# ChemBox Project Documentation

Tom Schneider

**Date:** April 20, 2024

**Center Number:** 29065
**Candidate Number:** 7638

**Ellesmere College**

# Contents

# 1 Analysis

## 1.1 Introduction

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

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

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

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

## 1.2 Prospective Users

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

## 1.3 Specific Objectives

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

### 1.3.1 Chemical Equations and Calculations - ChemCalculator

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

Required functionalities:

1. Standard moles calculation:
$$moles = \frac{mass}{molar\ mass}$$
2. Calculation to find the concentration:
$$concentration = \frac{moles}{volume}$$

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

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

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

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

4. Atom Economy calculation:

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

5. Percentage Yield calculation:

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

6. 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)

7. Equilibrium Constant calculation for a reversible reaction:

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

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

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

8. Rate Equation and Rate constant calculation:

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

Non-essential objectives:

1. Gibbs Free Energy calculation:

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

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

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

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

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

3. Acid dissociation constants $K_a$ and $pK_a$

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

$$pK_a = -logK_a$$

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

### 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) as well as choosing the option to form a dative bond. Another essential option in form of buttons should be removing atoms and bonds as well as being able to safe the drawn structures as a document. A possible non-essential enhancement would be getting extra information about atoms upon highlighting as well as getting information like the molar mass and the empirical formula of a molecule after highlighting.

