Development

Program Code

This is the code that composes the program, supplemented with direction on how each segement works.

#This function starts the program, creating an instance of the window at the center of the screen.

def main():

print(threading.enumerate())

app = QtGui.QApplication(sys.argv)

font = QtGui.QFont("Lucida Sans Unicode")

app.setFont(font)

winW = 480

winH = 640

winX = 48

if winW <= app.desktop().screenGeometry().width():

winX = (app.desktop().screenGeometry().width() / 2) - (winW / 2)

winY = 48

if winH <= app.desktop().screenGeometry().height():

winY = (app.desktop().screenGeometry().height() / 2) - (winH / 2)

MainWindow = Window(winX, winY, winW, winH)

sys.exit(app.exec\_())

This function creates the window that the main program displays itself on. As can be seen, the window has dimensions of width 480 pixels and height 640 pixels and is created in the middle of the screen. The program determines the dimensions of the screen using the attribute of the app.desktop() class screenGeometry(), to find its own attributes width and height. However, if the program’s window is bigger than the monitor used to display it, it is created in the top left of the screen instead, to prevent any part of the window from being cut off by the edge of the screen.

#Creating the Window function.

class Window(QtGui.QMainWindow):

#Defining the default window dimensions.

winX = 48

winY = 48

winW = 640

winH = 480

def \_\_init\_\_(self, winX, winY, winW, winH, parent = None):

super(Window, self).\_\_init\_\_(parent)

#Defining default attributes for instances of the Window class.

self.winW = winW

self.winH = winH

self.winTitle = "Calculator"

self.setGeometry(winX, winY, winW, winH)

self.setWindowTitle(self.winTitle)

self.MenuWidgets = []

self.CalculatorWidgets = []

self.percent = 0

self.Equation = ""

self.CalculatorExpression = ""

self.expList = [[""]]

self.Answer = 0

self.AnswerExpression = ""

self.GraphWidgets = []

self.PropertiesWidgets = []

self.CurveSlots = []

self.Curve = None

self.CreatingGraph = False

self.PropertiesWidgetsCreated = False

self.BoundsUneven = False

self.Curves = []

self.Bounds = []

#Immediately calling the Menu with the buttons 'Calculator', 'Graph' and 'Quit'.

self.Menu((self.winW / 2), (self.winH / 3), 128, 64,

[

["Calculator", 0, self.MOVECalculator],

["Graph", 0, self.MOVEGraphProperties, [[]]],

["Quit", 0, sys.exit],

])

This class holds the attributes used by the window to determine its properties. As can be seen, the default values for the window’s placement are above \_\_init\_\_, setting what would be the values for the window if the values given by the main() function were erroneous. Self.winTitle contained the title of the window, editing the text that appears at the top of the window, in the window frame. In addition, the four lists MenuWidgets, CalculatorWidgets, GraphWidgets and PropertiesWidgets contained the widgets for each of their respective screens, and CurveSlots contained the five slot buttons holding the equations for each of the curves in the graphs created.

\_\_init\_\_ works immediately upon the creation of an instance of a class, and is the first subroutine to run before any other subroutine in a class. This means that the subroutine self.Menu, another subroutine attribute of the Window class, would also run inside \_\_init\_\_ immediately, calling the main menu of the program. As can be seen, the menu is created with buttons ‘Calculator’, ‘Graph’ and ‘Quit’. These buttons call self.MOVECalculator, self.MOVEGraphProperties, and sys.exit respectively, at an offset from the middle of 0 pixels for each one, meaning that the middles of the buttons were completely aligned with the centre of the window.

#Create the widgets for the menu.

def Menu(self, x, y, w, h, Buttons):

#Empty the widget list for Menu widgets.

self.MenuWidgets = []

for i in range(0, len(Buttons)):

#Set the buttons' offset so they appear in the middle of the screen.

Offset = (Buttons[i][1] - 1) \* (w / 2)

#Give the button parameters if it has any.

try:

self.Button(x+Offset, y+(i\*h), w, h, Buttons[i][0], self.MenuWidgets, True, "default", Buttons[i][2], Buttons[i][3])

except:

self.Button(x+Offset, y+(i\*h), w, h, Buttons[i][0], self.MenuWidgets, True, "default", Buttons[i][2])

self.ShowWidgets(self.MenuWidgets)

self.show()

The Menu subroutine creates the main menu for the program. It takes in the x and y positions of the menu, as well as the width and height of the buttons, and then takes all of the menu options as a list, with each option having a sub-list with its properties. In these sub-lists, the index 0 holds the name of the option that would be displayed on the button, index 1 holds the offset of the button in terms of half of its width (for example an offset of 1 would displace a button of width 64px 32 pixels to the right, as 64/2 = 32), however all buttons are created at the centre of the screen as programmed by the line (Offset = (Buttons[i][1] - 1) \* (w / 2).

What this line is doing is taking the offset given in index 1, and removing one from it. This is because of the fact that PyQt automatically right aligns all its widgets, meaning that all its widgets would be created with their top left corners at the specified coordinates. This would mean that in order to create a widget at the centre of the screen, one would have to remove half of its width from the centre coordinate, meaning that it would be shifted so that it became centrally aligned instead.

The try/except statement is there to cover cases where arguments may need to be inputted for the functions buttons are calling. It would mean that if the button’s call function had arguments, contained within a fourth optional index of the list, it would give those as arguments. If it didn’t, it would cause an IndexError, prompting the program to instead execute the except statement where no arguments had been specified. It then shows all of the menu widgets (the three buttons), and finishes running.

#The function used to create a QButton and append it to a list of widgets tied to a screen.

def Button(self, x, y, w, h, msg, List, Shown, color, cmd, args = ()):

#If the button needs HTML tag support a label is created to hold the text instead of the button, as labels support these tags while buttons don't.

if msg == self.CalculatorExpression or msg == self.Answer or color == "equation":

btn = QtGui.QPushButton("", self)

lbl = QtGui.QLabel(msg, self)

else:

msg = self.CorrectFontErrors(msg)

btn = QtGui.QPushButton(msg, self)

#Set what the button is to do if clicked, if there are arguments, pass those into the call function.

if args != ():

btn.clicked.connect(lambda: cmd(\*args))

else:

btn.clicked.connect(cmd)

#Set the button's position and size.

btn.move(x, y)

btn.resize(w, h)

try:

lbl.move(x+12, y)

lbl.resize(w-24, h)

lbl.setAlignment(QtCore.Qt.AlignCenter)

List.append([btn, lbl])

except:

List.append([btn])

#Show the button if it is meant to be immediately displayed.

if Shown == True:

btn.show()

try:

lbl.show()

except:

pass

#Change the background color and the text font.

if color != "default" and color != "equation":

btn.setStyleSheet("background-color: " + color);

btn.setStyleSheet("font-family: arial\_unicode\_ms”)

#One function containing all the necessary functions to create can instance of a label and append it to a list of widgets tied to a screen.

