

# ChemBox Project Documentation

Tom Schneider

**Date:** April 22, 2024

**Center Number:** 29065

**Candidate Number:** 7638

**Ellesmere College**

# Contents

<b>1</b>	<b>Analysis</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	Prospective Users . . . . .	4
1.2.1	Interview with Oliver Covill - Future User . . . . .	4
1.3	Specific Objectives . . . . .	5
1.3.1	Chemical Equations and Calculations - ChemCalculator . . . . .	5
1.3.2	Chemical Equation Balancer - ChemBalancer . . . . .	8
1.3.3	Visualisation of Chemical Molecules - ChemEditor . . . . .	8
1.4	Current and Proposed Systems . . . . .	9
<b>2</b>	<b>Documented Design</b>	<b>10</b>
2.1	Introduction to the Documented Design . . . . .	10
2.2	Project Hierarchy . . . . .	10
2.3	Structure of the GUI . . . . .	11
2.4	Algorithm Design for ChemCalculator . . . . .	14
2.5	Algorithm Design for ChemBalancer . . . . .	15
2.6	Algorithm Design for ChemEditor . . . . .	30
2.6.1	ChemEditor Logic . . . . .	30
2.6.2	ChemEditor GUI . . . . .	33
<b>3</b>	<b>Testing</b>	<b>40</b>
3.0.1	Tests for ChemCalculator . . . . .	40
3.0.2	Tests for ChemBalancer . . . . .	42
3.0.3	Tests for ChemEditor . . . . .	44
<b>4</b>	<b>Evaluation</b>	<b>46</b>
4.0.1	Objective Comparison for ChemCalculator . . . . .	46
4.0.2	Objective Comparison for ChemBalancer . . . . .	46
4.0.3	Objective Comparison for ChemEditor . . . . .	47
4.0.4	Potential for Future Development . . . . .	48
	<b>Appendices</b>	<b>49</b>
<b>A</b>	<b>Program Code</b>	<b>49</b>
A.1	main.py File . . . . .	49
A.2	GUI Components File . . . . .	50
A.3	ChemCalculator File . . . . .	52
A.4	ChemBalancer File . . . . .	82
A.5	ChemEditor GUI file . . . . .	88
A.6	ChemEditor Logic file . . . . .	99

A.7	Style.css file . . . . .	101
A.8	Elements.json file . . . . .	102

# 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:

$$\text{number of atoms} = \text{Avogadro's number} \times \text{moles}$$

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

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

4. Gibbs Free Energy calculation:

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

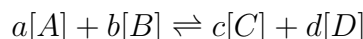
( $\Delta G$  = Gibbs Free Energy) ( $\Delta H$  = enthalpy)  
( $T$  = temperature) ( $\Delta S$  = entropy)

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

$$q = mc\Delta T$$

( $q$  = energy change) ( $m$  = mass) ( $c$  = specific heat capacity)  
( $\Delta T$  = temperature change)

6. Equilibrium Constant calculation for a reversible reaction:



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

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

7. Rate Equation and Rate constant calculation:

$$\text{Rate} = k[A]^m[B]^n$$

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  $pK_a$

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

$$pK_a = -\log K_a$$

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

3. Atom Economy calculation:

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

4. Percentage Yield calculation:

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



### 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 save 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.

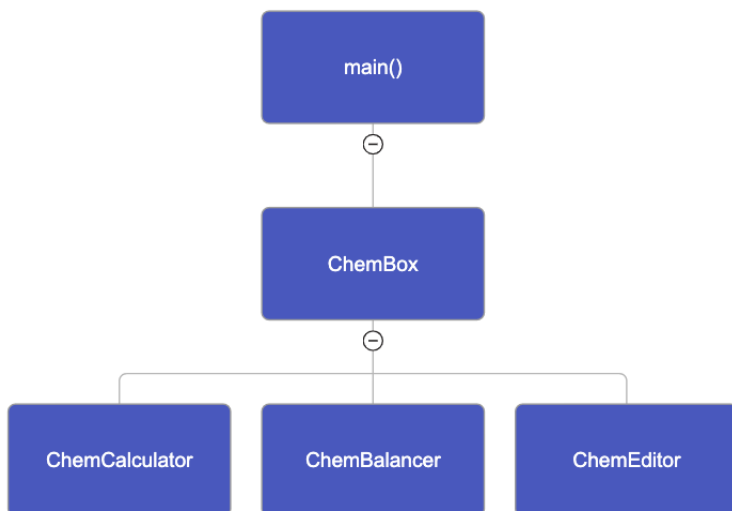


Figure 1: Program Hierarchy chart

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.
13        __height)
14        self.setFixedSize(self.__width, self.__height)
15
16        self.tab_bar = TabBar()
17        self.setCentralWidget(self.tab_bar)
18
19        self.chem_calculator = ChemCalculator()
20        self.tab_bar.tab1.setLayout(self.chem_calculator.main_layout)
21
22        self.chem_balancer = ChemBalancer()
23        self.tab_bar.tab2.setLayout(self.chem_balancer.balancer_layout)
24
25        self.chem_editor = ChemEditor()
26        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.

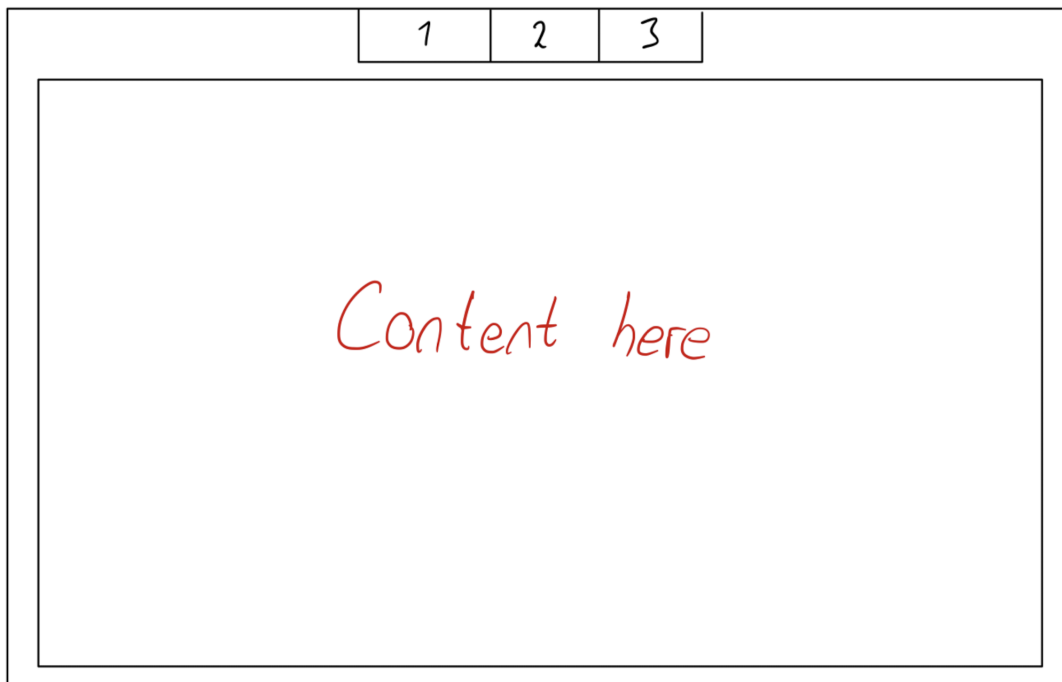


Figure 2: Mockup drawing of tab bar

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 its 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.

```
1 class ChemCalculator(QWidget):
2     def __init__(self):
3         super(QWidget, self).__init__()
4
5         self.side_bar_layout = QVBoxLayout()
6
7         self.gibbs_calc = GibbsFreeEnergyCalculator()
8
9         # Create buttons
10        self.gibbs_free_energy_tab_button = QPushButton("Gibbs Free Energy
11        Calculator")
12
13        self.gibbs_free_energy_tab_button.clicked.connect(self.
14        gibbs_free_energy_action)
15
16        # Create tabs
17        self.gibbs_free_energy_tab = QWidget()
18
19        # Initialise gibbs free energy calculator
20        self.gibbs_free_energy_tab.setLayout(self.gibbs_calc.layout)
21
22        # Add buttons to sidebar layout
23        self.side_bar_layout.addWidget(self.gibbs_free_energy_tab_button)
24
25        self.side_bar_widget = QWidget()
26        self.side_bar_widget.setLayout(self.side_bar_layout)
27
28        self.page_widget = QTabWidget()
29
30        self.page_widget.addTab(self.gibbs_free_energy_tab, "")
31
32        self.page_widget.setCurrentIndex(0)
33        self.page_widget.setStyleSheet('''QTabBar::tab{
34        width: 0;
35        height: 0;
36        margin: 0;
37        padding: 0;
38        border: none;
39        }''')
```

```

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

---

```

 ← []
count ← 0
empty_input ← NONE
FOR  $i \leftarrow 0$  to  $Len(input\_list)$  DO
    IF input_list[ $i$ ] is empty THEN
        count ← count + 1
        empty_input ← 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.

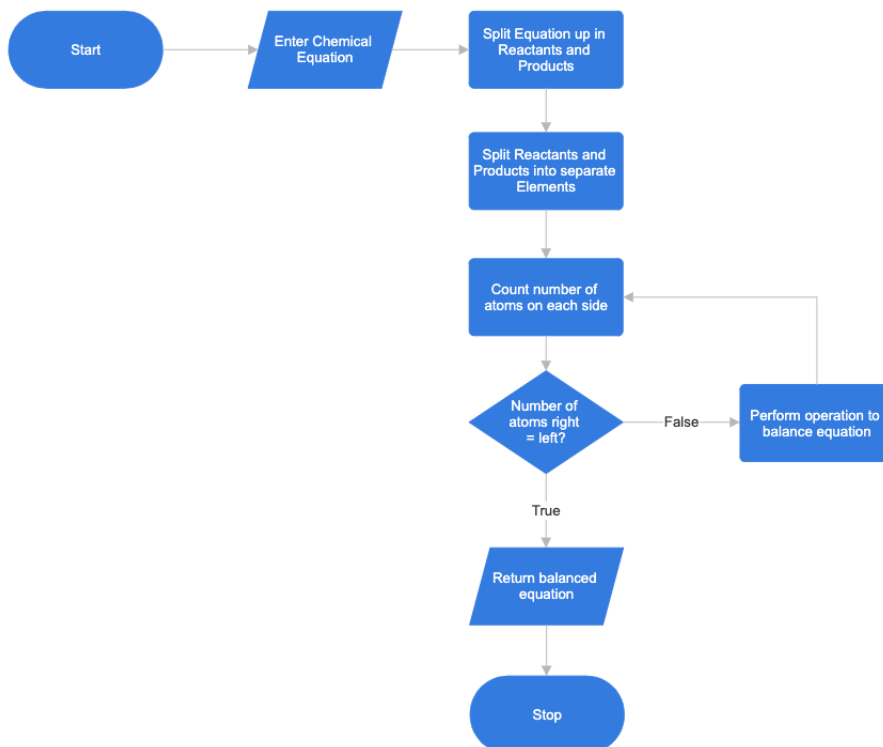


Figure 3: Initial flowchart for balancer

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

```
1 def splitEquation(self):
2     self.equationSplit = self.equationInput.text().split(" = ")
3     self.reactants = self.equationSplit[0]
4     try:
5         self.products = self.equationSplit[1]
6     except IndexError:
7         print("Wrong user input")
8     self.reactantComponents = self.reactants.split(" + ")
9     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 (H<sub>2</sub>O - 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.

