RMIT University

School of Engineering

EEET2248 – Electrical Engineering Analysis

Individual Lectorial Task

Unit Converter

Lecturer: Dr. Katrina Neville

Student Name: Nathan Paul Williams

Student Number: s3707244

Submission Due Date: 4th May 2018

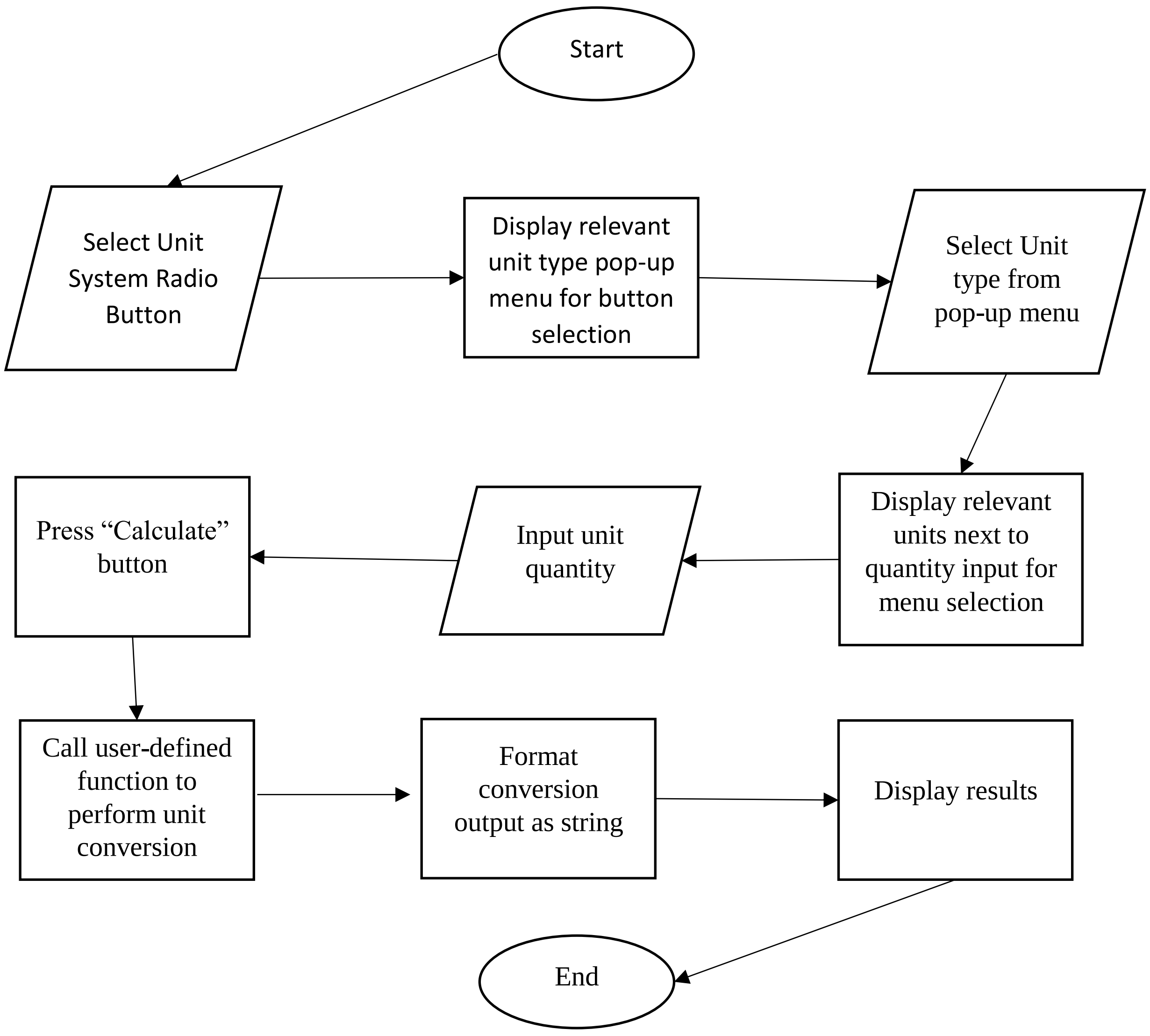
Individual Lectorial Task

Introduction

This project required a Graphical User Interface (GUI) that could convert to/from different unit systems (i.e. metric to imperial or vice versa). The program had to take user input, provide an accurate conversion and produce relevant errors in invalid input cases. It also had to use a single user-defined function for the unit conversion. A focus was put on making the program as user-friendly as possible with extra features added to ensure this; these are discussed in detail below.