def Label(self, x, y, w, h, msg, List):

lbl = QtGui.QLabel(msg, self)

lbl.move(x, y)

lbl.resize(w\*2.5, h)

lbl.setAlignment(QtCore.Qt.AlignCenter)

lbl.show()

List.append([lbl])

#One function containing all the necessary functions to create can instance of a text fill form and append it to a list of widgets tied to a screen.

def TextBox(self, x, y, w, h, msg, List):

box = QtGui.QLineEdit(self)

lbl = QtGui.QLabel(msg, self)

box.move(x, y)

box.resize(w, h)

box.show()

lbl.move(x-(w\*0.75), y-32)

lbl.resize(w\*2.5, h)

lbl.setAlignment(QtCore.Qt.AlignCenter)

lbl.show()

List.append([box, lbl])

#One function containing all the necessary functions to create can instance of a spin box form and append it to a list of widgets tied to a screen.

def SpinBox(self, x, y, w, h, msg, nMin, nMax, List):

box = QtGui.QSpinBox(self)

lbl = QtGui.QLabel(msg, self)

box.move(x, y)

box.resize(w, h)

box.show()

box.setMinimum(nMin)

box.setMaximum(nMax)

lbl.move(x-(w\*0.75), y-32)

lbl.resize(w\*2.5, h)

lbl.setAlignment(QtCore.Qt.AlignCenter)

lbl.show()

print(msg + ":", len(List))

List.append([box, lbl])

The button function was created for convenience’s sake. I mainly used it to prevent myself having to write more code for button widgets than I needed to. I did this for every widget I used, including text fills, spin boxes and labels, however it was especially needed for the button widget as I used it so often that it would have become incredibly difficult and cumbersome to rewrite all instances of a QButton to change one small thing that would apply to every button. This was especially apparent due to the nature of how PyQt handled the creation of buttons. Multiple lines of code needed to be added to size the button, position it and set its other attributes and this would have been much more problematic as it would have been difficult to locate where each individual button was being created, especially in code that was this long.

The use of a list to store all of the widgets of a particular screen meant that they were all kept together and were easily accessible. If one needed to be altered or removed, I could easily reference it inside the list, and the use of the lists also meant that it was much easier to affect all of the widgets on a screen at once, for example when I needed to refresh all widgets on a screen to update their messages, as would often need to happen with the calculator screen.

#Move to the calculator screen.

def MOVECalculator(self):

self.Equation = ""

self.CalculatorExpression = self.Equation

self.HideWidgets(self.MenuWidgets)

self.Calculator(

[

["##############################################"],

["1", "2", "3", "4", "5", "6", "7", "8", "9", "0"],

[" + ", " - ", " \* ", " / ", "^", "√"],

[".", " ", " ", "a", "x", " ", " ", "S", "L", "π"],

["|", "(", ")", "b", "y", " ", " ", "C", "E", "e"],

["!", " ", " ", "c", "z", " ", " ", "T", " ", "φ"],

[" GRAPH ", " CALC ", " ← ", " AC "],

[" ROUND ", " = "],

[" CLOSE "],

])

If one clicked the ‘Calculator’ button on the main menu, the program at this point would be running the self.MOVECalculator procedure. Like self.MOVEMenu, this subroutine is contained within the Window class, and as such, affects the program window. It is instrumental in the creation and display of the calculator screen, as it determines the layout of the calculator in preparation for the creation of the widgets.

MOVECalculator has one main function, which is to clear the window of all widgets and display the calculator screen, effectively moving between whichever screen the user was on and the calculator screen. In this, it hides all of the current widgets from the screen, refreshing it in the process, and then runs self.Calculator to create or display the widgets as necessary.

#Create the calculator screen and layout.

def Calculator(self, Layout):

#Set the base factor for button width.

gbw = self.winW / len(Layout[1])

print(gbw)

#Set the base factor for button height.

gbh = self.winH / len(Layout)

#Iterate through the calculator layout grid.

for y in range(0, len(Layout)):

cx = 0

for x in range(len(Layout[y])):

#Set the width of the button.

bw = int(gbw)

if Layout[y][x] == " ":

bw = int(gbw)

#Create a larget width based on a multiplication of the base factor.

elif Layout[y][x].count(" ") > 0 or Layout[y][x].count("#") == len(Layout[y][x]):

bw = (round((len(Layout[y][x]) - 1) / 3) \* gbw)

#If there is a button to be created...

if Layout[y][x] != " ":

print(Layout[y][x])

#Create all buttons that have a more complex display than their code.

#Create the expression display.

if Layout[y][x].count("#") == len(Layout[y][x]):

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, self.CalculatorExpression, self.CalculatorWidgets, True, "default", print, self.CalculatorExpression)

#Create the graph button.

elif "GRAPH" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, (Layout[y][x]).replace(" ",""), self.CalculatorWidgets, True, "default", self.CalculateGraph)

print("GRAPH Button")

#Create the close button.

elif "CLOSE" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, (Layout[y][x]).replace(" ",""), self.CalculatorWidgets, True, "default", self.MOVEMenu)

print("CLOSE Button")

#Create the calculate button.

elif "=" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, (Layout[y][x]).replace(" ",""), self.CalculatorWidgets, True, "default", self.Calculate, (True, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

print("EQUATION Button")

#Create the all clear button.

elif "AC" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, (Layout[y][x]).replace(" ",""), self.CalculatorWidgets, True, "default", self.THREADUpdate, ("AC", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the rounding spin box.

elif "ROUND" in Layout[y][x]:

self.SpinBox(cx, y\*gbh+(gbh/2), min(self.winW, bw, self.winW - cx), (gbh/2), "Round to ... d.p.", 0, 10, self.CalculatorWidgets)

#Create the multiplication and division buttons.

elif "\*" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "×", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("\*", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif "/" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "÷", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("/", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif "^" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL X}" + "\N{SUPERSCRIPT LATIN SMALL LETTER N}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("^", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the modulus button.

elif "|" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "abs", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("A(", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the Differentiation button.

elif "CALC" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "DIFFERENTIATE", self.CalculatorWidgets, True, "default", self.Differentiate, (0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the buttons for the trigonometric functions.

elif "S" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "sin", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("S(", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif "C" in Layout[y][x] and not "AC" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "cos", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("C(", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif "T" in Layout[y][x]:

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "tan", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("T(", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the factorial button.

elif Layout[y][x] == "!":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL X}" + "!", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("!", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the pi and phi buttons.

elif Layout[y][x] == "π":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL BOLD SMALL PI}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("π", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "φ":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL BOLD SMALL PHI}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("φ", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create ln and log buttons.

elif Layout[y][x] == "L":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "log", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("L(", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "E":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "ln", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("E(", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the square root button.

elif Layout[y][x] == "√":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{SUPERSCRIPT LATIN SMALL LETTER N}" + "\N{SQUARE ROOT}" + "\N{MATHEMATICAL ITALIC SMALL X}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("√x", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the variables buttons.