```

1 def parseComponent(self, component, countsDict, totalDict):
2     # Check for coefficient
3     try:
4         if component[0] in self.integers:
5             try:
6                 if component[0 + 1] in self.integers:
7                     try:
8                         if component[0 + 2] in self.integers:
9                             coefficient = int(component[0: 0 + 3])
10                        else:
11                            coefficient = int(component[0: 0 + 2])
12                        except IndexError:
13                            coefficient = int(component[0: 0 + 2])
14                    else:
15                        coefficient = int(component[0])
16                except IndexError:
17                    coefficient = int(component[0])
18            else:
19                coefficient = 1
20        for i in range(len(component)):
21            try:
22                openBracket = component.find("(")
23                closedBracket = component.find(")")
24            except IndexError:
25                continue
26            try:
27                if component[i].isupper() and component[i - 1] != "(":
28                    try:
29                        if component[i + 1].islower():
30                            try:
31                                if component[i + 2].islower():
32                                    element = component[i:(i + 3)]
33                                    if openBracket < i < closedBracket:
34                                        subCoefficient = self.
35                                else:
36                                    subCoefficient = 1
37

```

```

38                                     # Check for subscript
39                                     try:
40                                         if component[i + 3] in self.
41                                             integers:
42                                             try:
43                                                 if component[i + 4] in
44                                                     self.integers:
45                                                         try:
46                                                             if component[i +
47                                                                 5] in self.integers:
48                                                                 subscript =
49                                                                 int(component[(i + 3): (i + 6)])
50                                                                 else:
51                                                                 subscript =
52                                                                 int(component[(i + 3): (i + 5)])
53                                                                 except IndexError:
54                                                                 subscript = int(
55                                                                 component[(i + 3): (i + 5)])
56                                                                 else:
57                                                                 subscript = int(
58                                                                 component[i + 3])
59                                                                 except IndexError:
60                                                                 subscript = int(
61                                                                 component[i + 3])
62                                                                 else:
63                                                                 subscript = 1
64                                                                 except IndexError:
65                                                                 subscript = 1
66                                     else:
67                                         element = component[i:(i + 2)]
68                                     if openBracket < i < closedBracket:
69                                         subCoefficient = self.
70                                         getSubCoefficient(component)
71                                     else:
72                                         subCoefficient = 1
73                                     # Check for subscript
74                                     try:
75                                         if component[i + 2] in self.
76                                             integers:
77                                             try:
78                                                 if component[i + 3] in
79                                                     self.integers:
80                                                         try:
81                                                             if component[i +

```

```

82         except IndexError:
83             element = component[i:(i + 2)]
84             if openBracket < i < closedBracket:
85                 subCoefficient = self.
getSubCoefficient(component)
86             else:
87                 subCoefficient = 1
88         else:
89             element = component[i]
90             if openBracket < i < closedBracket:
91                 subCoefficient = self.getSubCoefficient(
component)
92             else:
93                 subCoefficient = 1
94             try:
95                 # Check for subscript
96                 if component[i + 1] in self.integers:
97                     try:
98                         if component[i + 2] in self.
integers:
99                             try:
100                                 if component[i + 3] in
self.integers:
101                                     subscript = int(
component[i + 1: i + 4])
102                                     except IndexError:
103                                         subscript = int(
component[i + 1: i + 3])
104                                     else:
105                                         subscript = int(component[i
+ 1])
106                                     except IndexError:
107                                         subscript = int(component[i +
1])
108                                     else:
109                                         subscript = 1
110                                     except IndexError:
111                                         subscript = 1
112             except IndexError:
113                 element = component[i]
114
115             if openBracket < i < closedBracket:
116                 subCoefficient = self.getSubCoefficient(
component)
117             else:
118                 subCoefficient = 1
119
120             try:
121                 # Check for subscript
122                 if component[i + 1] in self.integers:
123                     try:
124                         if component[i + 2] in self.integers
:
125                             try:
126                                 if component[i + 3] in self.
integers:
127                                     subscript = int(
component[i + 1: i + 4])
128                                     except IndexError:

```

```

129         subscript = int(component[i
+ 1: i + 3])
130
131         else:
132             subscript = int(component[i +
1])
133
134         except IndexError:
135             subscript = int(component[i + 1])
136
137         else:
138             subscript = 1
139
140         except IndexError:
141             subscript = 1
142
143         try:
144             if element in countsDict:
145                 countsDict[element] = int(countsDict[element
]) + subscript * coefficient * subCoefficient
146
147             else:
148                 countsDict[element] = subscript *
coefficient * subCoefficient
149
150             if element in totalDict:
151                 totalDict[element] = int(totalDict[element])
+ subscript * coefficient * subCoefficient
152
153             else:
154                 totalDict[element] = subscript * coefficient
* subCoefficient
155
156         except UnboundLocalError:
157             continue
158
159         # Check for brackets / complex ion in equation
160         elif component[i] == "(":
161             try:
162                 if component[i + 2] == ")":
163                     element = component[i + 1]
164                     try:
165                         if component[i + 3] in self.integers:
166                             try:
167                                 if component[i + 4] in self.
integers:
168
169                                     try:
170                                         if component[i + 5] in
self.integers:
171
172                                             subscript = int(
component[(i + 3): (i + 6)])
173
174                                             else:
175                                                 subscript = int(
component[(i + 3): (i + 5)])
176
177                                         except IndexError:
178                                             subscript = int(
component[(i + 3): (i + 5)])
179
180                                         else:
181                                             subscript = int(component[(i
+ 3): (i + 4)])
182
183                                         except IndexError:
184                                             subscript = int(component[i +
3])
185
186                                         else:
187                                             subscript = 1
188
189                                         except IndexError:
190                                             subscript = 1

```

```

175         elif component[i + 1].isupper():
176             if component[i + 2].islower():
177                 try:
178                     if component[i + 3].islower():
179                         element = component[(i + 1): (i
+ 4)]
180
181             # Check for subscript within
brackets
182             try:
183                 if component[i + 4] in self.
integers:
184                 try:
185                     if component[i + 5]
in self.integers:
186                     try:
187                         if component
[i + 6] in self.integers:
188
subscript = int(component[(i + 4): (i + 7)])
189
else:
190
subscript = int(component[(i + 4): (i + 6)])
191
except
IndexError:
192
subscript =
int(component[(i + 4): (i + 6)])
193
else:
194
subscript = int(
component[i + 4])
195
except IndexError:
196
subscript = int(
component[i + 4])
197
else:
198
subscript = 1
199
except IndexError:
200
subscript = 1
201
202
# Find subscript coefficient of
complex ion
203
subCoefficient = self.
getSubCoefficient(component)
204
else:
205
element = component[(i + 1): (i
+ 3)]
206
207
# Check for subscript within
brackets
208
try:
209
if component[i + 3] in self.
integers:
210
try:
211
if component[i + 4]
in self.integers:
212
try:
213
if component
[i + 5] in self.integers:
214
subscript = int(component[(i + 3): (i + 6)])

```

```

215 |                                     else:
216 |
217 |     subscript = int(component[(i + 3): (i + 5)])
218 |                                     except
219 |     IndexError:
220 |                                     subscript =
221 |     int(component[(i + 3): (i + 5)])
222 |                                     except IndexError:
223 |                                     subscript = int(
224 |     component[i + 3])
225 |                                     else:
226 |                                     subscript = 1
227 |     except IndexError:
228 |                                     subscript = 1
229 |
230 |     # Find subscript coefficient of
231 |     complex ion
232 |     subCoefficient = self.
233 |     getSubCoefficient(component)
234 |     except IndexError:
235 |     element = component[(i + 1): (i + 3)]
236 |
237 |     # Check for subscript within
238 |     brackets
239 |     try:
240 |     if component[i + 3] in self.
241 |     integers:
242 |     try:
243 |     if component[i + 4] in
244 |     self.integers:
245 |     try:
246 |     if component[i +
247 |     5] in self.integers:
248 |     subscript =
249 |     int(component[(i + 3): (i + 6)])
250 |     else:
251 |     subscript =
252 |     int(component[(i + 3): (i + 5)])
253 |     except IndexError:
254 |     subscript = int(
255 |     component[(i + 3): (i + 5)])
256 |     except IndexError:
257 |     subscript = int(
258 |     component[i + 3])
259 |     else:
260 |     subscript = 1
261 |     except IndexError:
262 |     subscript = 1
263 |
264 |     # Find subscript coefficient of
265 |     complex ion
266 |     subCoefficient = self.
267 |     getSubCoefficient(component)
268 |     else:
269 |     element = component[i + 1]
270 |
271 |     # Check for subscript within brackets
272 |     try:

```

```

257         if component[i + 2] in self.integers
258     :
259         try:
260             if component[i + 3] in self.
261                 integers:
262                     try:
263                         if component[i + 4]
264                             subscript = int(
265                                 component[(i + 2): (i + 5)])
266                             else:
267                                 subscript = int(
268                                     component[(i + 2): (i + 4)])
269                             except IndexError:
270                                 subscript = int(
271                                     component[(i + 2): (i + 4)])
272                             except IndexError:
273                                 subscript = int(component[i
274                                     + 2])
275                             else:
276                                 subscript = 1
277                             except IndexError:
278                                 subscript = 1
279                             # Find subscript coefficient of complex
280                             ion
281                             subCoefficient = self.getSubCoefficient(
282                                 component)
283                             except IndexError:
284                                 print("wrong user input ")
285                             try:
286                                 if element in countsDict:
287                                     countsDict[element] += subscript *
288                                     subCoefficient * coefficient
289                                 else:
290                                     countsDict[element] = subscript *
291                                     subCoefficient * coefficient
292                                 if element in totalDict:
293                                     totalDict[element] += subscript *
294                                     subCoefficient * coefficient
295                                 else:
296                                     totalDict[element] = subscript *
297                                     subCoefficient * coefficient
298                                 except UnboundLocalError:
299                                     continue
300                             else:
301                                 continue
302                             except IndexError:
303                                 continue
304                             except IndexError:
305                                 continue
306                             except IndexError:
307                                 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.

```

1  def balance(self):
2      if self.balanced:
3          equation = str()
4          for dictionary in self.left:
5              compound = str()
6              for element in dictionary:
7                  compound += element
8                  if dictionary[element] > 1:
9                      compound += str(dictionary[element])
10                 equation += compound
11                 equation += " + "
12             equation = equation[:len(equation) - 3] + " = "
13         for dictionary in self.right:
14             compound = str()
15             for element in dictionary:
16                 compound += element
17                 if dictionary[element] > 1:
18                     compound += str(dictionary[element])
19             else:
20                 pass
21             equation += compound
22             equation += " + "
23         equation = equation[:len(equation) - 2]
24     else:
25         while not self.balanced:
26             tempLeft = list()
27             tempRight = list()
28             totalLeft = dict()
29             totalRight = dict()
30
31             for item in self.left:
32                 newDict = dict()
33                 for key in item:
34                     newDict[key] = item[key]
35                 tempLeft.append(newDict)
36
37             for item in self.right:
38                 newDict = dict()
39                 for key in item:
40                     newDict[key] = item[key]
41                 tempRight.append(newDict)
42
43             leftCoefficients = [randint(1, 10) for _ in range(len(
44                 tempLeft))]