The purpose of the program is to allow users to convert units to/from their native unit system to allow effective communication between international members of various fields or even for general purpose use by civilians. The target audience is extremely broad for this application however it’s clear that it can be used as a powerful communication and interpretation tool in a wide range of scenarios.

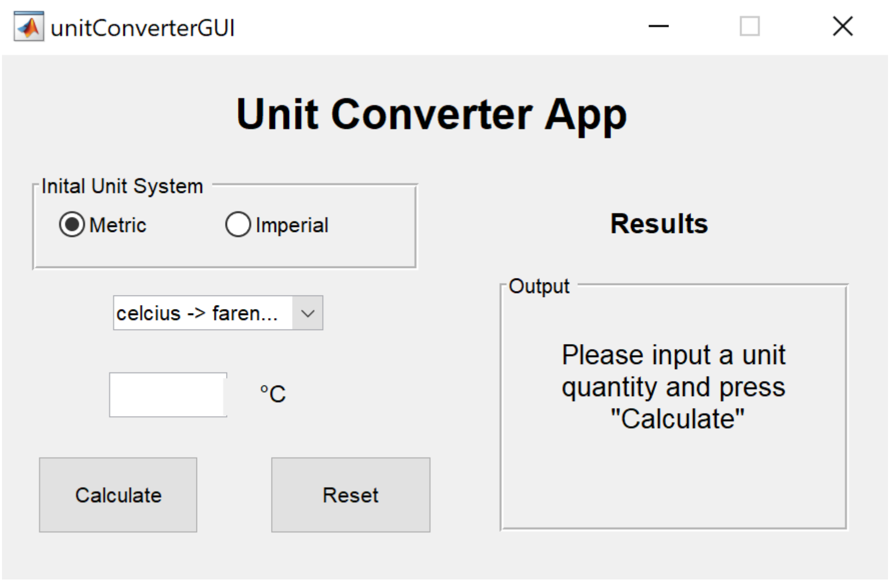
Design and Methodology



Start
Select Unit System Radio Button
Display relevant unit type pop-up menu for button selection
Select Unit type from pop-up menu
Display relevant units next to quantity input for menu selection
Input unit quantity
Press “Calculate” button
Call user-defined function to perform unit conversion
Format conversion output as string
Display results
End

Figure 1: Flowchart showing program structure

For my GUI design I decided to focus on user friendliness, error handling and minimising the amount of work required by the user (e.g. clicks) to obtain a result(s). This resulted with the GUI pictured below.

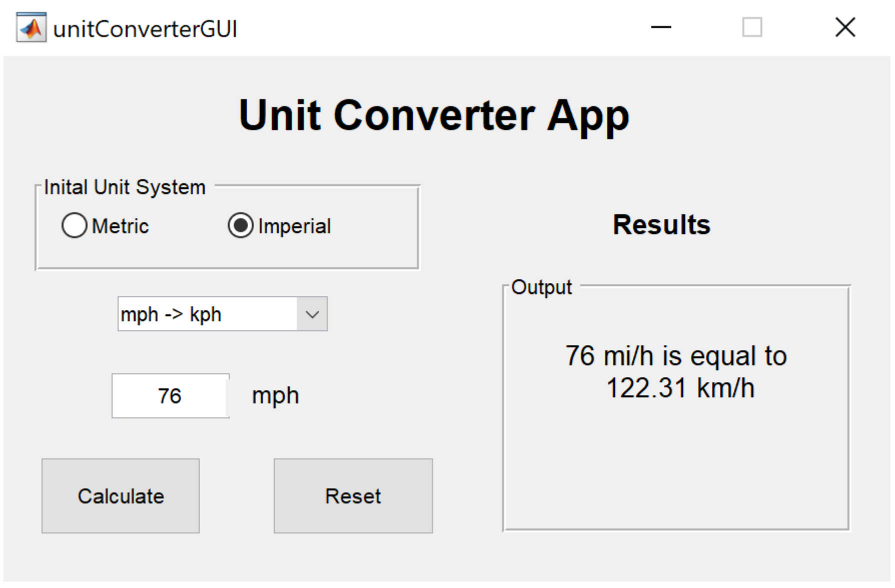


Please input a unit quantity and press "Calculate"