elif Layout[y][x] == "a":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL A}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("a", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "b":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL B}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("b", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "c":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL C}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("c", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "e":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL E}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("e", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "x":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL X}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("x", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "y":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL Y}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("y", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "z":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "\N{MATHEMATICAL ITALIC SMALL Z}", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("z", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Create the f(x) and var buttons (inactive).

elif Layout[y][x] == "F":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "f(" + "\N{MATHEMATICAL ITALIC SMALL X}" + ")", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

elif Layout[y][x] == "v":

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, "var", self.CalculatorWidgets, True, "default", self.THREADUpdate, ("", False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

else:

#Create a button that has the same label as its layout code.

self.Button(cx, y\*gbh, min(self.winW, bw, self.winW - cx), gbh, (Layout[y][x]).replace(" ",""), self.CalculatorWidgets, True, "default", self.THREADUpdate, ((Layout[y][x]).replace(" ",""), False, 0, 0, min(self.winW, round((len(Layout[0][0]) - 1) / 3) \* gbw), gbh))

#Add the button width onto the current x-coordinate on the window.

cx += bw

#Show all of the buttons.

self.ShowWidgets(self.CalculatorWidgets)

Calculator functions to create all the widgets for the calculator screen. It is only run once, not every frame, as that would be unnecessary, especially since I did not need to create multiple versions of a widget due to the presence of the widget lists as Window attributes.

The procedure works by reading the parameter of the two-dimensional list given to it as an argument as a layout of the calculator. This was done to make it easily amendable if complications arose. It would read it through line by line, creating the buttons as it went. The size of the strings the buttons were created due to corresponded to the width of the button on the screen. For example, “2”, with only one character, would have a width of only 32, as . The program would also automatically censure any spaces from the string it was created from as well, as if the string was even in its number of characters, a single character label would appear slightly off centre on the button. As such, a string of " = " would be formatted to make a button label of “=” before the button was created.

#Calculates the answer to the expression given.

def Calculate(self, GiveAnswer, x, y, w, h):

print("Expression:", self.Equation)

if self.Equation != "":

#Calculate an answer if the expression is not empty.

self.Answer = Calculation\_Process.Input(self.Equation, "Calculation", [['x', 1], ['y', 2], ['z', 3], ['a', 5], ['b', 10], ['c', 20]])

#If there were no errors in the expression...

if not isinstance(self.Answer, str):

#If the answer is to be rounded to a whole number...

if self.CalculatorWidgets[40][0].text() == "0":

#Round it, making it an integer.

self.Answer = int(round(self.Answer, int(self.CalculatorWidgets[40][0].text())))

#Otherwise...

else:

#Round it, making it a float.

self.Answer = round(self.Answer, int(self.CalculatorWidgets[40][0].text()))

#Format the displayed expression.

self.AnswerExpression = self.FormatExpression(str(self.Answer), str(self.AnswerExpression))

print("Answer:", self.AnswerExpression)

print("")

#Update the expression display to show the answer.

self.UpdateExpression("", True, x, y, w, h)

The self.Calculate procedure as the name suggests, calculated an answer to the expression inputted by the user. It used the Input function of another file, Calculation\_Process, to produce its answers. The algorithm used in the calculation was the algorithm that I had created, and that an outline could be seen in the design section of this document. The self.Calculate function would then format the answer given in case it was not a purely numerical answer, and updated the expression in the calculator display to reflect the obtained answer. It is important to note that in the calculation, the variables ‘x’, ‘y’, ‘z’, ‘a’, ‘b’ and ‘c’ had been given their own values of 1, 2, 3, 5, 10 and 20 respectively. This was to ensure that calculations using them were performed smoothly, as the program would run into an error if it ran into a string whilst calculating, as would happen if variables were inputted that didn’t have a value (they would not have been substituted for integer values in the preparation). In addition, before the display was updated, the program would round the value based on the contents of the spin box labelled ‘Round to … d.p.’. It was only then that the display would be updated.

#Differentiate the equation.

def Differentiate(self, x, y, w, h):

print("Expression:", self.Equation)

#Differentiate if the expression is not empty.

if self.Equation != "":

self.Answer = Calculation\_Process.Input(self.Equation, "Differentiation", [["x"], ["y"], ["z"], ["a"], ["b"], ["c"]])

#Format the displayed expression.

self.AnswerExpression = self.Answer

self.AnswerExpression = self.FormatExpression(str(self.Answer), str(self.AnswerExpression))

print("Differential:", self.AnswerExpression)

print("")

#Update the expression display to show the differential.

self.UpdateExpression("", True, x, y, w, h)

The self.Differentiate procedure acted very similarly to the self.Calculate procedure. It still formatted the resulting differential, as the formatting of the differential was invariably more important here. Unlike results from the self.Calculate function, differentials would usually still have algebraic notation embedded into them, which would have required proper formatting to ensure they were readable.

#Format the expression given to it.

def FormatExpression(self, Equation, CalculatorExpression):

print("-----------Formatting-----------")

print(Equation)

CalculatorExpression = Equation

print(CalculatorExpression)

Term = ""

Digits = "0123456789."

Variables = "πφeabcdefghijklmnopqrstuvwxyz"

Operators = "+-×÷^"

Functions = [["S(", "sin("], ["C(", "cos("], ["T(", "tan("], ["L(", "log("], ["E(", "ln("], ["A(", "abs("]]

Primary = Digits + Variables + "-("

Precendentary = ")"

Prohibited = ["√x", "^", "+", "-", "×", "÷"]

Primary.replace("-", Operators)

Modulus = False

InBrackets = False

Done = False

#Turn 'n\*x' and ')\*(' into 'nx' and ')('.

for i in range(0, len(Variables)):

CalculatorExpression = CalculatorExpression.replace("\*" + Variables[i], Variables[i])

CalculatorExpression = CalculatorExpression.replace("\*" + "(", "(")

CalculatorExpression = CalculatorExpression.replace(")" + "\*" + "(", ")(")

CalculatorExpression = CalculatorExpression.replace(Variables[i] + "^0.5", "√" + Variables[i])

#Turn any remaining '\*' into '×', and '/' into '÷'.

CalculatorExpression = CalculatorExpression.replace("\*", "×")

CalculatorExpression = CalculatorExpression.replace("/", "÷")

#Check if there are any indices.

if Term != "^":

#Check if the power is a function or within brackets.

Encapsulations = [["S", "C", "T", "L", "E", "A", "("], [")"]]

#Check if there are any power symbols.

while CalculatorExpression.count("^") > 0:

#Jump to the index of the power symbol.

index = CalculatorExpression.index("^")

print("Index:",index)

e = ''

print("Length:",len(CalculatorExpression))

#Move through the characters of the power.

for i in range(index+1, len(CalculatorExpression)):

#If there are no brackets or functions, only power a single term.

if CalculatorExpression[index+1] not in Encapsulations[0]:

if CalculatorExpression[i] in ("1234567890." + Variables):

e = e + CalculatorExpression[i]

print('e:',e)

else:

break;

#Else, power the entire bracket.

else:

e = e + CalculatorExpression[i]

print('e:',e)

if e[-1] in Encapsulations[1]:

break;

print("i:",i)

print("")

eo = e

#If there are no characters of the power, generate an empty slot to allow the user to know they are inputting one.

if e == '':

e = '\N{REPLACEMENT CHARACTER}'

#Make the power superscript.

e = '<sup>' + e + '</sup>'

print("New e:",e)

CalculatorExpression = CalculatorExpression.replace("^"+eo,e)

#Remove multiplication symbols from functions.

for i in range(0, len(Functions)):

CalculatorExpression = CalculatorExpression.replace("\*" + Functions[i][0], Functions[i][0])

CalculatorExpression = CalculatorExpression.replace(Functions[i][0], Functions[i][1])

print("Equation:",Equation)

print("Expression:",CalculatorExpression)

#Replace all variables with algebraic letters.