```



```

45     tempRight))]
46
47         for index in range(0, len(leftCoefficients)):
48             for key in tempLeft[index]:
49                 tempLeft[index][key] *= leftCoefficients[index]
50                 if key not in totalLeft:
51                     totalLeft[key] = tempLeft[index][key]
52                 else:
53                     totalLeft[key] += tempLeft[index][key]
54             for index in range(0, len(rightCoefficients)):
55                 for key in tempRight[index]:
56                     tempRight[index][key] *= rightCoefficients[index]
57                     if key not in totalRight:
58                         totalRight[key] = tempRight[index][key]
59                     else:
60                         totalRight[key] += tempRight[index][key]
61         self.balanced = True
62         for key in totalLeft:
63             if totalLeft[key] != totalRight[key]:
64                 self.balanced = False
65         else:
66             continue
67
68         bigTup = tuple(leftCoefficients + rightCoefficients)
69         leftCoefficients = list(map(lambda x: int(x / reduce(gcd, bigTup
70         )), leftCoefficients))
71         rightCoefficients = list(map(lambda x: int(x / reduce(gcd,
72         bigTup)), rightCoefficients))
73
74         balancedEquation = str()
75         for index in range(0, len(self.left)):
76             if leftCoefficients[index] != 1:
77                 compound = str(leftCoefficients[index])
78             else:
79                 compound = str()
80             for key in self.left[index]:
81                 compound += key
82                 if self.left[index][key] != 1:
83                     compound += str(self.left[index][key])
84             balancedEquation += compound
85             balancedEquation += " + "
86         balancedEquation = balancedEquation[:len(balancedEquation) - 3]
87         + " = "
88         for index in range(0, len(self.right)):
89             if rightCoefficients[index] != 1:
90                 compound = str(rightCoefficients[index])
91             else:
92                 compound = str()
93             for key in self.right[index]:
94                 compound += key
95                 if self.right[index][key] != 1:
96                     compound += str(self.right[index][key])
97             balancedEquation += compound
98             balancedEquation += " + "
99         balancedEquation = balancedEquation[:len(balancedEquation) - 2]
100         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.

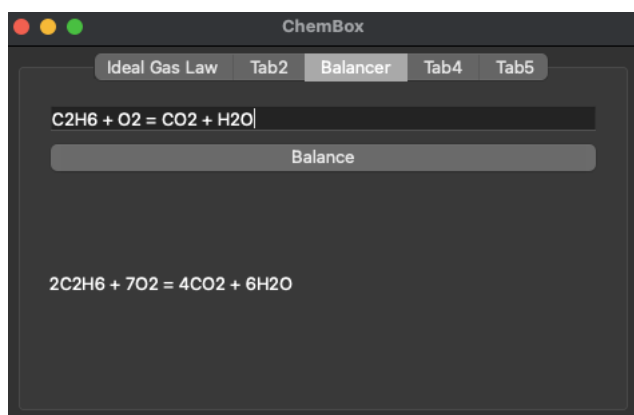


Figure 4: Initial balancer test

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  $C_2H_6 + O_2 = CO_2 + H_2O$  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.

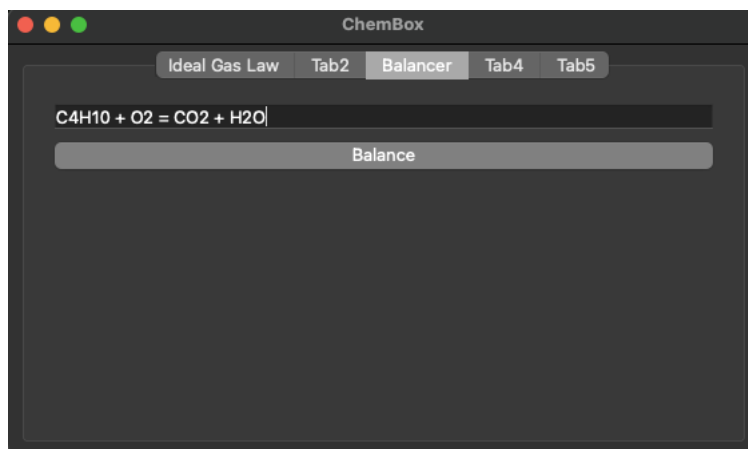


Figure 5: Initial balancer test No. 2

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 white spaces 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(" ")
    equation_split  $\leftarrow$  stripped_equation.SPLIT(" = ")
    reactants  $\leftarrow$  equation_split[0].SPLIT(" + ")
    products  $\leftarrow$  equation_split[1].SPLIT(" + ")
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  $Cu(NO_3)_2$  into  $[Cu, (NO_3)_2]$ , where  $Cu$  and  $(NO_3)_2$  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*)<sub>2</sub> 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:  $Cr_2O_7 \rightarrow [("Cr", "2"), ("O", "7")]$ ). 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  $\leftarrow$  SPLIT reagent INTO element_counts USING REGEX PATTERN  
  FOR element, subscript IN element_counts DO  
    IF subscript DOES NOT EXIST THEN  
      subscript  $\leftarrow$  1  
    ELSE  
      subscript  $\leftarrow$  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 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 zero<sup>th</sup> vector under a given linear transformation: multiplication with the matrix A, represented as  $Ax = 0$ . 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 python's 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).

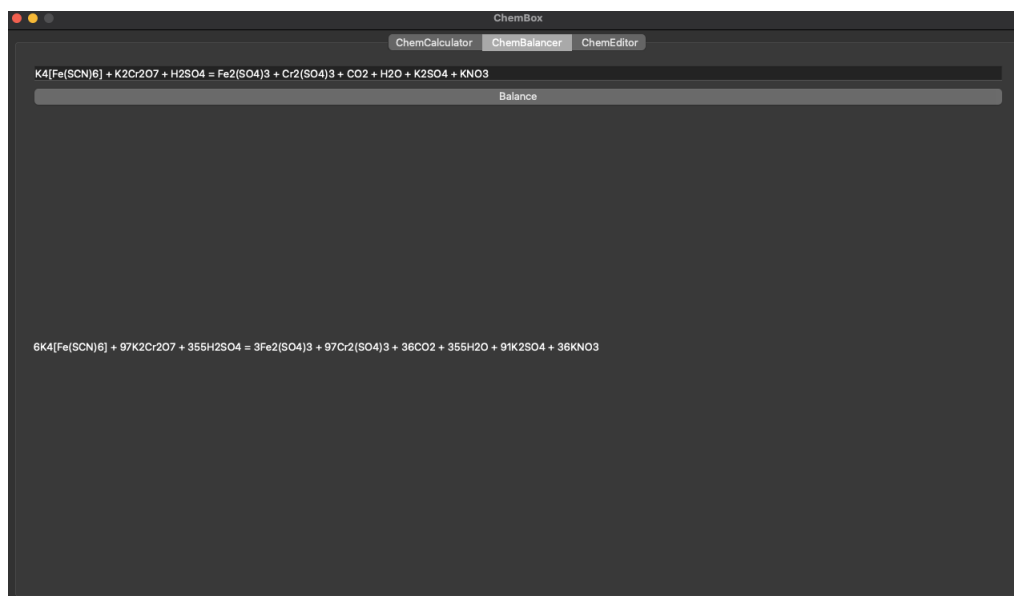


Figure 7: Balanced equation

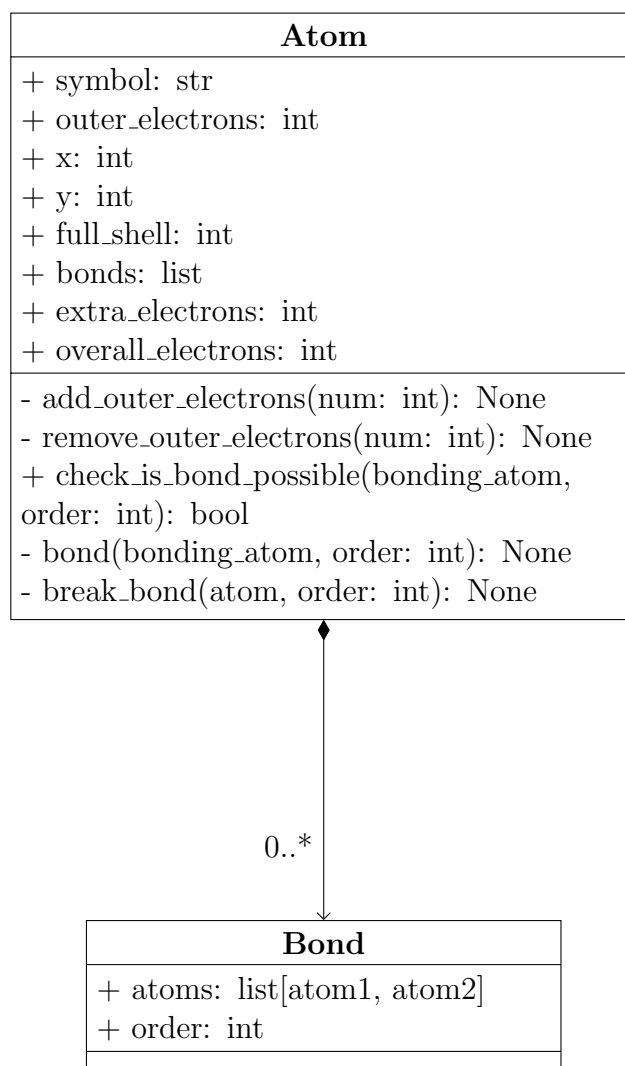
## 2.6 Algorithm Design for ChemEditor

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  $\leftarrow$  Bond(bonding_atom, order)
  bonds.append(new_bond)
  bonding_atom.bonds.append(new_bond)
  extra_electrons  $\leftarrow$  extra_electrons + order
  bonding_atom.extra_electrons  $\leftarrow$  bonding_atom.extra_electrons + order
  overall_electrons  $\leftarrow$  outer_electrons + extra_electrons
  bonding_atom.overall_electrons  $\leftarrow$  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 if 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).

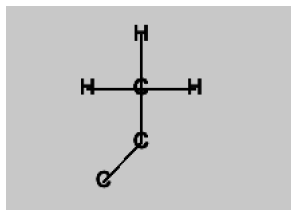


Figure 8: Overlap between atoms and bonds

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).

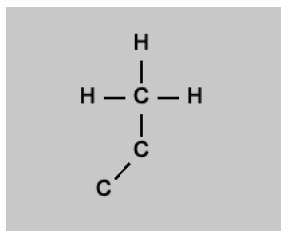


Figure 9: No overlap between atoms and bonds

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)
   $x \leftarrow atom.x\_coords$ 
   $y \leftarrow atom.y\_coords$ 
   $distance \leftarrow 40$ 
   $coordinate\_list \leftarrow NONE$ 
  FOR  $angle\_degrees \leftarrow 0$  TO 360 STEP 45 DO
     $angle\_radians \leftarrow math.radians(angle\_degrees)$ 
     $new\_x \leftarrow x + distance * math.cos(angle\_radians)$ 
     $new\_y \leftarrow y + distance * math.sin(angle\_radians)$ 
    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  $\leftarrow$  selected_atom.x_coords
  y1  $\leftarrow$  selected_atom.y_coords
  IF action_type! = "bond" THEN
    IF selected_atom IS NOT NONE THEN
      potential_positions  $\leftarrow$  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], pos[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
      END FOR
      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  $\leftarrow$  calc_potential_positions(selected_atom)
    IF check_atom_overlap(pos) = False THEN
      FOR pos IN potential_positions DO
        IF pos = clickposition THEN
          new_atom  $\leftarrow$  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  $\leftarrow$  False
    END IF
  END IF
IF check_clicked_on_atom(pos) = True THEN
  RETURN
END IF
new_atom  $\leftarrow$  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: Avogadro's 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:  $\text{C}_4\text{H}_{10} + \text{O}_2 \rightarrow \text{CO}_2 + \text{H}_2\text{O}$   
Balanced equation:  $2\text{C}_4\text{H}_{10} + 13\text{O}_2 \rightarrow 8\text{CO}_2 + 10\text{H}_2\text{O}$
2. Unbalanced equation:  $(\text{NH}_4)_2\text{Cr}_2\text{O}_7 \rightarrow \text{N}_2 + \text{Cr}_2\text{O}_3 + \text{H}_2\text{O}$   
Balanced equation:  $(\text{NH}_4)_2\text{Cr}_2\text{O}_7 \rightarrow \text{N}_2 + \text{Cr}_2\text{O}_3 + 4\text{H}_2\text{O}$
3. Unbalanced equation:  $\text{C}_{57}\text{H}_{110}\text{O}_6 + \text{O}_2 \rightarrow \text{CO}_2 + \text{H}_2\text{O}$   
Balanced equation:  $2\text{C}_{57}\text{H}_{110}\text{O}_6 + 163\text{O}_2 \rightarrow 114\text{CO}_2 + 110\text{H}_2\text{O}$
4. Unbalanced equation:  $\text{KNO}_3 + \text{C}_{12}\text{H}_{22}\text{O}_{11} \rightarrow \text{N}_2 + \text{CO}_2 + \text{H}_2\text{O} + \text{K}_2\text{CO}_3$   
Balanced equation:  $48\text{KNO}_3 + 5\text{C}_{12}\text{H}_{22}\text{O}_{11} \rightarrow 24\text{N}_2 + 36\text{CO}_2 + 55\text{H}_2\text{O} + 24\text{K}_2\text{CO}_3$
5. Unbalanced equation:  $\text{Cu}_2\text{S} + \text{HNO}_3 \rightarrow \text{Cu}(\text{NO}_3)_2 + \text{CuSO}_4 + \text{NO}_2 + \text{H}_2\text{O}$   
Balanced equation:  $1\text{Cu}_2\text{S} + 12\text{HNO}_3 \rightarrow 1\text{Cu}(\text{NO}_3)_2 + 1\text{CuSO}_4 + 10\text{NO}_2 + 6\text{H}_2\text{O}$

6. Unbalanced equation:  $\text{K}_4[\text{Fe}(\text{SCN})_6] + \text{K}_2\text{Cr}_2\text{O}_7 + \text{H}_2\text{SO}_4 \rightarrow \text{Fe}_2(\text{SO}_4)_3 + \text{Cr}_2(\text{SO}_4)_3 + \text{CO}_2 + \text{H}_2\text{O} + \text{K}_2\text{SO}_4 + \text{KNO}_3$   
 Balanced equation:  $6\text{K}_4[\text{Fe}(\text{SCN})_6] + 97\text{K}_2\text{Cr}_2\text{O}_7 + 355\text{H}_2\text{SO}_4 \rightarrow 3\text{Fe}_2(\text{SO}_4)_3 + 97\text{Cr}_2(\text{SO}_4)_3 + 36\text{CO}_2 + 355\text{H}_2\text{O} + 91\text{K}_2\text{SO}_4 + 36\text{KNO}_3$
7. Unbalanced equation:  $\text{Na}_2\text{S}_2\text{O}_4 + \text{NaOH} \rightarrow \text{Na}_2\text{SO}_3 + \text{Na}_2\text{S} + \text{H}_2\text{O}$   
 Balanced equation:  $3\text{Na}_2\text{S}_2\text{O}_4 + 6\text{NaOH} \rightarrow 5\text{Na}_2\text{SO}_3 + 1\text{Na}_2\text{S} + 3\text{H}_2\text{O}$
8. Unbalanced equation:  $\text{C}_6\text{H}_8\text{O}_7 + \text{NaHCO}_3 \rightarrow \text{Na}_3\text{C}_6\text{H}_6\text{O}_7 + \text{CO}_2 + \text{H}_2\text{O}$   
 Balanced equation:  $19\text{C}_6\text{H}_8\text{O}_7 + 54\text{NaHCO}_3 \rightarrow 18\text{Na}_3\text{C}_6\text{H}_6\text{O}_7 + 60\text{CO}_2 + 49\text{H}_2\text{O}$
9. Unbalanced equation:  $\text{P}_4\text{O}_{10} + \text{H}_2\text{O} \rightarrow \text{H}_3\text{PO}_4$   
 Balanced equation:  $1\text{P}_4\text{O}_{10} + 6\text{H}_2\text{O} \rightarrow 4\text{H}_3\text{PO}_4$

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.:  $\text{C}_4\text{H}_{10} + \text{O}_2 \rightarrow \text{CO}_2 + \text{H}_2\text{O}$  instead of  $\text{C}_4\text{H}_{10} + \text{O}_2 = \text{CO}_2 + \text{H}_2\text{O}$ ), 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.