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

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

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

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

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

## 1.4 Current and Proposed Systems

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

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

# 2 Documented Design

## 2.1 Introduction to the Documented Design

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

## 2.2 Project Hierarchy

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

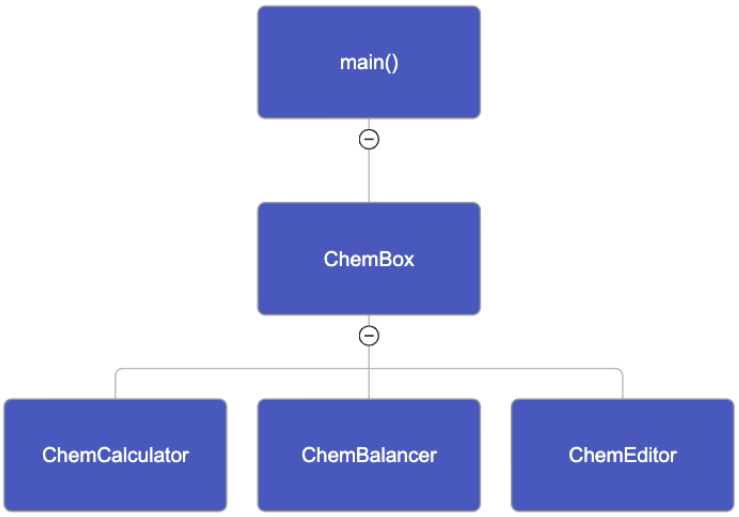



Figure 1: Program Hierarchy chart

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

## 2.3 Structure of the GUI

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

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

Listing 1: Code snippet of ChemBox class

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

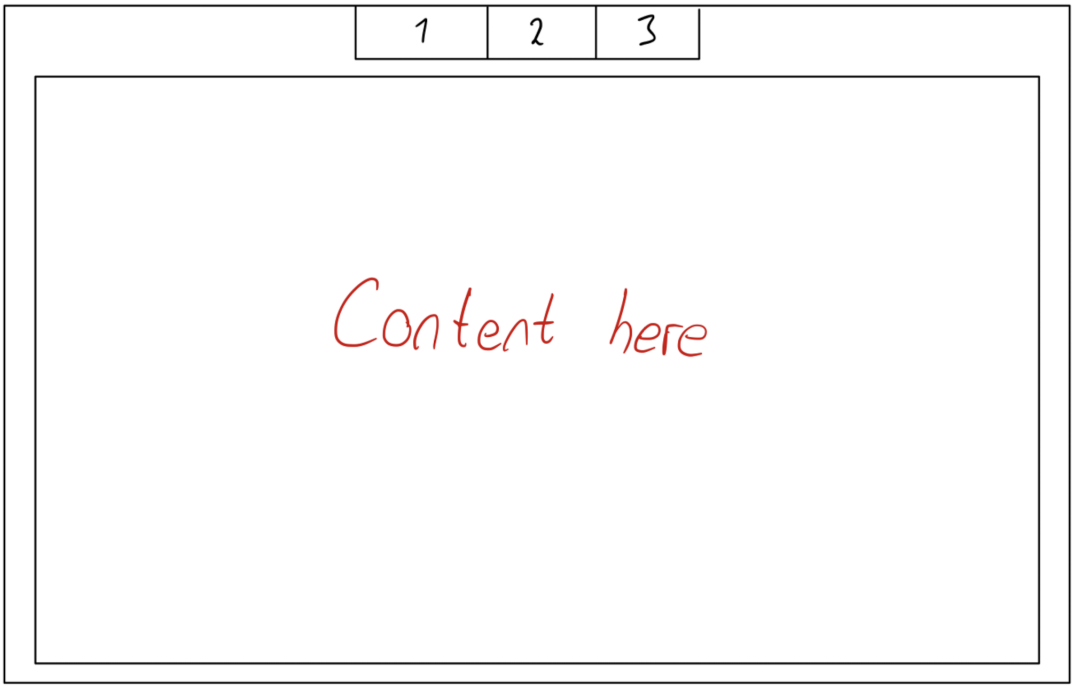



Figure 2: Mockup drawing of tab bar

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

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

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

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

        self.side_bar_layout = QVBoxLayout()

        self.gibbs_calc = GibbsFreeEnergyCalculator()

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

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

        # Create tabs
        self.gibbs_free_energy_tab = QWidget()

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

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

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

        self.page_widget = QTabWidget()

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

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

        self.main_layout = QHBoxLayout()
```

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

Listing 2: Example of ChemCalculator implementation

## 2.4 Algorithm Design for ChemCalculator

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

**Algorithm 1** Algorithm to find empty input

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

## 2.5 Algorithm Design for ChemBalancer

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

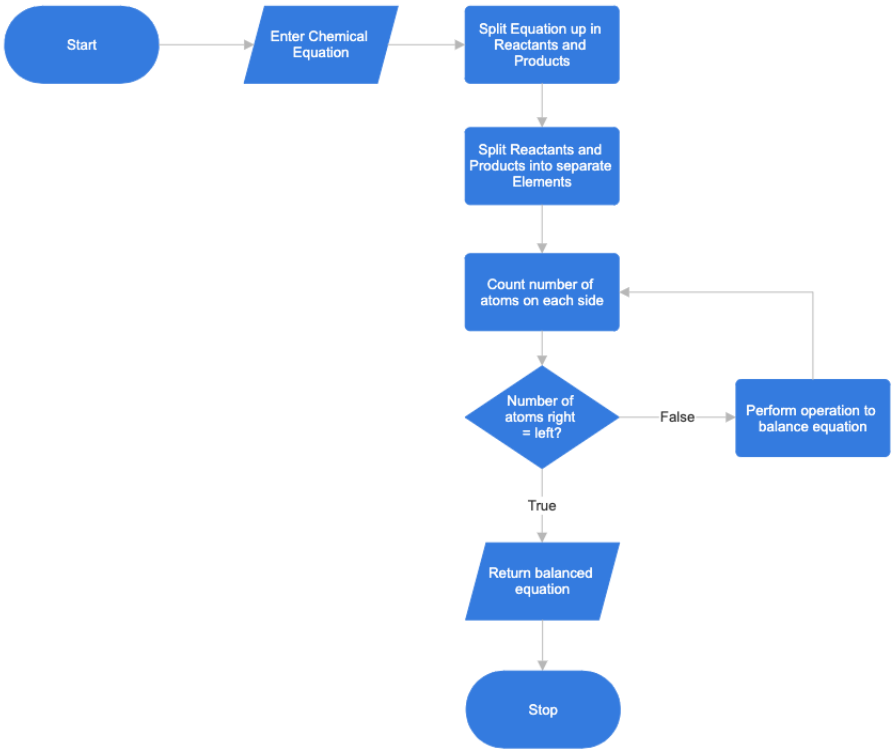



Figure 3: Initial flowchart for balancer

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

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

Listing 3: Method for splitting equation into components

The splitEquation method uses string manipulation to extract the individual molecules of the equation and store them in separate lists for reactants

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

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

                                        # Check for subscript
                                        try:
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
    subscript = int(component[(i + 3): (i + 5)])
217                                                         except
    IndexError:
218                                                             subscript =
    int(component[(i + 3): (i + 5)])
219                                                 except IndexError:
220                                                     subscript = int(
    component[i + 3])
221                                             else:
222                                                 subscript = 1
223                                         except IndexError:
224                                             subscript = 1
225
226                                         # Find subscript coefficient of
    complex ion
227                                         subCoefficient = self.
    getSubCoefficient(component)
228                                 except IndexError:
229                                     element = component[(i + 1): (i + 3)
    ]
230
231                                     # Check for subscript within
    brackets
232                                     try:
233                                         if component[i + 3] in self.
    integers:
234                                             try:
235                                                 if component[i + 4] in
    self.integers:
236                                                     try:
237                                                         if component[i +
     5] in self.integers:
238                                                             subscript =
    int(component[(i + 3): (i + 6)])
239                                                         else:
240                                                             subscript =
    int(component[(i + 3): (i + 5)])
241                                                     except IndexError:
242                                                         subscript = int(
    component[(i + 3): (i + 5)])
243                                             except IndexError:
244                                                 subscript = int(
    component[i + 3])
245                                         else:
246                                             subscript = 1
247                                     except IndexError:
248                                         subscript = 1
249
250                                     # Find subscript coefficient of
    complex ion
251                                     subCoefficient = self.
    getSubCoefficient(component)
252                                 else:
253                                     element = component[i + 1]
254
255                                     # Check for subscript within brackets
256                                     try:
257                                         if component[i + 2] in self.integers
    :
```