CalculatorExpression = CalculatorExpression.replace('a', '\N{MATHEMATICAL ITALIC SMALL A}')

CalculatorExpression = CalculatorExpression.replace('b', '\N{MATHEMATICAL ITALIC SMALL B}')

CalculatorExpression = CalculatorExpression.replace('c', '\N{MATHEMATICAL ITALIC SMALL C}')

CalculatorExpression = CalculatorExpression.replace('x', '\N{MATHEMATICAL ITALIC SMALL X}')

CalculatorExpression = CalculatorExpression.replace('y', '\N{MATHEMATICAL ITALIC SMALL Y}')

CalculatorExpression = CalculatorExpression.replace('z', '\N{MATHEMATICAL ITALIC SMALL Z}')

CalculatorExpression = CalculatorExpression.replace('e', '\N{MATHEMATICAL ITALIC SMALL E}')

CalculatorExpression = self.CorrectFontErrors(CalculatorExpression)

print("Formatted Expression:", CalculatorExpression)

print("---------!Formatting-----------")

#Pass back the formatted expression.

return CalculatorExpression

The self.FormatExpression function was a complex one, requiring the input of two similar but different variables to manipulate them simultaneously. It would first check the CalculatorExpression (the version of the expression that would appear to the user) against the Equation (the version of the expression that would be manipulated behind the scenes) to check for consistency.

Finally, it would check the CalculatorExpression against a list of known variables, and would alter them to their mathematical Unicode formats (for example, ‘x’ would become and ‘a’ would become ). After this was done, CalculatorExpression would be returned back to the main program to be displayed to the user.

#Font error correction if on asny other system than Windows 10.

def CorrectFontErrors(self, text):

if platform.system == "Windows" and platform.release in ["10"]:

#Pass.

return text

else:

CharReplace = [

[",", ";"],

["\N{MATHEMATICAL ITALIC SMALL X}", "x"],

["\N{MATHEMATICAL ITALIC SMALL Y}", "y"],

["\N{MATHEMATICAL ITALIC SMALL Z}", "z"],

["\N{MATHEMATICAL ITALIC SMALL A}", "a"],

["\N{MATHEMATICAL ITALIC SMALL B}", "b"],

["\N{MATHEMATICAL ITALIC SMALL C}", "c"],

["\N{MATHEMATICAL BOLD SMALL PI}", "π"],

["\N{MATHEMATICAL BOLD SMALL PHI}", "φ"],

["\N{MATHEMATICAL ITALIC SMALL E}", "e"],

]

#Iterate through CharReplace, replacing all instances of the first string in a button with the second string.

for i in range(0, len(CharReplace)):

text = text.replace(CharReplace[i][0], CharReplace[i][1])

print(text)

return text

This function corrects errors involved in the display of certain buttons that use Unicode symbols. On certain systems, certain Unicode characters would not be recognised and would not be displayed. For example, a button that uses would have its label replaced by a replacement character like “�”. This would make it difficult for the user to understand the functions of those buttons, and would also make it much less simple to use, as there would be much trial and error involved in the use of the program, which would cause much frustration.

This function searches the text given to it for any specific Unicode characters, replacing them with ASCII variants. For example, would be replaced by ‘x’ to prevent this. It would then return the corrected text to the main program.

#Change the properties of can already existing button in a list of widgets, given its index.

def UpdateButton(self, x, y, w, h, msg, List, Index, color, cmd, args = ()):

if msg == self.CalculatorExpression or msg == str(self.Answer) or msg == self.AnswerExpression or color == "equation":

btn = QtGui.QPushButton("", self)

lbl = QtGui.QLabel(msg, self)

else:

btn = QtGui.QPushButton(msg, self)

if args != ():

btn.clicked.connect(lambda: cmd(\*args))

else:

btn.clicked.connect(cmd)

List[Index][0] = btn

btn.move(x, y)

btn.resize(w, h)

if msg == self.CalculatorExpression or msg == str(self.Answer) or msg == self.AnswerExpression:

try:

List[Index][1].hide()

List[Index][1] = lbl

lbl.move(x+16, y)

lbl.resize(w-16, h)

lbl.show()

except:

pass

if color != "default" and color != "equation":

btn.setStyleSheet("background-color:" + color);

#If there is an error in the text...

if "Error" in msg:

#Turn the text red.

if msg == str(self.Answer):

try:

lbl.setStyleSheet("color: red");

except:

pass

else:

btn.setStyleSheet("color: red");

btn.show()

This particular function acts extremely similarly to the self.Button procedure, except instead of adding new buttons to the necessary list, it amends the button in the index given as a parameter.

#Move from the calculator screen to the graoph creation screen with only one curve.

def CalculateGraph(self):

self.expList = [self.Equation]

self.MOVEGraphProperties(self.expList)

This procedure ensures that if electing to create a graph from the Calculator screen, the program will always cause the equation given to be the expression entered into the calculator.

def THREADUpdate(self, char, GiveAnswer, x, y, w, h):

self.UpdateExpression(char, GiveAnswer, x, y, w, h)

This procedure updates the expression in a different thread from the regular processes, to prevent lag after pressing a button.

#Refresh all widgets on a screen, updating them.

def RefreshWidgets(self, List):

print(List)

for i in range(len(List)):

try:

List[i][0].update()

try:

List[i][1].update()

except:

pass

except:

List[i].update()

#Hide all widgets on a screen.

def HideWidgets(self, List):

print(List)

for i in range(len(List)):

try:

List[i][0].hide()

try:

List[i][1].hide()

except:

pass

except:

List[i].hide()

#Show all widgets on a screen.

def ShowWidgets(self, List):

print(List)

for i in range(len(List)):

try:

List[i][0].show()

try:

List[i][1].show()

except:

pass

except:

List[i].show()

These procedures respectively refresh, hide and show all the widgets of a specific screen. They are instrumental in the MOVE procedures to make sure the previous screens are actually cleared.