```
Traceback (most recent call last):
  File "/Users/tom/Documents/Programmieren/ChemBox/chem_balancer.py", line 242, in run_balancer
    self.find_reagents(self.reactants[1], 1, 1)
  File "/Users/tom/Documents/Programmieren/ChemBox/chem_balancer.py", line 160, in find_reagents
    self.find_elements(reagent, index, bracket_subscript, side)
  File "/Users/tom/Documents/Programmieren/ChemBox/chem_balancer.py", line 185, in find_elements
    self.add_to_matrix(element, index, bracket_subscript * subscript, side)
  File "/Users/tom/Documents/Programmieren/ChemBox/chem_balancer.py", line 222, in add_to_matrix
    self.element_matrix[index][column] += count * side
    ~~~~~^~~~~~
  File "/Users/tom/Documents/Programmieren/ChemBox/venv/lib/python3.11/site-packages/sympy/matrices/repnmatrix.py", line 233, in __getitem__
    return _getitem_RepMatrix(self, key)
    ~~~~~^~~~~~
  File "/Users/tom/Documents/Programmieren/ChemBox/venv/lib/python3.11/site-packages/sympy/matrices/repnmatrix.py", line 741, in _getitem_RepMatrix
    return [][key]
    ~~~~~^~~~~~
IndexError: list index out of range

Process finished with exit code 134 (interrupted by signal 6:SIGABRT)
```

Figure 10: Balancer Crash Report

I have resolved this issue using python's exception handling. As the Chem-Balancer 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 of this solution is shown in Listing 6.

```

1 def lowest_level_function():
2     try:
3         # Do something...
4     except Exception:
5         raise # Raise the exception to be caught by the next higher level
6             function
7
8 def middle_level_function():
9     try:
10         lowest_level_function()
11     except Exception:
12         raise # Re-raise the exception to be caught by the highest level
13             function
14
15 def highest_level_function():
16     try:
17         middle_level_function()
18     except Exception:
19         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.

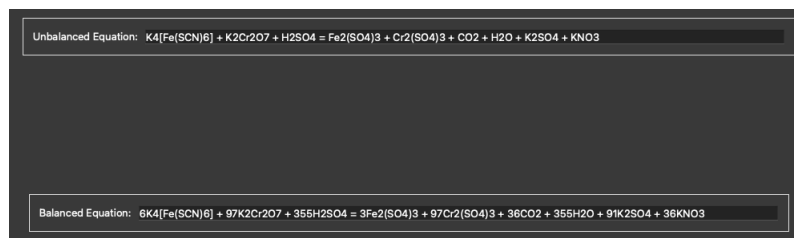


Figure 11: Working test of balancer

### 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 area 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.

# Appendices

## A Program Code

### A.1 main.py File

```
1 import sys
2
3 from PyQt6.QtWidgets import QApplication, QMainWindow
4
5 from chem_editor_gui import ChemEditor
6
7 from chem_calculator import ChemCalculator
8
9 from chem_balancer import ChemBalancer
10 from gui_comps import TabBar
11
12
13 class ChemBox(QMainWindow):
14     def __init__(self):
15         super().__init__()
16
17         # set window properties
18         self.__left = 300
19         self.__top = 300
20         self.__width = 1280
21         self.__height = 720
22         self.__title = "ChemBox"
23         self.setWindowTitle(self.__title)
24         self.setGeometry(self.__left, self.__top, self.__width, self.
25         __height)
26         self.setFixedSize(self.__width, self.__height)
27
28         self.tab_bar = TabBar()
29         self.setCentralWidget(self.tab_bar)
30
31         self.chem_calculator = ChemCalculator()
32
33         self.tab_bar.tab1.setLayout(self.chem_calculator.main_layout)
34
35         self.chem_balancer = ChemBalancer()
36         self.tab_bar.tab2.setLayout(self.chem_balancer.balancer_layout)
37
38         self.chem_editor = ChemEditor()
39         self.tab_bar.tab3.setLayout(self.chem_editor.editor_layout)
40
41 def main():
42     app = QApplication(sys.argv)
43
44     # Load CSS file
45     app.setStyleSheet(open('style.css').read())
46
47     main_win = ChemBox()
48     main_win.show()
```

```

49     sys.exit(app.exec())
50
51
52 if __name__ == "__main__":
53     main()

```

Listing 7: main.py File Program Code

## A.2 GUI Components File

```

1 from PyQt6.QtWidgets import QTabWidget, QWidget, QHBoxLayout, QGridLayout,
  QFrame, QLabel, QLineEdit, QComboBox
2
3
4 class TabBar(QWidget):
5     def __init__(self):
6         super(QWidget, self).__init__()
7
8         self.layout = QHBoxLayout(self)
9
10        self.tabs = QTabWidget()
11        self.tab1 = QWidget()
12        self.tab2 = QWidget()
13        self.tab3 = QWidget()
14        self.tabs.addTab(self.tab1, "ChemCalculator")
15        self.tabs.addTab(self.tab2, "ChemBalancer")
16        self.tabs.addTab(self.tab3, "ChemEditor")
17
18        self.layout.addWidget(self.tabs)
19        self.setLayout(self.layout)
20
21
22 class RateBox(QWidget):
23     """
24     This class contains the gui components for molecules for the
25     RateCalculator.
26
27     #####
28     # Order of reaction: ----- #
29     # Concentration: ----- #
30     #####
31     """
32
33     def __init__(self, symbol):
34         super(QWidget, self).__init__()
35
36         self.layout = QGridLayout()
37
38         self.box = QFrame(self)
39         self.box.setFrameStyle(0x0001)
40         self.box.setLineWidth(3)
41
42         self.box_layout = QGridLayout()
43
44         self.order_label = QLabel("Order of reaction: ")
45         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

```
1 from PyQt6.QtGui import QFont
2 from PyQt6.QtWidgets import QGridLayout, QWidget, QPushButton, QLineEdit,
  QLabel, QComboBox, QHBoxLayout, QTabWidget, \
3     QVBoxLayout, QMessageBox
4
5 from math import log
6
7 import re
8
9 from gui_comps import RateBox, RateResultBox
10
11
12 def is_numeric(user_input):
13     if not user_input:
14         return True
15     # Regular expression to match integers or decimals
16     pattern = r'^[-+]?[0-9]*\.[0-9]+$'
17     return bool(re.match(pattern, user_input))
18
19
20 def find_empty_input(input_list: list[QLineEdit]) -> QLineEdit | None:
21     """
22     An algorithm for finding the empty input out of a list of inputs.
23     Only works when looking for a single empty input.
24     """
25
26     count = 0
27     empty = None
28
29     for i in range(len(input_list)):
30         if not input_list[i].text().strip():
31             count += 1
32             empty = input_list[i]
33
34     if count == 1:
35         return empty
36
37
38 def check_invalid_symbol(input_list: list[QLineEdit]) -> bool:
39     """
40     An algorithm for checking for invalid symbols.
41     """
42
43     invalid = False
44
45     for i in range(len(input_list)):
46         if not is_numeric(input_list[i].text().strip()):
47             invalid = True
48
49     return invalid
50
51
52 def show_dialog(message):
53     dlg = QMessageBox()
54     dlg.setWindowTitle("Invalid Input!")
55     dlg.setText(f"Invalid user input!\n {message}")
```

```

56     dlg.setIcon(QMessageBox.Icon.Critical)
57     button = dlg.exec()
58
59     if button == QMessageBox.StandardButton.Ok:
60         print("OK!")
61
62
63 class ChemCalculator(QWidget):
64     def __init__(self):
65         super(QWidget, self).__init__()
66
67         self.side_bar_layout = QVBoxLayout()
68
69         self.moles_calc = MolesCalculator()
70         self.concentration_calc = ConcCalculator()
71         self.avogadro_calc = AvogadroCalculator()
72         self.ideal_gas_calc = IdealGasLawCalculator()
73         self.equilibrium_calc = EquilibriumCalculator()
74         self.gibbs_calc = GibbsFreeEnergyCalculator()
75         self.specific_heat_calc = SpecificHeatCalculator()
76         self.rate_calc = RateCalculator()
77
78         # Create buttons
79         self.moles_tab_button = QPushButton("Moles")
80         self.conc_tab_button = QPushButton("Concentration")
81         self.avogadro_tab_button = QPushButton("Avogadro's Calculator")
82         self.ideal_gas_tab_button = QPushButton("Ideal Gas Equation")
83         self.equilibrium_tab_button = QPushButton("Equilibrium Constant")
84         self.gibbs_free_energy_tab_button = QPushButton("Gibbs Free Energy
Calculator")
85         self.specific_heat_tab_button = QPushButton("Specific Heat
Calculator")
86         self.rate_tab_button = QPushButton("Rate Constant")
87
88         self.moles_tab_button.clicked.connect(self.moles_action)
89         self.conc_tab_button.clicked.connect(self.conc_action)
90         self.avogadro_tab_button.clicked.connect(self.avogadro_action)
91         self.ideal_gas_tab_button.clicked.connect(self.ideal_gas_action)
92         self.equilibrium_tab_button.clicked.connect(self.equilibrium_action)
93         self.gibbs_free_energy_tab_button.clicked.connect(self.
gibbs_free_energy_action)
94         self.specific_heat_tab_button.clicked.connect(self.
specific_heat_action)
95         self.rate_tab_button.clicked.connect(self.rate_action)
96
97         # Create tabs
98         self.moles_tab = QWidget()
99         self.conc_tab = QWidget()
100        self.avogadro_tab = QWidget()
101        self.ideal_gas_tab = QWidget()
102        self.equilibrium_tab = QWidget()
103        self.gibbs_free_energy_tab = QWidget()
104        self.specific_heat_tab = QWidget()
105        self.rate_tab = QWidget()
106
107        # Initialise moles tab in sidebar
108        self.moles_tab.setLayout(self.moles_calc.moles_layout)
109
110        # 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
258         performed.
259         """
260
261         if not find_empty_input(self.input_list.copy()):
262             show_dialog("Must leave exactly one input line empty for it to
263             be calculated!")
264             return
265         elif check_invalid_symbol(self.input_list.copy()):
266             show_dialog("Only numerical values in the form of integers or
267             decimals allowed!")
268             return
269
270         mass_unit = self.mass_conversions[self.mass_unit_dropdown.
271         currentText()]
272         moles_unit = self.mole_conversions[self.moles_unit_dropdown.
273         currentText()]
274
275         to_calc = find_empty_input(self.input_list.copy())
276
277         if to_calc is self.moles_input:
278             self.calculate_moles(mass_unit)
279         elif to_calc is self.mass_input:
280             self.calculate_mass(moles_unit)
281         elif to_calc is self.mr_input:
282             self.calculate_mr(mass_unit, moles_unit)
283         else:
284             return
285
286     def calculate_moles(self, mass_unit):

```

