ChemBox Project Documentation

Tom Schneider

Date: March 17, 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) (ΔT = 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:

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

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

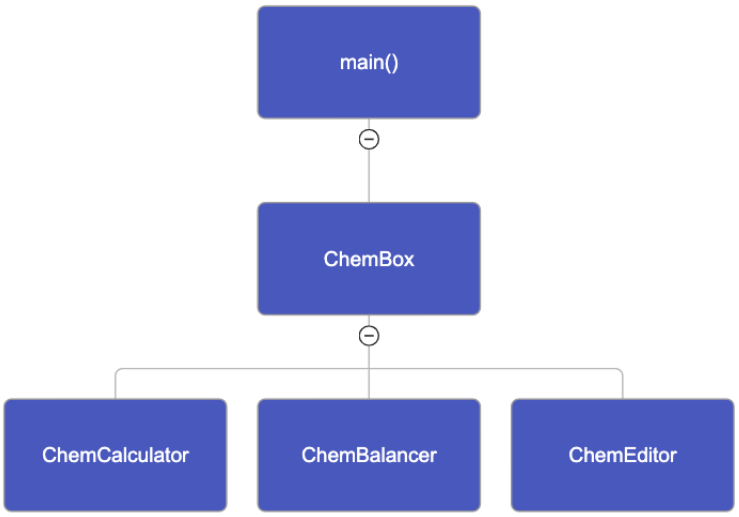

Figure 1: Program Hierarchy chart

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 No.1

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.

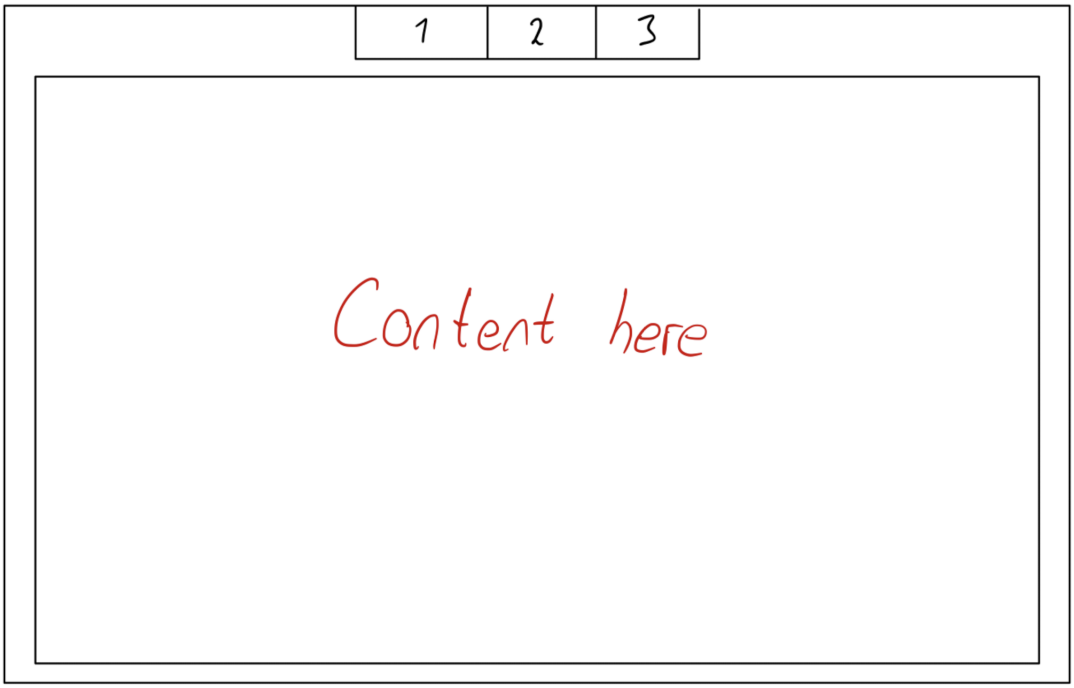

Figure 2: Mockup drawing of tab bar

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, "
    ")

```

```
30          self.page_widget.setCurrentIndex(0)
31          self.page_widget.setStyleSheet('''QTabBar::tab{
32          width: 0;
33          height: 0;
34          margin: 0;
35          padding: 0;
36          border: none;
37          }''')
38
39          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 ← []
count ← 0
empty_input ← NONE
for i ← 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
```

3 Testing

4 Evaluation