }
5
6 QTabBar::tab:selected {
7     background-color: darkgray; /* Background color when the tab is selected
       */
8 }
9
10 QTableWidget::item {
11     color: darkgray;
12     border: 1.5px solid #ccc;
13 }
```

A.8 Elements.json file

```
{
  "Hydrogen": {
    "symbol": "H",
    "atomic_number": 1,
    "group": 1,
    "period": 1,
    "outer_electrons": 1,
    "full_shell": 2
  },
  "Helium": {
    "symbol": "He",
    "atomic_number": 2,
    "group": 18,
    "period": 1,
    "outer_electrons": 2,
    "full_shell": 2
  },
  "Lithium": {
    "symbol": "Li",
    "atomic_number": 3,
    "group": 1,
    "period": 2,
    "outer_electrons": 1,
    "full_shell": 8
  },
  "Beryllium": {
    "symbol": "Be",
    "atomic_number": 4,
    "group": 2,
    "period": 2,
    "outer_electrons": 2,
    "full_shell": 8
  },
  "Boron": {
    "symbol": "B",
    "atomic_number": 5,
    "group": 13,
    "period": 2,
    "outer_electrons": 3,
    "full_shell": 8
  },
  "Carbon": {
    "symbol": "C",
    "atomic_number": 6,
    "group": 14,
    "period": 2,
    "outer_electrons": 4,
    "full_shell": 8
  },
  "Nitrogen": {
    "symbol": "N",
    "atomic_number": 7,
    "group": 15,
    "period": 2,
    "outer_electrons": 5,
    "full_shell": 8
```

```
  },
  "Oxygen": {
    "symbol": "O",
    "atomic_number": 8,
    "group": 16,
    "period": 2,
    "outer_electrons": 6,
    "full_shell": 8
  },
  "Fluorine": {
    "symbol": "F",
    "atomic_number": 9,
    "group": 17,
    "period": 2,
    "outer_electrons": 7,
    "full_shell": 8
  },
  "Neon": {
    "symbol": "Ne",
    "atomic_number": 10,
    "group": 18,
    "period": 2,
    "outer_electrons": 8,
    "full_shell": 8
  },
  "Sodium": {
    "symbol": "Na",
    "atomic_number": 11,
    "group": 1,
    "period": 3,
    "outer_electrons": 1,
    "full_shell": 8
  },
  "Magnesium": {
    "symbol": "Mg",
    "atomic_number": 12,
    "group": 2,
    "period": 3,
    "outer_electrons": 2,
    "full_shell": 8
  },
  "Aluminium": {
    "symbol": "Al",
    "atomic_number": 13,
    "group": 13,
    "period": 3,
    "outer_electrons": 3,
    "full_shell": 8
  },
  "Silicon": {
    "symbol": "Si",
    "atomic_number": 14,
    "group": 14,
    "period": 3,
    "outer_electrons": 4,
    "full_shell": 8
  },
  "Phosphorus": {
    "symbol": "P",
```

```
    "atomic_number": 15,
    "group": 15,
    "period": 3,
    "outer_electrons": 5,
    "full_shell": 8
  },
  "Sulfur": {
    "symbol": "S",
    "atomic_number": 16,
    "group": 16,
    "period": 3,
    "outer_electrons": 6,
    "full_shell": 8
  },
  "Chlorine": {
    "symbol": "Cl",
    "atomic_number": 17,
    "group": 17,
    "period": 3,
    "outer_electrons": 7,
    "full_shell": 8
  },
  "Argon": {
    "symbol": "Ar",
    "atomic_number": 18,
    "group": 18,
    "period": 3,
    "outer_electrons": 8,
    "full_shell": 8
  },
  "Potassium": {
    "symbol": "K",
    "atomic_number": 19,
    "group": 1,
    "period": 4,
    "outer_electrons": 1,
    "full_shell": 18
  },
  "Calcium": {
    "symbol": "Ca",
    "atomic_number": 20,
    "group": 2,
    "period": 4,
    "outer_electrons": 2,
    "full_shell": 18
  },
  "Scandium": {
    "symbol": "Sc",
    "atomic_number": 21,
    "group": 3,
    "period": 4,
    "outer_electrons": 3,
    "full_shell": 18
  },
  "Titanium": {
    "symbol": "Ti",
    "atomic_number": 22,
    "group": 4,
    "period": 4,
```

```
    "outer_electrons": 4,
    "full_shell": 18
  },
  "Vanadium": {
    "symbol": "V",
    "atomic_number": 23,
    "group": 5,
    "period": 4,
    "outer_electrons": 5,
    "full_shell": 18
  },
  "Chromium": {
    "symbol": "Cr",
    "atomic_number": 24,
    "group": 6,
    "period": 4,
    "outer_electrons": 6,
    "full_shell": 18
  },
  "Manganese": {
    "symbol": "Mn",
    "atomic_number": 25,
    "group": 7,
    "period": 4,
    "outer_electrons": 7,
    "full_shell": 18
  },
  "Iron": {
    "symbol": "Fe",
    "atomic_number": 26,
    "group": 8,
    "period": 4,
    "outer_electrons": 8,
    "full_shell": 18
  },
  "Cobalt": {
    "symbol": "Co",
    "atomic_number": 27,
    "group": 9,
    "period": 4,
    "outer_electrons": 9,
    "full_shell": 18
  },
  "Nickel": {
    "symbol": "Ni",
    "atomic_number": 28,
    "group": 10,
    "period": 4,
    "outer_electrons": 10,
    "full_shell": 18
  },
  "Copper": {
    "symbol": "Cu",
    "atomic_number": 29,
    "group": 11,
    "period": 4,
    "outer_electrons": 11,
    "full_shell": 18
  },
```