```

282     """
283     Calculates the moles and calls for an update of the gui input and
284     adjusts the moles unit dropdown.
285     """
286     moles = (float(self.mass_input.text()) * mass_unit) / float(self.
287     mr_input.text())
288     self.update_input(moles)
289     self.moles_unit_dropdown.setCurrentIndex(2)
290
291     def calculate_mass(self, moles_unit):
292         """
293         Calculates the mass and calls for an update of the gui input and
294         adjusts the moles unit dropdown.
295         """
296         mass = (float(self.moles_input.text()) * moles_unit) * float(
297         self.mr_input.text())
298         self.update_input(mass)
299         self.mass_unit_dropdown.setCurrentIndex(1)
300
301     def calculate_mr(self, mass_unit, moles_unit):
302         """
303         Calculates the mr and calls for an update of the gui input.
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
312         then updates it using the result parameter.
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.
319         moles_layout}
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     # Initialise concentration calculation Layout
354     self.conc_label = QLabel("Concentration:")
355     self.moles_label = QLabel("Moles:")
356     self.vol_label = QLabel("Volume:")
357
358     self.conc_input = QLineEdit()
359     self.moles_input = QLineEdit()
360     self.vol_input = QLineEdit()
361
362     self.input_list = [self.conc_input, self.moles_input, self.vol_input
363 ]
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
387         performed.
388         """
389         if not find_empty_input(self.input_list.copy()):
390             show_dialog("Must leave exactly one input line empty for it to
391 be calculated!")
392             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
577     if to_calc is self.num_atoms_input:
578         self.update_num_atoms(str(self.calculate_num_atoms()))
579
580 def calculate_mass(self):
581     mass = float(self.moles_input.text()) * float(self.
molecular_weight_input.text())
582     return mass
583
584 def calculate_moles(self, mass_unit):
585     try:
586         moles = float(self.num_atoms_input.text()) / self.
avogadros_constant
587     except ValueError:
588         try:
589             moles = (float(self.mass_input.text()) * self.
mass_conversions[mass_unit]) / float(
590                 self.molecular_weight_input.text())
591             return moles
592         except ValueError:
593             return ""
594     return moles
595
596 def calculate_mol_weight(self, mass_unit):
597     mol_weight = (float(self.mass_input.text()) * self.mass_conversions[
mass_unit]) / float(
598         self.moles_input.text())
599     return mol_weight
600
601 def calculate_num_atoms(self):
602     num_atoms = float(self.moles_input.text()) * self.avogadros_constant
603     return num_atoms
604
605 def update_mass(self, mass):
606     self.mass_input.setText(mass)
607
608 def update_moles(self, moles):
609     self.moles_input.setText(moles)

```

```

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.
785                        conc_d, self.coeff_a, self.coeff_b, self.coeff_c,
786                        self.coeff_d, self.equilibrium_constant]
787     self.concentration_list = [self.conc_a, self.conc_b, self.conc_c,
788                               self.conc_d]
789     self.coeff_list = [self.coeff_a, self.coeff_b, self.coeff_c, self.
790                       coeff_d]
791
792     self.calculated_value = None
793
794     self.layout.addWidget(self.equation_label, 0, 0)
795     self.layout.addWidget(self.calc_button, 10, 0, 2, 0)
796
797     self.layout.addWidget(self.conc_a_label, 1, 0)
798     self.layout.addWidget(self.conc_a, 1, 1)
799     self.layout.addWidget(self.coeff_a_label, 2, 0)
800     self.layout.addWidget(self.coeff_a, 2, 1)
801     self.layout.addWidget(self.conc_b_label, 3, 0)
802     self.layout.addWidget(self.conc_b, 3, 1)
803     self.layout.addWidget(self.coeff_b_label, 4, 0)
804     self.layout.addWidget(self.coeff_b, 4, 1)
805     self.layout.addWidget(self.conc_c_label, 5, 0)
806     self.layout.addWidget(self.conc_c, 5, 1)
807     self.layout.addWidget(self.coeff_c_label, 6, 0)
808     self.layout.addWidget(self.coeff_c, 6, 1)
809     self.layout.addWidget(self.conc_d_label, 7, 0)
810     self.layout.addWidget(self.conc_d, 7, 1)
811     self.layout.addWidget(self.coeff_d_label, 8, 0)
812     self.layout.addWidget(self.coeff_d, 8, 1)
813     self.layout.addWidget(self.equilibrium_constant_label, 9, 0)
814     self.layout.addWidget(self.equilibrium_constant, 9, 1)
815
816     def check_empty_input(self):
817         empty_count = 0
818         empty_input = None
819
820         if not find_empty_input(self.input_list.copy()):
821             show_dialog("Must leave exactly one input line empty for it to
be calculated!")
822             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.
872 text())) * (
873                 float(self.conc_d.text()) ** float(self.coeff_d.text
874 ()))) / (
875                 float(self.equilibrium_constant.text()) * (
876                 float(self.conc_a.text()) ** float(self.
877 coeff_a.text()))
878             if float(self.coeff_b.text()) > 1:
879                 conc = conc ** (1 / float(self.coeff_b.text()))
880         except OverflowError:
881             print("Overflow Error, inputted numbers too large")
882             show_dialog("Overflow Error, inputted numbers are too large!")
883     ")
884
885     return
886     self.calculated_value = to_find
887     self.update_gui(to_find, conc)
888     return
889
890     if to_find == self.conc_c or to_find == self.calculated_value:
891         try:
892             conc = ((float(self.conc_a.text()) ** float(self.coeff_a.
893 text())) * (
894                 float(self.conc_b.text()) ** float(self.coeff_b.text
895 ())) * (
896                 float(self.equilibrium_constant.text())) / (
897                 float(self.conc_d.text()) ** float(self.
898 coeff_d.text()))
899             if float(self.coeff_c.text()) > 1:
900                 conc = conc ** (1 / float(self.coeff_c.text()))
901         except OverflowError:
902             print("Overflow Error, inputted numbers too large")
903             show_dialog("Overflow Error, inputted numbers are too large!")
904     ")
905
906     return
907     self.calculated_value = to_find
908     self.update_gui(to_find, conc)
909     return
910
911     if to_find == self.conc_d or to_find == self.calculated_value:
912         try:
913             conc = ((float(self.conc_a.text()) ** float(self.coeff_a.
914 text())) * (
915                 float(self.conc_b.text()) ** float(self.coeff_b.text
916 ())) * (
917                 float(self.equilibrium_constant.text())) / (
918                 float(self.conc_c.text()) ** float(self.
919 coeff_c.text()))
920             if float(self.coeff_d.text()) > 1:
921                 conc = conc ** (1 / float(self.coeff_d.text()))
922         except OverflowError:
923             print("Overflow Error, inputted numbers too large")
924             show_dialog("Overflow Error, inputted numbers are too large!")
925     ")
926
927     return
928     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.
920 text())) * (
921                             float(self.conc_d.text()) ** float(self.coeff_d.text
922 ()))) / (
923                             float(self.equilibrium_constant.text()) * (
924                             float(self.conc_b.text()) ** float(self.
925 coeff_b.text()))))
926                 coeff = int(log(coeff, float(self.conc_a.text())))
927             except OverflowError:
928                 print("Overflow Error, inputted numbers too large")
929                 show_dialog("Overflow Error, inputted numbers are too large!")
930         ")
931         return
932         self.calculated_value = to_find
933         self.update_gui(to_find, coeff)
934         return
935
936     if to_find == self.coeff_b or to_find == self.calculated_value:
937         try:
938             coeff = ((float(self.conc_c.text()) ** float(self.coeff_c.
939 text())) * (
940                             float(self.conc_d.text()) ** float(self.coeff_d.text
941 ()))) / (
942                             float(self.equilibrium_constant.text()) * (
943                             float(self.conc_a.text()) ** float(self.
944 coeff_a.text()))))
945                 coeff = int(log(coeff, float(self.conc_b.text())))
946             except OverflowError:
947                 print("Overflow Error, inputted numbers too large")
948                 show_dialog("Overflow Error, inputted numbers are too large!")
949         ")
950         return
951         self.calculated_value = to_find
952         self.update_gui(to_find, coeff)
953         return
954
955     if to_find == self.coeff_c or to_find == self.calculated_value:
956         try:
957             coeff = ((float(self.conc_a.text()) ** float(self.coeff_a.
958 text())) * (
959                             float(self.conc_b.text()) ** float(self.coeff_b.text
960 ())) * (
961                             float(self.equilibrium_constant.text())) / (
962                             float(self.conc_d.text()) ** float(self.
963 coeff_d.text()))))
964                 coeff = int(log(coeff, float(self.conc_c.text())))
965             except OverflowError:
966                 print("Overflow Error, inputted numbers too large")
967                 show_dialog("Overflow Error, inputted numbers are too large!")
968         ")
969         return
970         self.calculated_value = to_find
971         self.update_gui(to_find, coeff)
972         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.
2025-09-26 11:01:23
text())) * (
965                     float(self.conc_b.text()) ** float(self.coeff_b.text
2025-09-26 11:01:23
966                     ())) * (
967                         float(self.equilibrium_constant.text())) / (
2025-09-26 11:01:23
968                             float(self.conc_c.text()) ** float(self.
2025-09-26 11:01:23
coeff_c.text()))
969                 coeff = int(log(coeff, float(self.conc_d.text())))
970             except OverflowError:
971                 print("Overflow Error, inputted numbers too large")
972                 show_dialog("Overflow Error, inputted numbers are too large!")
973         ")
974         return
975         self.calculated_value = to_find
976         self.update_gui(to_find, coeff)
977         return
978
979     def update_gui(self, empty_input, value):
980         if empty_input is self.calculated_value:
981             empty_input.setText(str(value))
982
983
984 class GibbsFreeEnergyCalculator(QWidget):
985     def __init__(self):
986         super(QWidget, self).__init__()
987
988         self.layout = QGridLayout()
989
990         self.temperature_conversions = {
991             "C": 273.15,
992             "K": 0.0,
993         }
994
995         self.general_energy_conversions = {
996             "kJ": 1.0,
997             "J": 0.001
998         }
999
1000         self.gibbs_free_energy_label = QLabel("Gibbs Free Energy (delta G):")
1001
1002     )
1003         self.enthalpy_change_label = QLabel("Enthalpy Change (delta H):")
1004         self.temp_label = QLabel("Temperature:")
1005         self.entropy_change_label = QLabel("Entropy Change (delta S):")
1006
1007         self.gibbs_free_energy_input = QLineEdit()
1008         self.enthalpy_change_input = QLineEdit()
1009         self.temp_input = QLineEdit()
1010         self.entropy_change_input = QLineEdit()
1011
1012         self.input_list = [self.gibbs_free_energy_input, self.
2025-09-26 11:01:23
entropy_change_input, self.temp_input,
2025-09-26 11:01:23
self.enthalpy_change_input]
1013
1014         self.gibbs_free_energy_unit_dropdown = QComboBox()
1015         self.gibbs_free_energy_unit_dropdown.addItem("kJ")
1016         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]) * (

```

```

1116         float(self.entropy_change_input.text()) * self.
general_energy_conversions[entropy_unit]))
1117         enthalpy_change = enthalpy_change * self.
general_energy_conversions[enthalpy_unit]
1118         return enthalpy_change
1119     except ValueError:
1120         print("Value Error")
1121         show_dialog("Value Error!")
1122         return
1123
1124     def calculate_temperature(self, free_energy_unit, enthalpy_unit,
temp_unit, entropy_unit):
1125         try:
1126             temperature = ((float(self.enthalpy_change_input.text()) * self.
general_energy_conversions[
1127                 enthalpy_unit]) - (float(self.gibbs_free_energy_input.text()
) * self.general_energy_conversions[
1128                 free_energy_unit])) / (
1129                 float(self.entropy_change_input.text()) *
self.general_energy_conversions[
1130                 entropy_unit])
1131             temperature = temperature + self.temperature_conversions[
temp_unit]
1132             return temperature
1133         except ValueError:
1134             print("Value Error")
1135             show_dialog("Value Error!")
1136             return
1137
1138     def calculate_entropy_change(self, free_energy_unit, enthalpy_unit,
temp_unit, entropy_unit):
1139         try:
1140             entropy_change = ((float(self.enthalpy_change_input.text()) *
self.general_energies[
1141                 enthalpy_unit]) - (float(self.gibbs_free_energy_input.text()
) * self.general_energy_conversions[
1142                 free_energy_unit])) / (
1143                 float(self.temp_input.text()) + self.
temperature_conversions[temp_unit])
1144             entropy_change = entropy_change / self.
general_energy_conversions[entropy_unit]
1145             return entropy_change
1146         except ValueError:
1147             print("Value Error")
1148             show_dialog("Value Error!")
1149             return
1150
1151
1152 class SpecificHeatCalculator(QWidget):
1153     def __init__(self):
1154         super(QWidget, self).__init__()
1155
1156         self.layout = QGridLayout()
1157
1158         self.mass_conversions = {
1159             "mg": 0.001,
1160             "g": 1,
1161             "kg": 1000,
1162             "t": 1000000

```