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

                                    # Find subscript coefficient of complex
ion
                                    subCoefficient = self.getSubCoefficient(
component)
                        except IndexError:
                            print("wrong user input ")
                        try:
                            if element in countsDict:
                                countsDict[element] += subscript *
subCoefficient * coefficient
                            else:
                                countsDict[element] = subscript *
subCoefficient * coefficient
                            if element in totalDict:
                                totalDict[element] += subscript *
subCoefficient * coefficient
                            else:
                                totalDict[element] = subscript *
subCoefficient * coefficient
                        except UnboundLocalError:
                            continue
                    else:
                        continue
                except IndexError:
                  continue
            except IndexError:
                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 balancing of the equations. The version uses a loop to attempt balancing, where copies of the left and right hand side components are created together with

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

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

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

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

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

                for index in range(0, len(leftCoefficients)):
```

```
                    for key in tempLeft[index]:
                        tempLeft[index][key] *= leftCoefficients[index]
                        if key not in totalLeft:
                            totalLeft[key] = tempLeft[index][key]
                        else:
                            totalLeft[key] += tempLeft[index][key]

                for index in range(0, len(rightCoefficients)):
                    for key in tempRight[index]:
                        tempRight[index][key] *= rightCoefficients[index]
                        if key not in totalRight:
                            totalRight[key] = tempRight[index][key]
                        else:
                            totalRight[key] += tempRight[index][key]

                self.balanced = True
                for key in totalLeft:
                    if totalLeft[key] != totalRight[key]:
                        self.balanced = False
                    else:
                        continue

            bigTup = tuple(leftCoefficients + rightCoefficients)
            leftCoefficients = list(map(lambda x: int(x / reduce(gcd, bigTup
)), leftCoefficients))
            rightCoefficients = list(map(lambda x: int(x / reduce(gcd,
bigTup)), rightCoefficients))