Figure 2: GUI in initial state

From the initial state pictured above (fig 2) the program operation has the following structure. The user will select an initial unit system by clicking the relevant radio button. A selection changed callback function on the button group will cause the pop-up menu to then change to display options for the relevant system. In reality, there are two menus occupying the same position and the visibility is being switched on/off for the relevant menu using the set function. The user then chooses the unit type they would like to convert from the pop-up menu and enters a unit quantity in the box below. The unit symbol next to the quantity input box will always display whichever unit type is selected in the drop-down menu, this includes when changing unit systems and hence changing the menu options. This is done in the menu callback functions with a switch statement to decide which unit. In the case of the radio buttons being changed the selection changed function makes a call to the same menu callback function.

After the user has input the data they can click the calculate button. This executes the calculate callback which makes a call to the user-defined unit conversion function. This function is passed input of a unit system (as a character), a unit system (as a double based on the value of the pop-up menu obtained via get() function) and a unit quantity (as a double after being converted using str2double()). It then outputs the converted unit quantity, two strings for the original and converted units and two error counters which are used later to print error messages (the error process is explained in detail in the solution and testing section below). Sprintf() is used to format the output into a result string which we can display. Using set() this string is then displayed in the result box on the right. Prior to this the result box displays a small instructional message to help the user get started.



76 mi/h is equal to 122.31 km/h

Figure 3: Example output 1

Displaying the result as a string is user-friendly as it makes it as easy as possible to interpret while displaying original and converted unit quantity and type efficiently. Fig 3 above shows an example output in a standard scenario.

There is also a reset button which upon pressing has its own callback that uses the set function and calls other callback functions to reset the GUI to it's initial state displayed in fig2. Another minor feature is that when switching between unit systems via the radio buttons the drop down menus will transfer values. This means that for example, if g->oz is selected and the initial unit system is changed to imperial the drop down will now display oz->g and the unit next to the unit quantity input box will change respectively. This program structure delivers a user-friendly and intuitive experience with minimal work required for the user and maximises functionality.

The visual layout of the GUI was designed to follow a natural reading path of left to right as well as allow the user to input the data in a top-down format on the left side of the screen. After experimenting with several different layouts this appeared to be the most intuitive and user-friendly based on asking a small set of participants to try each layout and provide feedback.