```

1163     }
1164
1165     self.temperature_conversions = {
1166         "C": 273.15,
1167         "K": 0.0,
1168     }
1169
1170     self.energy_conversions = {
1171         "kJ": 0.001,
1172         "J": 1.0
1173     }
1174
1175     self.energy_label = QLabel("Heat Energy: ")
1176     self.mass_label = QLabel("Mass: ")
1177     self.heat_capacity_label = QLabel("Specific Heat Capacity: ")
1178     self.temperature_change_label = QLabel("Temperature Change: ")
1179
1180     self.energy_input = QLineEdit()
1181     self.mass_input = QLineEdit()
1182     self.heat_capacity_input = QLineEdit()
1183     self.temperature_change_input = QLineEdit()
1184
1185     self.input_list = [self.energy_input, self.mass_input, self.
heat_capacity_input, self.temperature_change_input]
1186
1187     self.energy_unit_dropdown = QComboBox()
1188     self.energy_unit_dropdown.addItem("J")
1189     self.energy_unit_dropdown.addItem("kJ")
1190
1191     self.energy_unit_dropdown.setCurrentIndex(0)
1192
1193     self.mass_unit_dropdown = QComboBox()
1194     self.mass_unit_dropdown.addItem("mg")
1195     self.mass_unit_dropdown.addItem("g")
1196     self.mass_unit_dropdown.addItem("kg")
1197
1198     self.mass_unit_dropdown.setCurrentIndex(1)
1199
1200     self.temp_unit_dropdown = QComboBox()
1201     self.temp_unit_dropdown.addItem("C")
1202     self.temp_unit_dropdown.addItem("K")
1203
1204     self.temp_unit_dropdown.setCurrentIndex(1)
1205
1206     self.calculate_button = QPushButton("Calculate")
1207
1208     self.calculate_button.clicked.connect(self.calculate)
1209
1210     self.layout.addWidget(self.energy_label, 0, 0)
1211     self.layout.addWidget(self.energy_input, 0, 1)
1212     self.layout.addWidget(self.energy_unit_dropdown, 0, 2)
1213
1214     self.layout.addWidget(self.mass_label, 1, 0)
1215     self.layout.addWidget(self.mass_input, 1, 1)
1216     self.layout.addWidget(self.mass_unit_dropdown, 1, 2)
1217
1218     self.layout.addWidget(self.heat_capacity_label, 2, 0)
1219     self.layout.addWidget(self.heat_capacity_input, 2, 1)
1220

```

```

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:

```

```

1272         q = (float(self.mass_input.text()) * mass_unit) * float(self.
heat_capacity_input.text()) * (
1273             float(self.temperature_change_input.text()) + temp_unit)
1274         q = q * energy_unit
1275
1276         return q
1277     except ValueError:
1278         print("ValueError")
1279         return
1280
1281     def __calculate_mass(self, energy_unit, mass_unit, temp_unit):
1282         try:
1283             mass = (float(self.energy_input.text()) * energy_unit) / (float(
self.heat_capacity_input.text()) * (
1284                 float(self.temperature_change_input.text()) + temp_unit)
)
1285             mass = mass * mass_unit
1286
1287             return mass
1288         except ValueError:
1289             print("ValueError")
1290             return
1291
1292     def __calculate_heat_capacity(self, energy_unit, mass_unit, temp_unit):
1293         try:
1294             heat_capacity = (float(self.energy_input.text()) * energy_unit)
/ (
1295                 (float(self.mass_input.text()) * mass_unit) * (
1296                     float(self.temperature_change_input.text()) + temp_unit)
)
1297
1298             return heat_capacity
1299         except ValueError:
1300             print("ValueError")
1301             return
1302
1303     def __calculate_temp_change(self, energy_unit, mass_unit, temp_unit):
1304         try:
1305             temp_change = (float(self.energy_input.text()) * energy_unit) /
(
1306                 (float(self.mass_input.text()) * mass_unit) * (
1307                     float(self.heat_capacity_input.text()))
)
1308             temp_change = temp_change + temp_unit
1309
1310             return temp_change
1311         except ValueError:
1312             print("ValueError")
1313             return
1314
1315
1316     class RateCalculator(QWidget):
1317         def __init__(self):
1318             super(QWidget, self).__init__()
1319
1320             self.order_conversion = {
1321                 "First": 1,
1322                 "Second": 2
1323             }
1324

```

```

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
(unimolecular, bimolecular, trimolecular), and then calls the
1480         update_input() method to display the calculated

```



```

1481         result.
1482         """
1483
1484         rate = float(self.rate_constant_input.text()) * (float(self.
1485         unimol_conc_input.text()) ** unimol_order)
1486
1487         if self.step_dropdown.currentIndex() == 1:
1488             rate = rate * (float(self.bimol_conc_input.text()) **
1489             bimol_order)
1490         elif self.step_dropdown.currentIndex() == 2:
1491             rate = rate * (float(self.bimol_conc_input.text()) **
1492             bimol_order) * (
1493                 float(self.trimol_conc_input.text()) ** trimol_order)
1494
1495         self.update_input(rate)
1496
1497     def calculate_rate_constant(self, unimol_order: int, bimol_order: int,
1498     trimol_order: int) -> None:
1499         """
1500         Calculates the rate constant based on the elementary step
1501         (unimolecular, bimolecular, trimolecular), and calls the
1502         update_input method to display the calculated result.
1503         """
1504
1505         total_conc = (float(self.unimol_conc_input) ** unimol_order)
1506
1507         if self.step_dropdown.currentIndex() == 1:
1508             total_conc = total_conc * (float(self.bimol_conc_input.text())
1509             ** bimol_order)
1510         elif self.step_dropdown.currentIndex() == 2:
1511             total_conc = total_conc * (float(self.bimol_conc_input.text())
1512             ** bimol_order) * (
1513                 float(self.trimol_conc_input.text()) ** trimol_order)
1514
1515         rate_constant = float(self.rate_input) / total_conc
1516
1517         self.update_input(rate_constant)
1518
1519     def calculate_concentration(self, to_find: QLineEdit, unimol_order: int,
1520     bimol_order: int,
1521                                trimol_order: int) -> None:
1522         """
1523         Calculates the concentration of the given empty concentration input,
1524         based on the elementary step
1525         (unimolecular, bimolecular, trimolecular), and then calls the
1526         update_input() method to display the calculated
1527         result.
1528         """
1529
1530         concentration = float(self.rate_input.text()) / float(self.
1531         rate_constant_input.text())
1532
1533         if to_find is self.unimol_conc_input:
1534             if self.step_dropdown.currentIndex() == 1:
1535                 concentration = concentration / (float(self.bimol_conc_input
1536                 .text()) ** bimol_order)
1537             elif self.step_dropdown.currentIndex() == 2:
1538                 concentration = concentration / ((float(self.
1539                 bimol_conc_input.text()) ** bimol_order) * (

```

```

1527         float(self.trimol_conc_input.text()) ** trimol_order
1528     ))
1529     concentration = concentration ** (1 / unimol_order)
1530     elif to_find is self.bimol_conc_input:
1531         if self.step_dropdown.currentIndex() == 1:
1532             concentration = concentration / (float(self.
1533 unimol_conc_input.text()) ** unimol_order)
1534         elif self.step_dropdown.currentIndex() == 2:
1535             concentration = concentration / ((float(self.
1536 unimol_conc_input.text()) ** unimol_order) * (
1537         float(self.trimol_conc_input.text()) ** trimol_order
1538     ))
1539     concentration = concentration ** (1 / bimol_order)
1540     elif to_find is self.trimol_conc_input:
1541         concentration = concentration / (
1542             (float(self.unimol_conc_input.text()) ** unimol_order) *
1543             (
1544                 float(self.bimol_conc_input.text()) ** bimol_order))
1545         concentration = concentration ** (1 / trimol_order)
1546     else:
1547         return
1548     self.update_input(concentration)
1549
1550 def update_input(self, result: float) -> None:
1551     """
1552     Uses the find_empty_input() function to find the empty input, and
1553     then updates it using the result parameter.
1554     """
1555     find_empty_input(self.input_list.copy()).setText(str(result))
1556
1557 def update_total_order(self, order: float) -> None:
1558     """
1559     Updates the total order input with the order parameter.
1560     """
1561     self.total_order_input.setText(str(order))

```

Listing 9: chem\_calculator.py File Program Code

## A.4 ChemBalancer File

```
1 from PyQt6.QtWidgets import QWidget, QPushButton, QLineEdit, QLabel,
   QVBoxLayout, QFrame, QHBoxLayout, QMessageBox
2 from PyQt6.QtCore import Qt
3
4 import re
5 from sympy import Matrix, lcm
6
7
8 def show_dialog(message):
9     dlg = QMessageBox()
10    dlg.setWindowTitle("Invalid Input!")
11    dlg.setText(f"Invalid user input!\n {message}")
12    dlg.setIcon(QMessageBox.Icon.Critical)
13    button = dlg.exec()
14
15    if button == QMessageBox.StandardButton.Ok:
16        print("OK!")
17
18
19 class ChemBalancer(QWidget):
20     """
21     This module is responsible for balancing chemical equations.
22     It parses user-provided chemical equations, identifies reactants and
23     products,
24     and calculates the coefficients to achieve a balanced equation.
25     The class utilizes SymPy for symbolic mathematics to find the null space
26     and perform matrix operations,
27     ensuring accurate and balanced chemical equations.
28     """
29
30     def __init__(self):
31         super(QWidget, self).__init__()
32         self.balancer_layout = QVBoxLayout()
33
34         self.unbalanced_frame = QFrame()
35         self.unbalanced_frame.setFrameShape(QFrame.Shape.Panel)
36
37         self.balanced_frame = QFrame()
38         self.balanced_frame.setFrameShape(QFrame.Shape.Panel)
39
40         self.unbalanced_label = QLabel("Unbalanced Equation:")
41         self.equation_input = QLineEdit()
42         self.balanced_label = QLabel("Balanced Equation:")
43         self.balanced_output = QLineEdit()
44         self.balanced_output.setReadOnly(True)
45         self.balance_button = QPushButton("Balance")
46
47         self.equation_input.setMinimumWidth(800)
48         self.balanced_output.setMinimumWidth(800)
49         self.balance_button.setFixedWidth(300)
50
51         self.unbalanced_frame_layout = QHBoxLayout()
52         self.unbalanced_frame.setLayout(self.unbalanced_frame_layout)
53         self.unbalanced_frame_layout.addWidget(self.unbalanced_label)
54         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
and splits it up into separate reactants and products.
100         """
101
102         # Strip equation from any whitespaces
103         try:
104             self.stripped_equation = "".join(self.equation_input.text().
split())
105         except IndexError:
106             return None
107

```