            balancedEquation = str()
            for index in range(0, len(self.left)):
                if leftCoefficients[index] != 1:
                    compound = str(leftCoefficients[index])
                else:
                    compound = str()
                for key in self.left[index]:
                    compound += key
                    if self.left[index][key] != 1:
                        compound += str(self.left[index][key])
                balancedEquation += compound
                balancedEquation += " + "
            balancedEquation = balancedEquation[:len(balancedEquation) - 3]
+ " = "
            for index in range(0, len(self.right)):
                if rightCoefficients[index] != 1:
                    compound = str(rightCoefficients[index])
                else:
                    compound = str()
                for key in self.right[index]:
                    compound += key
                    if self.right[index][key] != 1:
                        compound += str(self.right[index][key])
                balancedEquation += compound
                balancedEquation += " + "
            balancedEquation = balancedEquation[:len(balancedEquation) - 2]
            return self.balancedLabel.setText(f"{balancedEquation}")
```

Listing 5: balance method

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

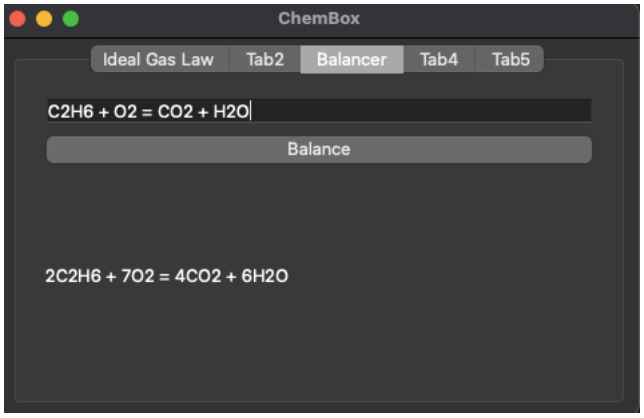



Figure 4: Initial balancer test

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

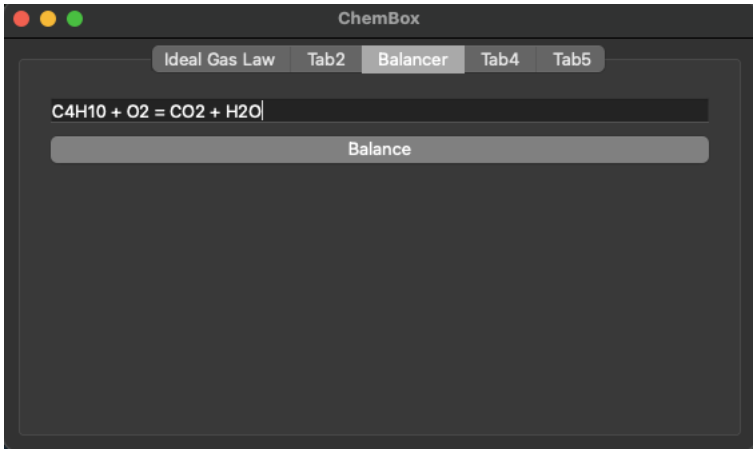



Figure 5: Initial balancer test No. 2

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

**Algorithm 2** Algorithm to split equation

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

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

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

**Algorithm 3** Algorithm to find reagents

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

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

**Algorithm 4** Algorithm to find elements

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

While I was conducting my research, I found a neat solution to balancing equations, matrix operations. As we had only covered the rudimentary principles of matrix operations, I chose to use the article 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$^{th}$ 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 pythons built in exception handling to catch any expected and unexpected errors or exceptions.