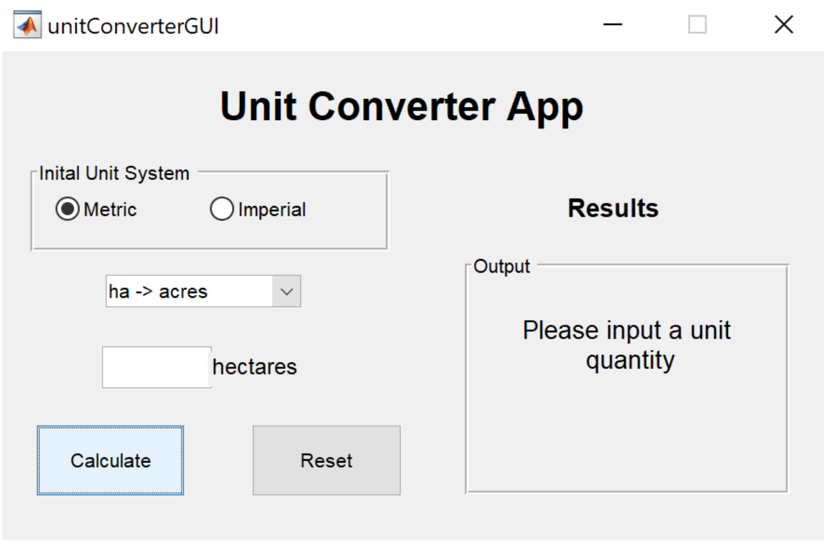
Output and Testing

Original Input	Google's Converted Output	My Converted Output
55 degrees Fahrenheit	12.7778 degrees Celsius	12.7778 degrees Celsius
20 centimetres	7.87402 inches	7.87402 inches
5 metres	16.4042 feet	16.4042 feet
10 km	6.21371 miles	6.21371 miles
40 grams	1.41096 oz	1.41096 oz
88 kilograms	194.007 lbs	194.004 lbs
100 km/h	62.1371 mph	62.1371 mph
20 Litres	5.28344	5.2834
50 hectares	123.553	123.555
-12 degrees Celsius	10.4 degrees Fahrenheit	10.4 degrees Fahrenheit
5 inches	12.7 centimetres	12.7 centimetres
20 feet	6.096 metres	6.096 metres
3 miles	4.82803 km	4.82803 km

28 oz	793.787 grams	793.786 grams
200 lbs	90.7185 kg	90.72 kg
88 mph	141.622	141.622
4 gallons	15.1416 Litres	15.1418 Litres
75 acres	30.3514	30.3509

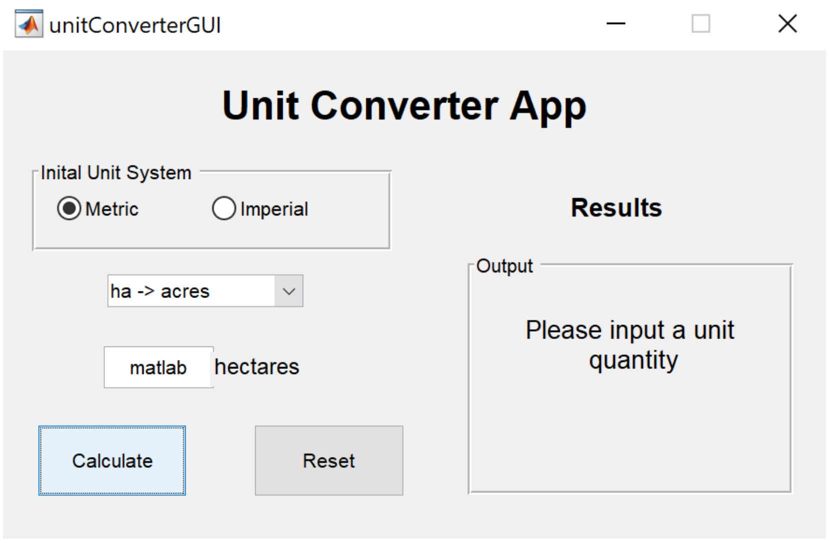
Figure 4: Table of test data

To test the accuracy of the program the same input data as milestone 3 was input into both Google's unit converter and the new GUI. The results displayed in figure 4 show that the GUI results are identical to Google's in 12 out of 18 cases. In the other 6 cases the answers are within a very small error range (in the worst case there is a 0.0015 difference in the results). This is evidence that our program is producing accurate results if we assume that Google's unit converter (a widely used app) is also functioning correctly. This is expected as the user-defined function which is responsible for the arithmetic of the conversions is almost identical to the one used in milestone 3 which produced the same output.



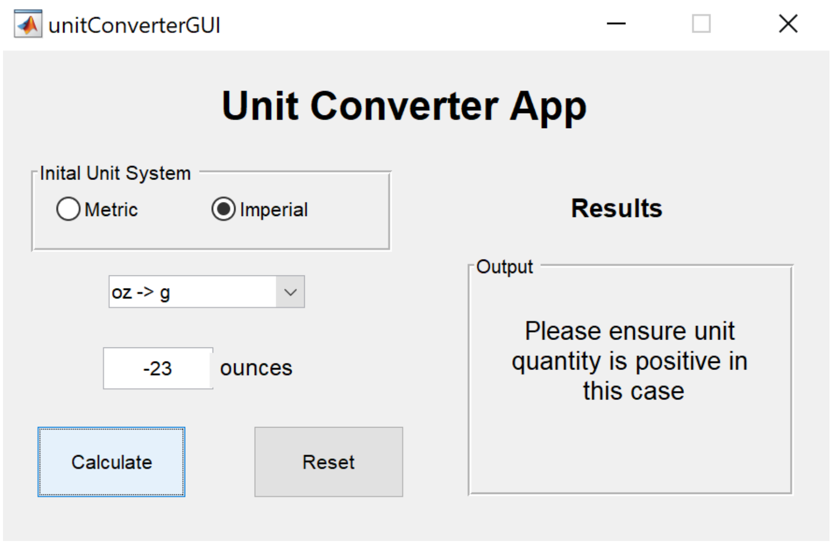
Please input a unit quantity

Figure 5: Error case empty input



Please input a unit quantity

Figure 6: Error case text input


unitConverterGUI

Unit Converter App

Inital Unit System

Metric [✓ Imperial]