#Move to the graph creation screen.

def MOVEGraphProperties(self, Curves):

self.CreatingGraph = True

self.HideWidgets(self.CalculatorWidgets)

self.HideWidgets(self.MenuWidgets)

#If the widgets for this screen have not been created.

if self.PropertiesWidgetsCreated == False:

self.Curves = Curves

w = 128

h = 64

s = 64

#Create the curve textbox, and the curve creation button.

self.TextBox(64, self.winH - (h \* 4.5 + 32), self.winW - 128, 32, "", self.PropertiesWidgets)

self.Button((self.winW / 2) - (w / 2), self.winH - (h \* 3.5 + 32), w, h, "Create Curve", self.PropertiesWidgets, True, "default", self.CreateCurve)

#Create the spinbox for the number of curves, and the upper and lower bounds.

self.SpinBox(s, s, s, 32, "Number of Curves", 1, 5, self.PropertiesWidgets)

self.TextBox(self.winW - (s \* 4), s, s, 32, "Lower Bound", self.PropertiesWidgets)

self.TextBox(self.winW - (s \* 2), s, s, 32, "Upper Bound", self.PropertiesWidgets)

#Set the curve text fill to enable curve creation when edited, the button which creates a new curve to enable the creation of the graph when clicked, and the spin box to add a slot to the list when changed.

self.PropertiesWidgets[0][0].textChanged.connect(self.expressionTextChanged)

self.PropertiesWidgets[1][0].clicked.connect(self.graphEnabled)

self.PropertiesWidgets[2][0].valueChanged.connect(self.valueChanged)

#Set the bound text fills to compare bounds every time they are edited.

self.PropertiesWidgets[3][0].textChanged.connect(lambda: self.boundTextChanged("Lower"))

self.PropertiesWidgets[4][0].textChanged.connect(lambda: self.boundTextChanged("Upper"))

#Create the close button and the button to create the graph.

self.Button((self.winW / 2) - (w + 32), self.winH - (h + 64), w, h, "Exit", self.PropertiesWidgets, True, "default", self.MOVEMenu)

self.Button((self.winW / 2) + 32, self.winH - (h + 64), w, h, "Create Graph", self.PropertiesWidgets, True, "default", self.CreateGraph)

#Create the slots for the curves.

for i in range(0, 5):

if i < len(self.Curves):

self.Button(64, self.winH - (h \* 4.5 + (80 + (i \* 32))), self.winW - 128, 32, self.FormatExpression(self.Curves[i], self.Curves[i]), self.CurveSlots, False, "equation", print, self.Curves)

else:

self.Button(64, self.winH - (h \* 4.5 + (80 + (i \* 32))), self.winW - 128, 32, " ", self.CurveSlots, False, "equation", print, self.Curves)

print(" - " + str(self.CurveSlots[i]))

#Show the first slot for the list of curves.

self.CurveSlots[0][0].show()

self.CurveSlots[0][1].show()

print("len:", str(len(self.CurveSlots)))

#Disable the curve and graph creation buttons.

self.PropertiesWidgets[1][0].setEnabled(False)

self.PropertiesWidgets[6][0].setEnabled(False)

#Disable the curve and graph creation buttons if there are no curves.

if Curves != []:

self.Curves = Curves

self.PropertiesWidgets[0][0].setEnabled(False)

self.PropertiesWidgets[2][0].setEnabled(False)

else:

self.PropertiesWidgets[0][0].setEnabled(True)

self.PropertiesWidgets[2][0].setEnabled(True)

#Make sure the program does not create these widgets again.

self.PropertiesWidgetsCreated = True

else:

#Reset the values of all of the widgets.

self.PropertiesWidgets[0][0].setText('')

self.PropertiesWidgets[2][0].setValue(1)

for i in range(0, 5):

if i < len(self.Curves):

self.CurveSlots[i][0].setText(self.FormatExpression("y = " + self.Curves[i], "y = " + self.Curves[i]))

else:

self.CurveSlots[i][0].setText(" ")

self.ShowWidgets(self.PropertiesWidgets)

self.CurveSlots[0][0].show()

self.CurveSlots[0][1].show()

self.PropertiesWidgets[1][0].setEnabled(False)

self.PropertiesWidgets[6][0].setEnabled(False)

if Curves != []:

self.Curves = Curves

self.PropertiesWidgets[0][0].setEnabled(False)

self.PropertiesWidgets[2][0].setEnabled(False)

else:

self.PropertiesWidgets[0][0].setEnabled(True)

self.PropertiesWidgets[2][0].setEnabled(True)

print("C:", Curves)

This subroutine moves the user to the graph creation screen. It creates a button to allow the user to exit the screen at the bottom-left of the window, and a button to create the final graph at the bottom-right. It also creates the spin-box containing the maximum number of curves at the top-left, and the text fills containing the bounds at the top-right of the window. At the centre of the window, it creates the text fill for the input of curves and the buttons that act as the slots for the curves equations.

When the screen is displayed to the user, only the bottom slot is shown where the others are hidden. This can be changed by altering the number in the top-left spin box. In addition, by default the bottom-right button and the button to create a curve are both disabled. This is to prevent the user from creating a graph with no curves or no bounds.

#Create the curve.

def CreateCurve(self):

#If the number of curves are less than maximum number...

if len(self.Curves) < int(self.PropertiesWidgets[2][0].text()):

#List all functions.

functionText = [["sin", "S"], ["cos", "C"], ["tan", "T"], ["log", "L"], ["ln", "E"], ["abs", "A"]]

#Copy the curve from the text fill.

CurveText = self.PropertiesWidgets[0][0].text()

#Replace the proper function text with calculatable letter symbols.

for i in range(0, len(functionText)):

CurveText = CurveText.replace(functionText[i][0], functionText[i][1])

print(CurveText)

#Add the curve to the list.

self.Curves.append(CurveText)

#Delete the curve's expression from the text fill.

self.PropertiesWidgets[0][0].setText("")

print(self.Curves)

#Format all of the curves' expressions.

for i in range(0, 5):

if i < len(self.Curves):

self.CurveSlots[i][1].setText(self.FormatExpression("y = " + self.Curves[i], "y = " + self.Curves[i]))

else:

self.CurveSlots[i][1].setText(" ")

This subroutine formats curves contained inside the text fill the user enters their expressions into to make them suitable for calculation. It makes sure functions that have been inputted can be recognised, even if they are not in the single capital letter format, and also adds them to the list. After this, it clears the text fill to allow a new expression to be entered.

#Move to the graph screen.

def CreateGraph(self):

self.MOVEGraph(self.Curves, self.PropertiesWidgets[3][0].text(), self.PropertiesWidgets[4][0].text())

This procedure reads the contents of the spin boxes that the bounds are entered into, and takes the list of curves, and creates the graph using them.