```
  "Zinc": {
    "symbol": "Zn",
    "atomic_number": 30,
    "group": 12,
    "period": 4,
    "outer_electrons": 12,
    "full_shell": 18
  },
  "Gallium": {
    "symbol": "Ga",
    "atomic_number": 31,
    "group": 13,
    "period": 4,
    "outer_electrons": 13,
    "full_shell": 18
  },
  "Germanium": {
    "symbol": "Ge",
    "atomic_number": 32,
    "group": 14,
    "period": 4,
    "outer_electrons": 14,
    "full_shell": 18
  },
  "Arsenic": {
    "symbol": "As",
    "atomic_number": 33,
    "group": 15,
    "period": 4,
    "outer_electrons": 15,
    "full_shell": 18
  },
  "Selenium": {
    "symbol": "Se",
    "atomic_number": 34,
    "group": 16,
    "period": 4,
    "outer_electrons": 16,
    "full_shell": 18
  },
  "Bromine": {
    "symbol": "Br",
    "atomic_number": 35,
    "group": 17,
    "period": 4,
    "outer_electrons": 17,
    "full_shell": 18
  },
  "Krypton": {
    "symbol": "Kr",
    "atomic_number": 36,
    "group": 18,
    "period": 4,
    "outer_electrons": 18,
    "full_shell": 18
  },
  "Rubidium": {
    "symbol": "Rb",
    "atomic_number": 37,
```

```
    "group": 1,
    "period": 5,
    "outer_electrons": 1,
    "full_shell": 18
  },
  "Strontium": {
    "symbol": "Sr",
    "atomic_number": 38,
    "group": 2,
    "period": 5,
    "outer_electrons": 2,
    "full_shell": 18
  },
  "Yttrium": {
    "symbol": "Y",
    "atomic_number": 39,
    "group": 3,
    "period": 5,
    "outer_electrons": 3,
    "full_shell": 18
  },
  "Zirconium": {
    "symbol": "Zr",
    "atomic_number": 40,
    "group": 4,
    "period": 5,
    "outer_electrons": 4,
    "full_shell": 18
  },
  "Niobium": {
    "symbol": "Nb",
    "atomic_number": 41,
    "group": 5,
    "period": 5,
    "outer_electrons": 5,
    "full_shell": 18
  },
  "Molybdenum": {
    "symbol": "Mo",
    "atomic_number": 42,
    "group": 6,
    "period": 5,
    "outer_electrons": 6,
    "full_shell": 18
  },
  "Technetium": {
    "symbol": "Tc",
    "atomic_number": 43,
    "group": 7,
    "period": 5,
    "outer_electrons": 7,
    "full_shell": 18
  },
  "Ruthenium": {
    "symbol": "Ru",
    "atomic_number": 44,
    "group": 8,
    "period": 5,
    "outer_electrons": 8,
```

```
    "full_shell": 18
  },
  "Rhodium": {
    "symbol": "Rh",
    "atomic_number": 45,
    "group": 9,
    "period": 5,
    "outer_electrons": 9,
    "full_shell": 18
  },
  "Palladium": {
    "symbol": "Pd",
    "atomic_number": 46,
    "group": 10,
    "period": 5,
    "outer_electrons": 10,
    "full_shell": 18
  },
  "Silver": {
    "symbol": "Ag",
    "atomic_number": 47,
    "group": 11,
    "period": 5,
    "outer_electrons": 11,
    "full_shell": 18
  },
  "Cadmium": {
    "symbol": "Cd",
    "atomic_number": 48,
    "group": 12,
    "period": 5,
    "outer_electrons": 12,
    "full_shell": 18
  },
  "Indium": {
    "symbol": "In",
    "atomic_number": 49,
    "group": 13,
    "period": 5,
    "outer_electrons": 13,
    "full_shell": 18
  },
  "Tin": {
    "symbol": "Sn",
    "atomic_number": 50,
    "group": 14,
    "period": 5,
    "outer_electrons": 14,
    "full_shell": 18
  },
  "Antimony": {
    "symbol": "Sb",
    "atomic_number": 51,
    "group": 15,
    "period": 5,
    "outer_electrons": 15,
    "full_shell": 18
  },
  "Tellurium": {
```

```
    "symbol": "Te",
    "atomic_number": 52,
    "group": 16,
    "period": 5,
    "outer_electrons": 16,
    "full_shell": 18
  },
  "Iodine": {
    "symbol": "I",
    "atomic_number": 53,
    "group": 17,
    "period": 5,
    "outer_electrons": 17,
    "full_shell": 18
  },
  "Xenon": {
    "symbol": "Xe",
    "atomic_number": 54,
    "group": 18,
    "period": 5,
    "outer_electrons": 18,
    "full_shell": 18
  }
}
```