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

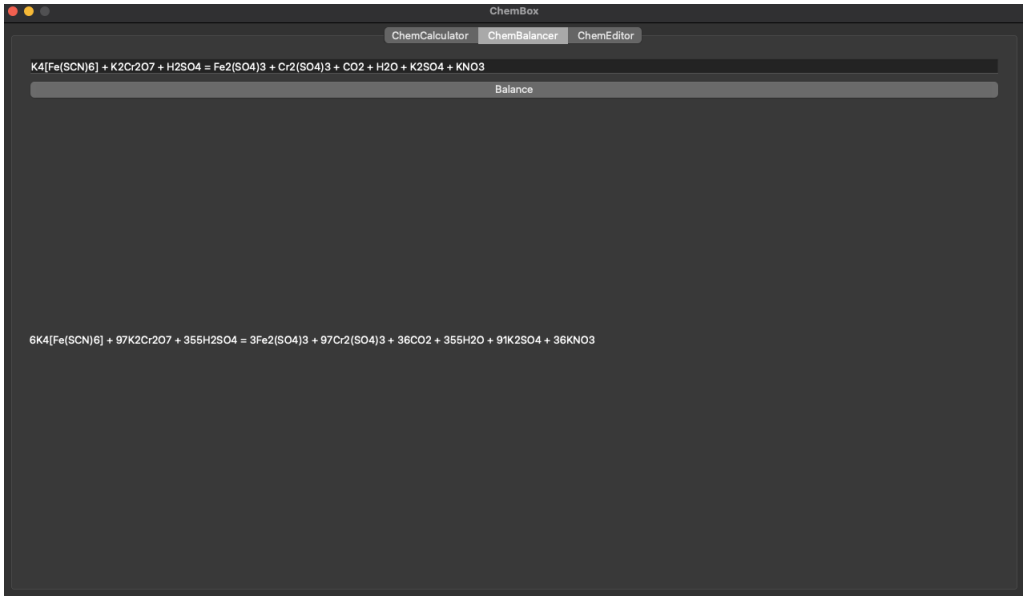



Figure 7: Balanced equation

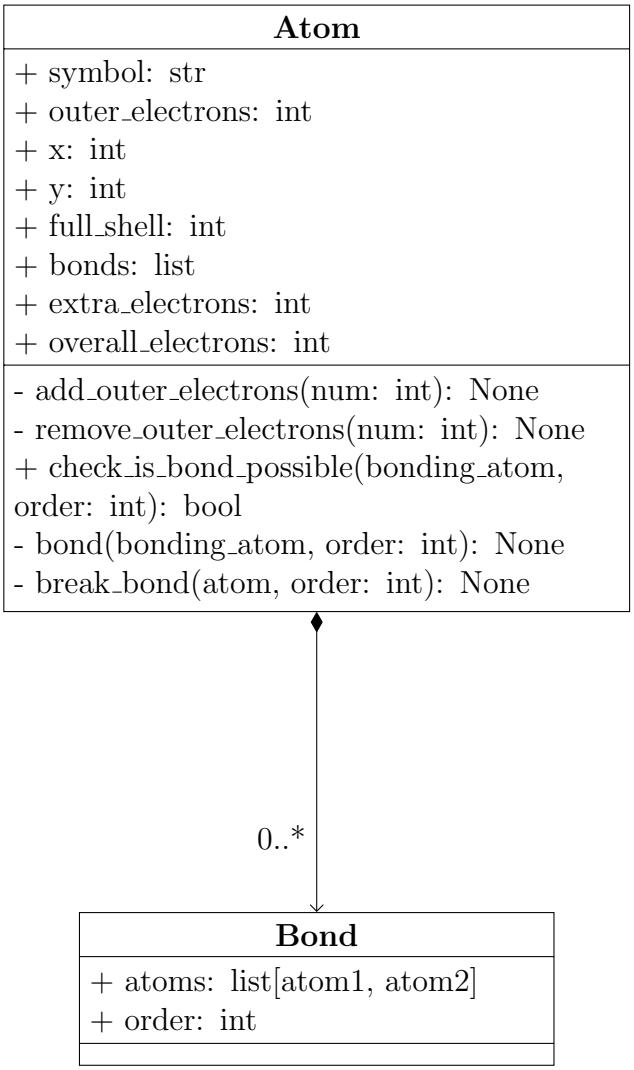
## 2.6 Algorithm Design for ChemEditor

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

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

### 2.6.1 ChemEditor Logic

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




**Algorithm 5** Algorithm to check whether a bond is possible

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

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

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

**Algorithm 6** Algorithm for bonding two atoms

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

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

### 2.6.2 ChemEditor GUI

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

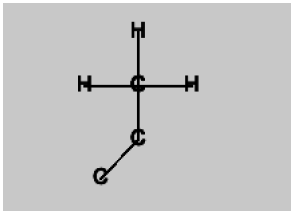



Figure 8: Overlap between atoms and bonds

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

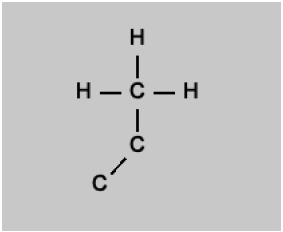



Figure 9: No overlap between atoms and bonds

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

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

**Algorithm 7** Algorithm for finding potential atom positions

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

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

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

Table 1: Method Descriptions

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

**Algorithm 8** paintEvent Algorithm, first part

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

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

**Algorithm 9** paintEvent Algorithm, second part

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

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

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

**Algorithm 10** mousePressEvent Algorithm, first part

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

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

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

**Algorithm 11** mousePressEvent Algorithm, second part

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

# 3 Testing

# 4 Evaluation