#Create the curves' coordinates for preparation of the graph's creation.

def MOVEGraph(self, Curves, lBound, uBound):

self.CreatingGraph = False

self.HideWidgets(self.CalculatorWidgets)

self.HideWidgets(self.PropertiesWidgets)

self.HideWidgets(self.CurveSlots)

self.HideWidgets(self.MenuWidgets)

if Curves == []:

for i in range(0, int(input("How many curves do you want to graph? "))):

Curves.append(input("Curve " + str(i+1) + ": "))

else:

if '' in Curves:

Curves = [self.Equation]

print(Curves)

print(self.Curves)

print("Equation:",self.Equation)

print("Curves:",Curves)

Bound = [str(lBound), str(uBound)]

if Bound[0] == "0" and Bound[1] == "0":

Bound[0] = input("Lower Bound: ")

Bound[1] = input("Upper Bound: ")

print("Given Bounds:", Bound)

for i in range(0, len(Bound)):

Bound[i] = Bound[i]

r = int(Calculation\_Process.Input(Bound[i], "Calculation", [['z', 3], ['a', 5], ['b', 10], ['c', 20]]))

Bound[i] = r

print("Calculated Bounds:", Bound)

self.bar = []

for i in range(0, len(Curves)):

self.bar.append(QtGui.QProgressBar(self))

w = 48

h = (self.winH - (w \* 2)) / len(Curves)

self.bar[i].setGeometry(48, (self.winH - 48) - ((i + 1) \* h), w, h)

self.bar[i].setOrientation(QtCore.Qt.Vertical)

self.bar[i].show()

self.bar[i].setValue(self.percent)

GraphQueue = queue.Queue()

self.THREADGraph = threading.Thread(target = self.Graph, args = (Curves, Bound, GraphQueue))

self.THREADGraph.start()

while self.percent < 100 and i < len(Curves):

QtGui.QApplication.processEvents()

pass

values = GraphQueue.get()

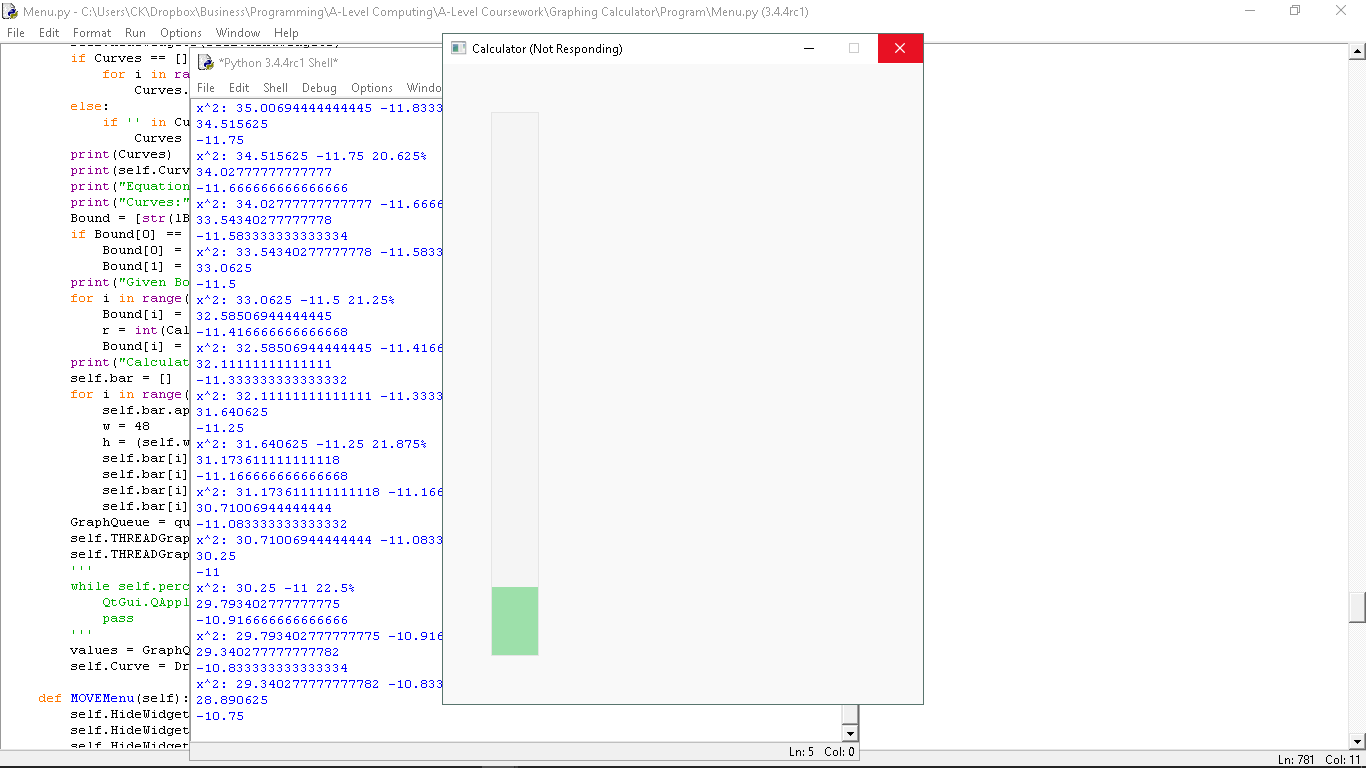
self.Curve = DrawCurve(values[0], values[1], Bound, 0, 0, self.winW, self.winH, self)

This subroutine processes the determination of the graph’s coordinates whilst providing a visual aid of its progress using a progress bar. These bars are vertical in orientation, and fill from the bottom up. For every set of coordinates the program creates, the bars will fill up by a percentage equal to . The process for them being filled uses the subroutine:

def SetBar(self, n, e):

self.percent += 100 / self.winW

self.bar[n].setValue(self.percent)

The subroutine creates coordinates by passing the equation of the curve into the calculation process subroutine, with the x-coordinate being determined by using the x-coordinate on the screen, and factoring it into the difference between the bounds. The result becomes the corresponding y-coordinate to form the coordinate sets. This process is iterated as the x-coordinate on the screen increases, creating a series of coordinates that are stored in a list. It then updates the program, making sure it doesn’t hang, which was a problem that did occur in previous iterations.

def Graph(self, Expression, Bound, Queue):

Points = []

Gradients = []

for i in range(0, len(Expression)):

self.percent = 0

Points.append([])

dydx = Calculation\_Process.Input(Expression[i], "Differentiation", [["x"], ["y"], ["z"], ["a"], ["b"], ["c"]])

print("y:", Expression[i])

print("dy/dx:", dydx)

print("")

for x in range(0, self.winW+1):

print(Expression[i])

y = Calculation\_Process.Input(Expression[i], "Calculation",

[

['x', Bound[0]+((Bound[1]-Bound[0])\*(x/self.winW))],

['z', 3], ['a', 5], ['b', 10], ['c', 20],

])

if y != "ERROR" and "Error" not in dydx:

print(dydx)

m = Calculation\_Process.Input(str(dydx), "Calculation",

[

['x', Bound[0]+((Bound[1]-Bound[0])\*(x/self.winW))],

['z', 3], ['a', 5], ['b', 10], ['c', 20],

])

print(Expression[i] + ":", str(y), str(m), str(round(self.percent, 3)) + "%")

Gradients.append(m)

else:

print(Expression[i] + ":", str(y), str(round(self.percent, 3)) + "%")

print(Bound[0]+((Bound[1]-Bound[0])\*(x/self.winW)))

print("")

Points[i].append([x, y, [Bound[0]+((Bound[1]-Bound[0])\*(x/self.winW))]])

self.SetBar(i, Expression)

for i in range(0, len(Expression)):

self.bar[i].hide()

print("")

Queue.put((Points, Gradients))

btn = self.Button(0, 0, 48, 32, "CLOSE", self.GraphWidgets, True, "default", self.MOVEMenu)

This procedure was performed in a separate thread to allow the creation of the graph’s y-values and the filling up of the bar to occur simultaneously. This would speed the program up, as there would be much less delay between coordinate creation and the drawing of the graph.