```

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

```

```

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

```

```

217         if element not in self.element_list:
218             self.element_list.append(element)
219             for i in range(len(self.element_matrix)):
220                 self.element_matrix[i].append(0)
221             print(self.element_matrix)
222     except Exception:
223         raise
224
225     column = self.element_list.index(element)
226     self.element_matrix[index][column] += count * side
227     print(self.element_list)
228     print(self.element_matrix)
229
230     def run_balancer(self):
231         f"""
232         This the core function of the {ChemBalancer} class,
233         responsible for balancing chemical equations.
234         It parses the user-provided equation, deciphers compounds,
235         constructs a matrix, finds the null space for balancing coefficients
236         ,
237         and computes the balanced equation.
238         This function leverages SymPy for mathematical operations, ensuring
239         accurate chemical equation balancing.
240         """
241
242         # Clear variables, in case the program was run before
243         self.clear_variables()
244
245         try:
246             self.split_equation()
247         except Exception:
248             show_dialog("")
249             return
250
251         for i in range(len(self.reactants)):
252             try:
253                 self.find_reagents(self.reactants[i], i, 1)
254             except Exception:
255                 show_dialog("")
256                 return
257
258         for i in range(len(self.products)):
259             try:
260                 self.find_reagents(self.products[i], i + len(self.reactants)
261 , -1)
262             except Exception:
263                 show_dialog("")
264                 return
265
266         self.element_matrix = Matrix(self.element_matrix)
267         self.element_matrix = self.element_matrix.transpose()
268
269         try:
270             num = self.element_matrix.nullspace()[0]
271         except IndexError:
272             show_dialog("Please provide a balanceable equation in the format
273 X + Y = XY + Z")
274             return
275
276         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

```
1 from PyQt6.QtCore import Qt, QPointF, pyqtSignal
2 from PyQt6.QtCore import QPoint
3 from PyQt6.QtWidgets import QWidget, QGridLayout, QPushButton, QLabel,
4   QFileDialog, QMessageBox
5 from PyQt6.QtGui import QPixmap, QPainter, QPen, QColor, QFont, QBrush
6
7 import chem_editor_logic
8
9 import math
10 import json
11
12 def show_dialog(message):
13     dlg = QMessageBox()
14     dlg.setWindowTitle("Invalid Action!")
15     dlg.setText(f"Invalid user action!\n {message}")
16     dlg.setIcon(QMessageBox.Icon.Critical)
17     button = dlg.exec()
18
19     if button == QMessageBox.StandardButton.Ok:
20         print("OK!")
21
22
23 class ChemEditor(QWidget):
24     def __init__(self):
25         super().__init__()
26
27         self.editor_layout = QGridLayout()
28         self.setLayout(self.editor_layout)
29
30         self.draw_action_button = QPushButton("Draw")
31         self.bond_action_button = QPushButton("Bond")
32
33         self.remove_button = QPushButton("Remove")
34         self.reset_button = QPushButton("Reset")
35
36         self.save_button = QPushButton("Save")
37
38         self.single_bond_button = QPushButton("Single")
39         self.double_bond_button = QPushButton("Double")
40         self.triple_bond_button = QPushButton("Triple")
41
42         self.editor_layout.addWidget(self.draw_action_button, 0, 10)
43         self.editor_layout.addWidget(self.bond_action_button, 0, 11)
44         self.editor_layout.addWidget(self.remove_button, 0, 12, 1, 2)
45         self.editor_layout.addWidget(self.reset_button, 0, 14)
46         self.editor_layout.addWidget(self.save_button, 0, 15)
47         self.editor_layout.addWidget(self.single_bond_button, 0, 22)
48         self.editor_layout.addWidget(self.double_bond_button, 0, 23)
49         self.editor_layout.addWidget(self.triple_bond_button, 0, 24)
50
51         self.bond_action_button.clicked.connect(self.choose_bond_action)
52         self.draw_action_button.clicked.connect(self.choose_draw_action)
53         self.remove_button.clicked.connect(self.remove_action)
54         self.reset_button.clicked.connect(self.reset_action)
55         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 *.
153             jpeg);;All Files(*.*) ")
154
155         # If file path is blank return back
156         if filePath == "":
157             return
158
159         # Save canvas at desired path
160         self.pixmap.save(filePath)
161
162     def set_element(self, new_element: dict) -> None:
163         self.element = new_element
164         self.update()
165
166     def set_action_type(self, action: str) -> None:
167         self.action_type = action
168         self.update()
169
170     def set_bond_order(self, order: int) -> None:
171         self.bond_order = order
172         self.update()

```

```

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     initPainter = QPainter(self)
189     initPainter.drawPixmap(0, 0, self.pixmap)
190
191     # Initialise pixmap painter
192     pixPainter = QPainter(self.pixmap)
193     font = QFont("Arial", 16)
194     pixPainter.setFont(font)
195
196     pen = QPen(QColor(0, 0, 0))
197     pen.setWidth(2)
198     pixPainter.setPen(pen)
199
200     # Draw every atom in self.atoms list
201     for atom in self.atoms:
202         self.draw_atom(atom.x_coors, atom.y_coors, atom.symbol,
203             pixPainter, pen, False)
204
205     # For every bond of atom, draw the bond, and redraw the atoms
206     again
207     for bond in atom.bonds:
208         if bond.order == 2:
209             self.draw_double_bond(bond.atoms[0].x_coors, bond.atoms
210                 [0].y_coors, bond.atoms[1].x_coors,
211                 bond.atoms[1].y_coors,
212                 pixPainter, pen, True)
213         elif bond.order == 3:
214             self.draw_triple_bond(bond.atoms[0].x_coors, bond.atoms
215                 [0].y_coors, bond.atoms[1].x_coors,
216                 bond.atoms[1].y_coors,
217                 pixPainter, pen, True)
218         else:
219             self.draw_single_bond(bond.atoms[0].x_coors, bond.atoms
220                 [0].y_coors, bond.atoms[1].x_coors,
221                 bond.atoms[1].y_coors,
222                 pixPainter, pen, True)
223
224         self.draw_atom_circle(bond.atoms[1].x_coors, bond.atoms[1].
225             y_coors, bond.atoms[0].x_coors,
226             bond.atoms[0].y_coors, pixPainter,
227             pen)
228
229         pen.setStyle(Qt.PenStyle.SolidLine)
230         pixPainter.setPen(pen)
231         self.draw_atom(bond.atoms[1].x_coors, bond.atoms[1].
232             y_coors, bond.atoms[1].symbol, pixPainter, pen,

```

```

221         False)
222         self.draw_atom(bond.atoms[0].x_coords, bond.atoms[0].
223 y_coords, bond.atoms[0].symbol, pixPainter, pen,
224 False)
225
226     # Check for selected atom and draw potential positions
227     if self.selected:
228         print("selected")
229         if self.action_type != "bond":
230             try:
231                 # Calculate possible positions for new atoms in 360
232 degrees around the selected atom
233                 if self.selected_atom is not None:
234                     potential_positions = self.calc_potential_positions(
235 self.selected_atom)
236                     for pos in potential_positions:
237                         # If atoms at position don't overlap, draw the
238 potential bonds and atoms in different colour
239                         if not self.check_atom_overlap(pos[0], pos[1]):
240                             if self.bond_order == 2:
241                                 self.draw_double_bond(self.selected_atom
242 .x_coords, self.selected_atom.y_coords,
243                                                         pos[0],
244                                                         pos[1],
245 pixPainter, pen, False)
246                             elif self.bond_order == 3:
247                                 self.draw_triple_bond(self.selected_atom
248 .x_coords, self.selected_atom.y_coords,
249                                                         pos[0], pos[1],
250 pixPainter, pen, False)
251                             else:
252                                 self.draw_single_bond(self.selected_atom
253 .x_coords, self.selected_atom.y_coords,
254                                                         pos[0], pos[1],
255 pixPainter, pen, False)
256                                 self.draw_atom_circle(pos[0], pos[1], self.
257 selected_atom.x_coords,
258                                                         self.selected_atom.
259 y_coords, pixPainter, pen)
260                                 pen.setStyle(Qt.PenStyle.SolidLine)
261                                 pixPainter.setPen(pen)
262
263                                 # Use of self.element["symbol"], as the
264 whole elements json data is stored in self.element
265                                 # E.g. {'symbol': 'C',
266                                 # 'atomic_number': 6,
267                                 # 'group': 14,
268                                 # 'period': 2,
269                                 # 'outer_electrons': 4,
270                                 # 'full_shell': 8}
271
272                                 self.draw_atom(pos[0], pos[1], self.element[
273 "symbol"], pixPainter, pen, True)
274                                 self.draw_atom(self.selected_atom.x_coords,
275 self.selected_atom.y_coords,
276                                                         self.selected_atom.symbol,
277 pixPainter, pen, False)
278                                 initPainter.drawPixmap(0, 0, self.pixmap)
279             except AttributeError:

```

```

264         return
265     initPainter.drawPixmap(0, 0, self.pixmap)
266     initPainter.end()
267     pixPainter.end()
268
269     # Function to draw potential positions for atoms
270     @staticmethod
271     def calc_potential_positions(atom: chem_editor_logic.Atom) -> list[tuple
272     [int, int]]:
273         """
274         :param atom:
275         :return: list of tuples containing x and y coordinates of potential
276         positions for new atoms
277         """
278         x = atom.x_coords
279         y = atom.y_coords
280         distance = 40
281
282         # Calculate coordinates for angles in steps of 45 degrees from 0 to
283         360
284         coordinates_list = []
285         for angle_degrees in range(0, 360, 45):
286             angle_radians = math.radians(angle_degrees)
287             new_x = x + distance * math.cos(angle_radians)
288             new_y = y + distance * math.sin(angle_radians)
289             coordinates_list.append((int(new_x), int(new_y)))
290
291         print(x, y)
292         print(coordinates_list)
293         return coordinates_list
294
295     def check_atom_overlap(self, pos_x: int, pos_y: int) -> bool:
296         atom_radius = Canvas.ATOM_RADIUS
297         for atom in self.atoms:
298             if (
299                 atom.x_coords - atom_radius <= pos_x <= atom.x_coords +
300                 atom_radius and
301                 atom.y_coords - atom_radius <= pos_y <= atom.y_coords +
302                 atom_radius
303             ):
304                 return True
305
306     def draw_single_bond(self, atom1_x: int, atom1_y: int, atom2_x: int,
307     atom2_y: int, painter: QPainter, pen: QPen,
308     actual_bond: bool = True) -> None:
309         if not actual_bond:
310             # Set colour red
311             pen.setColor(QColor(255, 0, 0))
312             painter.setPen(pen)
313
314             painter.drawLine(QPoint(atom1_x, atom1_y), QPoint(atom2_x, atom2_y))
315
316     def draw_double_bond(self, atom1_x: int, atom1_y: int, atom2_x: int,
317     atom2_y: int, painter: QPainter, pen: QPen,
318     actual_bond: bool = True) -> None:
319         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,
374                          atom2_y: int, painter: QPainter,
375                          pen: QPen) -> None:
376         """
377         This function draws a circle in the same colour as the background
378         colour to prevent bonds from overlapping with
379         atom. This function must be called after drawing bonds in order to
380         overwrite them, and before drawing atoms,
381         as this would make the atoms invisible.
382         """
383
384         # Set the brush color to match the background color
385         background_color = QColor(200, 200, 200)
386         brush = QBrush(background_color)
387         painter.setBrush(brush)
388         pen.setStyle(Qt.PenStyle.NoPen)
389         painter.setPen(pen)
390
391         # Draw a filled circle
392         circle_center = QPointF(atom2_x, atom2_y)
393         circle_radius = Canvas.ATOM_RADIUS
394         painter.drawEllipse(circle_center, circle_radius, circle_radius)
395
396         circle_center = QPointF(atom1_x, atom1_y)
397         painter.drawEllipse(circle_center, circle_radius, circle_radius)
398
399     def draw_atom(self, atom_x: int, atom_y: int, symbol: str, painter:
400                  QPainter, pen: QPen,
401                  potential: bool = False) -> None:
402
403         if potential:
404             pen.setColor(QColor(100, 100, 100))
405             painter.setPen(pen)
406
407         # Calculate position to draw atom in the center of the "atom circle"
408         letter_width = painter.fontMetrics().horizontalAdvance(symbol)
409         letter_height = painter.fontMetrics().height()
410         letter_x = atom_x - letter_width / 2
411         letter_y = atom_y + letter_height / 4
412         painter.drawText(int(letter_x), int(letter_y), symbol)
413
414     def check_clicked_on_atom(self, pos_x: int, pos_y: int) -> bool:
415         atom_radius = Canvas.ATOM_RADIUS
416         for atom in self.atoms:
417             if (
418                 atom.x_coords - atom_radius <= pos_x <= atom.x_coords +
419                 atom_radius and
420                 atom.y_coords - atom_radius <= pos_y <= atom.y_coords +
421                 atom_radius
422             ):
423                 self.selected = True
424                 self.selected_atom = atom
425                 self.update()
426                 return True
427
428     def remove_atom(self, pos_x: int, pos_y: int) -> None:
429         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

```
1 from PyQt6.QtWidgets import QMessageBox
2
3
4 def show_dialog(message):
5     dlg = QMessageBox()
6     dlg.setWindowTitle("Invalid Action!")
7     dlg.setText(f"Invalid user action!\n {message}")
8     dlg.setIcon(QMessageBox.Icon.Critical)
9     button = dlg.exec()
10
11     if button == QMessageBox.StandardButton.Ok:
12         print("OK!")
13
14
15 class Atom:
16     def __init__(self, element_data: dict, coordinates: list[int, int]):
17         self.symbol: str = element_data["symbol"]
18         self.outer_electrons: int = element_data["outer_electrons"]
19         self.x_coords: int = coordinates[0]
20         self.y_coords: int = coordinates[1]
21         self.full_shell: int = element_data["full_shell"]
22         self.bonds: list[Bond] = []
23         self.extra_electrons: int = 0
24         self.overall_electrons: int = (self.outer_electrons + self.
25         extra_electrons)
26
27     def __eq__(self, other) -> bool:
28         return self.symbol == other.symbol and [self.x_coords, self.y_coords
29 ] == [other.x_coords, other.y_coords]
30
31
32     def __add_outer_electrons(self, num: int) -> None:
33         self.outer_electrons += num
34
35     def __remove_outer_electrons(self, num: int) -> None:
36         self.outer_electrons -= num
37
38
39     def check_is_bond_possible(self, bonding_atom, order: int = 1) -> bool:
40         if (self.overall_electrons + order) > self.full_shell:
41             print("Bonding unavailable, shell is full.")
42             return False
43         elif (bonding_atom.overall_electrons + order) > bonding_atom.
44         full_shell:
45             print("Bonding unavailable, bonding atoms shell is full.")
46             return False
47         else:
48             return True
49
50     def bond(self, bonding_atom, order: int = 1) -> None:
51         if not self.check_is_bond_possible(bonding_atom, order):
52             show_dialog("Bonding unavailable, shell is full.")
53             return
54         new_bond = Bond(self, bonding_atom, order)
55         self.bonds.append(new_bond)
56         bonding_atom.bonds.append(new_bond)
57
58         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
11 QTableWidget::item {
12     color: darkgray;
13     border: 1.5px solid #ccc;
14 }
```

## A.8 Elements.json file