oz -> g

-23 ounces

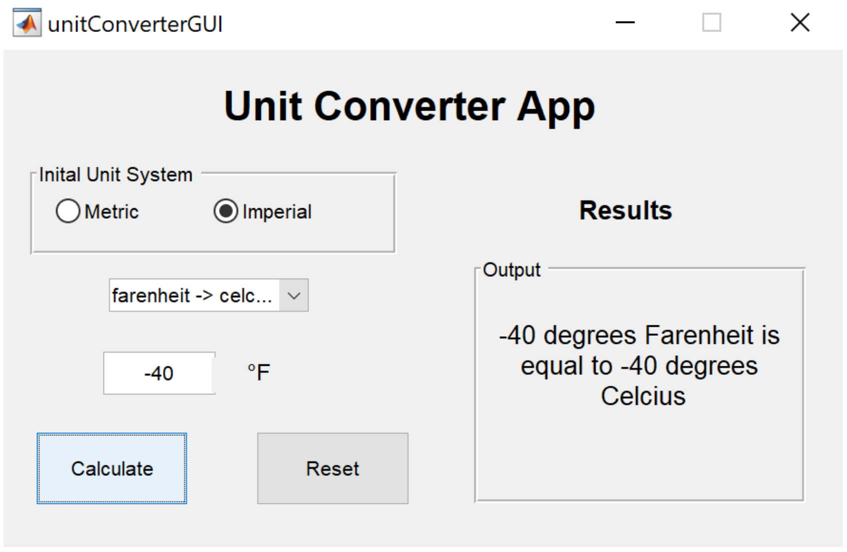
Calculate Reset

Results

Output

Please ensure unit quantity is positive in this case

Figure 7: Error case invalid negative input


unitConverterGUI

Unit Converter App

Inital Unit System

Metric [✓ Imperial]

farenheit -> celc...

-40 °F

Calculate Reset

Results

Output

-40 degrees Farenheit is equal to -40 degrees Celcius

Figure 8: Negative temperature example

Implementing a GUI allows the input to be more controlled hence reducing the chance of errors however there are still 2 error messages that our program can display. These messages are displayed in the result box. The first message is seen in figures 5 and 6. This occurs when the user inputs a blank input or a text input into the unit quantity box. The user-defined function is passed a str2double() conversion of whatever is in this box. In both these cases a NaN value will be passed in so the function performs an isnan() and if true returns the first error counter as true in the output which causes the program to print the displayed message. The other message is seen in figure 7 in the case of a negative input for an invalid unit. The user-defined function performs a check to see if the input quantity is negative and the unit type is not a temperature. If so it will return the second error counter as true which causes the program to print the displayed message. Figure 8 shows the program will correctly convert negative values if the unit type is a temperature.

The attached figures show that our program appears accurate, can handle invalid inputs and produces output that meets the requirements outlined and provides the user with an effective method to convert units from imperial to metric and vice versa. It solves the problem of different countries using different unit systems by providing an app that will allow people to convert units to their native system and work with them in any field or scenario.

Conclusion

The program fulfils the requirements of the task, providing a user-friendly functional GUI that can accurately convert to/from unit systems, using a single user-defined function, and handles any invalid inputs with neat error messages. The testing suggests that the program is outputting accurate results and functioning as intended in all aspects.

A possible improvement to the program could be to create a help button that opens a new dialog box providing the user with more detailed instructions of how to operate the application and possibly links to website containing more information such as the formulas used for the conversions. Another possible improvement could be to rewrite the program so that there is no need for the calculate button and results are automatically displayed upon data being input. The program could also be rewritten in another language (java, swift, etc.) to convert it to the form of a web or mobile app allowing it to be more accessible to a wider range of users. These tasks however would require a large addition of code and problem solving that is beyond the scope of this project specifically.