It creates the x-values using the formula , where: , , , and . This ensures that the x-coordinate is clamped between the upper bound and lower bound.

class DrawCurve(QtGui.QWidget):

Points = []

Bound = [0, 0]

winX = 48

winY = 48

winW = 640

winH = 480

def \_\_init\_\_(self, Points, Gradients, Bound, winX, winY, winW, winH, parent):

super(DrawCurve, self).\_\_init\_\_(parent)

self.setGeometry(winX, winY, winW, winH)

self.setWindowTitle("Graph")

self.winW = winW

self.winH = winH

self.winX = winX

self.winY = winY

self.Points = Points

self.Bound = Bound

self.initUI()

self.show()

This class manages the drawing of the curve, calling self.initUI() immediately. It then shows itself to make sure the graph can actually be seen.

def initUI(self):

btn = QtGui.QPushButton('Exit', self)

btn.clicked.connect(self.appRestart)

btn.resize(128, 64)

btn.move(16, 16)

btn.show()

This subprocess creates the exit button, which closes the window, setting its size and position and making it visible.

def appRestart(self):

#os.system("Menu.py 1")

new = subprocess.Popen(['Menu.py', 'htmlfilename.htm'], shell = True)

sys.exit()

This procedure is called when the exit button created in self.initUI is clicked. It closes the window and creates a new menu window at the same location.

def paintEvent(self, e):

print("...")

print("")

print("Bound:", self.Bound)

Line = QtGui.QPainter()

Line.begin(self)

axis\_pen = QtGui.QPen(QtCore.Qt.black, 1, QtCore.Qt.SolidLine)

Line.setPen(axis\_pen)

y\_values = []

for i in range(0, len(self.Points)):

for ii in range(0, len(self.Points[i])):

if "Error" not in str(self.Points[i][ii][1]):

if isinstance(self.Points[i][ii][1], str):

if sum(i.count(".") for i in str(self.Points[i][ii][1])) > 0:

y\_values.append(float(self.Points[i][ii][1]))

else:

y\_values.append(int(self.Points[i][ii][1]))

else:

y\_values.append(self.Points[i][ii][1])

min\_y = min(y\_values)

max\_y = max(y\_values)

print("Min:", min\_y)

print("Max:", max\_y)

if self.Bound[0] <= 0 and self.Bound[1] >= 0:

i = (0 - self.Bound[0]) / (self.Bound[1] - self.Bound[0]) \* self.winW

print("Y Axis:", i)

Line.drawLine(i, 0, i, self.winH)

x\_axis = ((max\_y)/(max\_y-min\_y))\*self.winH

print("X Axis:", x\_axis)

Line.drawLine(0, x\_axis, self.winW, x\_axis)

curve\_pen = QtGui.QPen(QtCore.Qt.black, 2, QtCore.Qt.SolidLine)

Line.setPen(curve\_pen)

print('\n' \* 99)

for i in range(0, len(self.Points)):

for ii in range(0, len(self.Points[i])-1):

print(self.Points[i])

try:

Line.drawLine(ii\*(self.winW/(len(self.Points[i]))), ((max\_y-self.Points[i][ii][1])/(max\_y-min\_y))\*self.winH, (ii+1)\*(self.winW/(len(self.Points[i]))), ((max\_y-self.Points[i][ii+1][1])/(max\_y-min\_y))\*self.winH)

except:

pass

self.show()

Line.end()

This subroutine uses PyQt’s QPainter event to draw the graph using a culmination of connected lines. It starts at the left edge of the screen and moves to the right, drawing a line from the respective coordinate set to the next coordinate set. This creates a multitude of straight lines to create the illusion of a curved line. The self.paintEvent occurs every tick, making the program continuously draw the graph over again.

#Import all the necessary libraries and modules.

import copy

import math

import Variables

Copy needed to be imported to

def Input(Expression, Mode, var):

print("Input your equation. Always space the elements amongst the operators.")

#Input equation.

print(Expression)

if Expression[0] == "-":

Expression = "0" + Expression

Replacements = [["--", "+"], ["++", "+"], ["+-", "-"], ["-+", "-"]]

for i in range(0, len(Replacements)):

try:

Expression = Expression.replace(Replacements[i][0], Replacements[i][1])

except:

pass;

print(Expression)

#Split the expression into the individual elements.

expList = Expression.split()

At the start of the Input function, it checks if the expression starts with a negative number and reformats it to This makes sure that unbracketed negative numbers and bracketed negative numbers differentiate themselves, for example in the case of and . After this, it would replace the double operators with singular operators, following the double negative rules. Finally, it would split the expression into a list instead of keeping it as an immutable string.

#Split the elements in a personal list.

for i in range(0, len(expList)):

expList[i] = list(expList[i])

Any still combined elements would then be split into a nested list, making the entire list partially two-dimensional. This is actually an archaic relic from an older version of the program, when there was no Graphical User Interface and the calculator just consisted of a terminal-based printed interface, as the user would enter in spaces, guided by the prompt “Always space the elements amongst the operators.” Due to the way that list(string) works, spaces would be eliminated and would separate phrases broken by them into separate lists of their own.

#Add preset variables.

for i in range(0, len(Variables.presetVariables)):

var.append([Variables.presetVariables[i].Alias, Variables.presetVariables[i].Value])

#Create new variables.

try:

if Mode == "Calculation":

for v\_i in range(0, len(var)):

for i in range(0, len(expList)):

for ii in range(0, len(expList[i])):

print(i, ii, v\_i)

print(expList[i][ii], var[v\_i])

if expList[i][ii] == var[v\_i][0]:

expList[i][ii] = var[v\_i][1]

except:

pass;

This section of code substitutes variable values into the expression, checking if it is to be calculated, and iterates through the variable list, searching for instances of characters that notate variables, and substituting them for the necessary values.

for i in range(len(expList)):

for j in range(len(expList[i])):

expList[i][j] = str(expList[i][j])

This code section turns all integer or float values in the list into strings.

print("Combination:")

#Combine the individual figures into numbers.

for i in range(0, len(expList)):

#Search through the personal lists.

o = 0

for j in range(len(expList[i])):

print(expList[i][j-o], "(" + str(i) + ",", str(j) + ",", str(o) + ")")

NegativeForebears = ["(", "\*", "/", "^"]

if any(char.isdigit() for char in expList[i][j-o]) == True or (expList[i][j-o] == "-" and j == 0) or (expList[i][j-o-1] in NegativeForebears and expList[i][j-o] == "-"):

if j-o+1 < len(expList[i]):

if expList[i][j-o+1] in {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '.'}:

expList[i][j-o] = str(expList[i][j-o]) + str(expList[i][j-o+1])

del expList[i][j-o+1]

print(expList)

o += 1

else:

if j-o == len(expList[i]) - 1:

break;

This code section combines each of the digit characters that make up integers into a single numerical string, by detecting whether the currently selected character is composed of digits and the next character is also a digit, then appending the next digit character to the current string. In addition, if the current character is a minus sign and the previous character was in the list of NegativeForebears, it will combine the characters any to create negative numbers.