```
1 {
2   "Hydrogen": {
3     "symbol": "H",
4     "atomic_number": 1,
5     "group": 1,
6     "period": 1,
7     "outer_electrons": 1,
8     "full_shell": 2
9   },
10  "Helium": {
11    "symbol": "He",
12    "atomic_number": 2,
13    "group": 18,
14    "period": 1,
15    "outer_electrons": 2,
16    "full_shell": 2
17  },
18  "Lithium": {
19    "symbol": "Li",
20    "atomic_number": 3,
21    "group": 1,
22    "period": 2,
23    "outer_electrons": 1,
24    "full_shell": 8
25  },
26  "Beryllium": {
27    "symbol": "Be",
28    "atomic_number": 4,
29    "group": 2,
30    "period": 2,
31    "outer_electrons": 2,
32    "full_shell": 8
33  },
34  "Boron": {
35    "symbol": "B",
36    "atomic_number": 5,
37    "group": 13,
38    "period": 2,
39    "outer_electrons": 3,
40    "full_shell": 8
41  },
42  "Carbon": {
43    "symbol": "C",
44    "atomic_number": 6,
45    "group": 14,
46    "period": 2,
47    "outer_electrons": 4,
48    "full_shell": 8
49  },
50  "Nitrogen": {
51    "symbol": "N",
52    "atomic_number": 7,
53    "group": 15,
54    "period": 2,
55    "outer_electrons": 5,
56    "full_shell": 8
```

```

57 | },
58 | "Oxygen": {
59 |   "symbol": "O",
60 |   "atomic_number": 8,
61 |   "group": 16,
62 |   "period": 2,
63 |   "outer_electrons": 6,
64 |   "full_shell": 8
65 | },
66 | "Fluorine": {
67 |   "symbol": "F",
68 |   "atomic_number": 9,
69 |   "group": 17,
70 |   "period": 2,
71 |   "outer_electrons": 7,
72 |   "full_shell": 8
73 | },
74 | "Neon": {
75 |   "symbol": "Ne",
76 |   "atomic_number": 10,
77 |   "group": 18,
78 |   "period": 2,
79 |   "outer_electrons": 8,
80 |   "full_shell": 8
81 | },
82 | "Sodium": {
83 |   "symbol": "Na",
84 |   "atomic_number": 11,
85 |   "group": 1,
86 |   "period": 3,
87 |   "outer_electrons": 1,
88 |   "full_shell": 8
89 | },
90 | "Magnesium": {
91 |   "symbol": "Mg",
92 |   "atomic_number": 12,
93 |   "group": 2,
94 |   "period": 3,
95 |   "outer_electrons": 2,
96 |   "full_shell": 8
97 | },
98 | "Aluminium": {
99 |   "symbol": "Al",
100 |   "atomic_number": 13,
101 |   "group": 13,
102 |   "period": 3,
103 |   "outer_electrons": 3,
104 |   "full_shell": 8
105 | },
106 | "Silicon": {
107 |   "symbol": "Si",
108 |   "atomic_number": 14,
109 |   "group": 14,
110 |   "period": 3,
111 |   "outer_electrons": 4,
112 |   "full_shell": 8
113 | },
114 | "Phosphorus": {
115 |   "symbol": "P",

```



```

116     "atomic_number": 15,
117     "group": 15,
118     "period": 3,
119     "outer_electrons": 5,
120     "full_shell": 8
121 },
122 "Sulfur": {
123     "symbol": "S",
124     "atomic_number": 16,
125     "group": 16,
126     "period": 3,
127     "outer_electrons": 6,
128     "full_shell": 8
129 },
130 "Chlorine": {
131     "symbol": "Cl",
132     "atomic_number": 17,
133     "group": 17,
134     "period": 3,
135     "outer_electrons": 7,
136     "full_shell": 8
137 },
138 "Argon": {
139     "symbol": "Ar",
140     "atomic_number": 18,
141     "group": 18,
142     "period": 3,
143     "outer_electrons": 8,
144     "full_shell": 8
145 },
146 "Potassium": {
147     "symbol": "K",
148     "atomic_number": 19,
149     "group": 1,
150     "period": 4,
151     "outer_electrons": 1,
152     "full_shell": 18
153 },
154 "Calcium": {
155     "symbol": "Ca",
156     "atomic_number": 20,
157     "group": 2,
158     "period": 4,
159     "outer_electrons": 2,
160     "full_shell": 18
161 },
162 "Scandium": {
163     "symbol": "Sc",
164     "atomic_number": 21,
165     "group": 3,
166     "period": 4,
167     "outer_electrons": 3,
168     "full_shell": 18
169 },
170 "Titanium": {
171     "symbol": "Ti",
172     "atomic_number": 22,
173     "group": 4,
174     "period": 4,

```

```

175     "outer_electrons": 4,
176     "full_shell": 18
177 },
178 "Vanadium": {
179     "symbol": "V",
180     "atomic_number": 23,
181     "group": 5,
182     "period": 4,
183     "outer_electrons": 5,
184     "full_shell": 18
185 },
186 "Chromium": {
187     "symbol": "Cr",
188     "atomic_number": 24,
189     "group": 6,
190     "period": 4,
191     "outer_electrons": 6,
192     "full_shell": 18
193 },
194 "Manganese": {
195     "symbol": "Mn",
196     "atomic_number": 25,
197     "group": 7,
198     "period": 4,
199     "outer_electrons": 7,
200     "full_shell": 18
201 },
202 "Iron": {
203     "symbol": "Fe",
204     "atomic_number": 26,
205     "group": 8,
206     "period": 4,
207     "outer_electrons": 8,
208     "full_shell": 18
209 },
210 "Cobalt": {
211     "symbol": "Co",
212     "atomic_number": 27,
213     "group": 9,
214     "period": 4,
215     "outer_electrons": 9,
216     "full_shell": 18
217 },
218 "Nickel": {
219     "symbol": "Ni",
220     "atomic_number": 28,
221     "group": 10,
222     "period": 4,
223     "outer_electrons": 10,
224     "full_shell": 18
225 },
226 "Copper": {
227     "symbol": "Cu",
228     "atomic_number": 29,
229     "group": 11,
230     "period": 4,
231     "outer_electrons": 11,
232     "full_shell": 18
233 },

```

```

234 | "Zinc": {
235 |     "symbol": "Zn",
236 |     "atomic_number": 30,
237 |     "group": 12,
238 |     "period": 4,
239 |     "outer_electrons": 12,
240 |     "full_shell": 18
241 | },
242 | "Gallium": {
243 |     "symbol": "Ga",
244 |     "atomic_number": 31,
245 |     "group": 13,
246 |     "period": 4,
247 |     "outer_electrons": 13,
248 |     "full_shell": 18
249 | },
250 | "Germanium": {
251 |     "symbol": "Ge",
252 |     "atomic_number": 32,
253 |     "group": 14,
254 |     "period": 4,
255 |     "outer_electrons": 14,
256 |     "full_shell": 18
257 | },
258 | "Arsenic": {
259 |     "symbol": "As",
260 |     "atomic_number": 33,
261 |     "group": 15,
262 |     "period": 4,
263 |     "outer_electrons": 15,
264 |     "full_shell": 18
265 | },
266 | "Selenium": {
267 |     "symbol": "Se",
268 |     "atomic_number": 34,
269 |     "group": 16,
270 |     "period": 4,
271 |     "outer_electrons": 16,
272 |     "full_shell": 18
273 | },
274 | "Bromine": {
275 |     "symbol": "Br",
276 |     "atomic_number": 35,
277 |     "group": 17,
278 |     "period": 4,
279 |     "outer_electrons": 17,
280 |     "full_shell": 18
281 | },
282 | "Krypton": {
283 |     "symbol": "Kr",
284 |     "atomic_number": 36,
285 |     "group": 18,
286 |     "period": 4,
287 |     "outer_electrons": 18,
288 |     "full_shell": 18
289 | },
290 | "Rubidium": {
291 |     "symbol": "Rb",
292 |     "atomic_number": 37,

```

```

293     "group": 1,
294     "period": 5,
295     "outer_electrons": 1,
296     "full_shell": 18
297 },
298 "Strontium": {
299     "symbol": "Sr",
300     "atomic_number": 38,
301     "group": 2,
302     "period": 5,
303     "outer_electrons": 2,
304     "full_shell": 18
305 },
306 "Yttrium": {
307     "symbol": "Y",
308     "atomic_number": 39,
309     "group": 3,
310     "period": 5,
311     "outer_electrons": 3,
312     "full_shell": 18
313 },
314 "Zirconium": {
315     "symbol": "Zr",
316     "atomic_number": 40,
317     "group": 4,
318     "period": 5,
319     "outer_electrons": 4,
320     "full_shell": 18
321 },
322 "Niobium": {
323     "symbol": "Nb",
324     "atomic_number": 41,
325     "group": 5,
326     "period": 5,
327     "outer_electrons": 5,
328     "full_shell": 18
329 },
330 "Molybdenum": {
331     "symbol": "Mo",
332     "atomic_number": 42,
333     "group": 6,
334     "period": 5,
335     "outer_electrons": 6,
336     "full_shell": 18
337 },
338 "Technetium": {
339     "symbol": "Tc",
340     "atomic_number": 43,
341     "group": 7,
342     "period": 5,
343     "outer_electrons": 7,
344     "full_shell": 18
345 },
346 "Ruthenium": {
347     "symbol": "Ru",
348     "atomic_number": 44,
349     "group": 8,
350     "period": 5,
351     "outer_electrons": 8,

```

```

352     "full_shell": 18
353 },
354 "Rhodium": {
355     "symbol": "Rh",
356     "atomic_number": 45,
357     "group": 9,
358     "period": 5,
359     "outer_electrons": 9,
360     "full_shell": 18
361 },
362 "Palladium": {
363     "symbol": "Pd",
364     "atomic_number": 46,
365     "group": 10,
366     "period": 5,
367     "outer_electrons": 10,
368     "full_shell": 18
369 },
370 "Silver": {
371     "symbol": "Ag",
372     "atomic_number": 47,
373     "group": 11,
374     "period": 5,
375     "outer_electrons": 11,
376     "full_shell": 18
377 },
378 "Cadmium": {
379     "symbol": "Cd",
380     "atomic_number": 48,
381     "group": 12,
382     "period": 5,
383     "outer_electrons": 12,
384     "full_shell": 18
385 },
386 "Indium": {
387     "symbol": "In",
388     "atomic_number": 49,
389     "group": 13,
390     "period": 5,
391     "outer_electrons": 13,
392     "full_shell": 18
393 },
394 "Tin": {
395     "symbol": "Sn",
396     "atomic_number": 50,
397     "group": 14,
398     "period": 5,
399     "outer_electrons": 14,
400     "full_shell": 18
401 },
402 "Antimony": {
403     "symbol": "Sb",
404     "atomic_number": 51,
405     "group": 15,
406     "period": 5,
407     "outer_electrons": 15,
408     "full_shell": 18
409 },
410 "Tellurium": {

```

```

411     "symbol": "Te",
412     "atomic_number": 52,
413     "group": 16,
414     "period": 5,
415     "outer_electrons": 16,
416     "full_shell": 18
417 },
418 "Iodine": {
419     "symbol": "I",
420     "atomic_number": 53,
421     "group": 17,
422     "period": 5,
423     "outer_electrons": 17,
424     "full_shell": 18
425 },
426 "Xenon": {
427     "symbol": "Xe",
428     "atomic_number": 54,
429     "group": 18,
430     "period": 5,
431     "outer_electrons": 18,
432     "full_shell": 18
433 }
434 }

```