#Convert all numbers in the list to integers.

for i in range(0, len(expList)):

for j in range(0, len(expList[i])):

try:

if sum(i.count(".") for i in str(expList[i][j])) > 0:

expList[i][j] = float(expList[i][j])

print("FLT:", expList)

else:

expList[i][j] = int(expList[i][j])

print("INT:", expList)

except ValueError:

pass

This code section converts all numerical strings in the list into either integers or floats, depending on whether the string contains a dot or not.

if Mode == "Calculation":

expList = Brackets(expList)

print("l:", expList)

elif Mode == "Differentiation":

v = []

for i in range(0, len(var)):

v.append(var[i])

expList = Differentiation(expList, v)

try:

if (isinstance(expList, complex) or ((any(char.isdigit() for char in expList) == False) and "Error" not in expList)):

print("[ERROR]:", expList, "is not a real number.")

expList = "Complex Error"

except:

pass

if isinstance(expList, float):

if expList.is\_integer():

expList = int(expList)

print(expList)

return expList

This code section checks if the expression is to be calculated upon or differentiated, and then calls Brackets() or Differentiate() respectively. After calculating or differentiating, it then checks if the result is complex or a string. If it is, then it will call a complex error.

After this, it makes sure that whole numbers that are stored as floats are converted into integers and returns the final result.

#B - Brackets

def Brackets(e):

e\_temp = e

for x in range(len(e)):

for y in range(len(e[x])):

if isinstance(e[x][y], int) or isinstance(e[x][y], float):

#print("1a. PRV:", e)

e[x][y] = str(e[x][y])

#print("1b. PST:", e)

for x in range(len(e\_temp)):

for y in range(len(e\_temp[x])):

if isinstance(e\_temp[x][y], int) or isinstance(e[x][y], float):

#print("2a. PRV:", e)

e\_temp[x][y] = str(e\_temp[x][y])

#print("2b. PST:", e)

digits = "0123456789."

for x in range(len(e\_temp)):

y = 0

while y < len(e\_temp[x]):

digits.split()

if y > 0:

operators = "+-\*/^()"

if e\_temp[x][y] not in operators:

if y < len(e\_temp[x]) - 1:

digits.join('')

if e\_temp[x][y+1] in digits:

e\_temp[x][y] = e\_temp[x][y] + e\_temp[x][y+1]

del e\_temp[x][y+1]

else:

try:

if sum(i.count(".") for i in e\_temp[x][y]) > 0:

e\_temp[x][y] = float(e\_temp[x][y])

else:

e\_temp[x][y] = int(e\_temp[x][y])

except:

pass

y += 1

else:

try:

if sum(i.count(".") for i in e\_temp[x][y]) > 0:

e\_temp[x][y] = float(e\_temp[x][y])

else:

e\_temp[x][y] = int(e\_temp[x][y])

except:

pass

y += 1

else:

y += 1

else:

operators = "+\*/^()"

#print("0a. STR:", e\_temp)

if e\_temp[x][y] not in operators:

if y < len(e\_temp[x]) - 1:

#print("0b. STR:", e\_temp)

digits.join('')

if e\_temp[x][y+1] in digits:

#print("0c. STR:", e\_temp)

e\_temp[x][y] = e\_temp[x][y] + e\_temp[x][y+1]

del e\_temp[x][y+1]

else:

#print("0d. STR:", e\_temp)

try:

if sum(i.count(".") for i in e\_temp[x][y]) > 0:

e\_temp[x][y] = float(e\_temp[x][y])

else:

e\_temp[x][y] = int(e\_temp[x][y])

except:

pass

y += 1

else:

try:

#print("0ei. STR:", e\_temp)

if sum(i.count(".") for i in e\_temp[x][y]) > 0:

e\_temp[x][y] = float(e\_temp[x][y])

else:

e\_temp[x][y] = int(e\_temp[x][y])

#print("0e2. STR:", e\_temp)

except:

pass

y += 1

else:

y += 1

#print("1. STR:", e\_temp)

e = e\_temp

if sum(i.count("(") for i in e\_temp) == sum(i.count(")") for i in e\_temp):

if sum(i.count("(") for i in e\_temp) > 0:

BracketBegin = [-1, -1]

BracketEnd = [-1, -1]

while sum(i.count(")") for i in e\_temp) > 0:

#print("")

for x in range(len(e\_temp)):

#print("Length:", len(e\_temp[x]))

#print("y:", end = " ")

for y in range(len(e\_temp[x])):

#print(y, end = "")

if e\_temp[x][y] == "(":

BracketBegin = [x, y]

#print("B", end = "")

elif e\_temp[x][y] == ")":

BracketEnd = [x, y]

#print("E", end = "")

break;

#print("", end = " ")

#print("\n" \* 0)

#print("[" + str(BracketBegin[0]), str(BracketBegin[1]) + "] [" + str(BracketEnd[0]), str(BracketEnd[1]) + "]")

#print(e\_temp[BracketBegin[0]][BracketBegin[1]] + "..." + e\_temp[BracketEnd[0]][BracketEnd[1]])

eu = e\_temp[BracketBegin[0]][BracketBegin[1]:BracketEnd[1]+1]

#print("1. eu:",eu)

eu.remove("(")

#print("1a. eu:",eu)

eu.remove(")")

#print("1b. eu:",eu)

eu = [eu]

#print("2. eu:",eu)

#print("3. e\_temp:",e\_temp)

eo = copy.deepcopy(eu)

#print("4. eu:",eu)

#print("5. eo:",eo)

for i in range(0, len(eo[0])):

if isinstance(eo[0][i], int):

eo[0][i] = str(eo[0][i])

#print("6. eu:",eu)

#print("7. eo:",eo)

try:

eo = "".join(eo[0])

except:

pass;

#print("8. ej:",eo)

res = eu[0][0]

#print(res)

try:

res = Calculation(eu)

except:

pass;

#print("res:", res)

del e\_temp[0][BracketBegin[1]:BracketEnd[1]+1]

e\_temp[0].insert(BracketBegin[1], res)

else:

pass;

#print("brac:", e\_temp)

try:

e = Functions(e\_temp)

except:

return "Math Domain Error"

#print("func:", e)

res = Calculation(e)

#print("calc:", res)

return res

else:

#print("[ERROR]: Invalid expression - Imbalanced brackets.")

return "Parse Error"

This function controls the manipulation of brackets and bracketed expressions. If the number of open brackets and closed brackets are equal, the program will jump to the first instance of an open bracket in the list, recording its position. Then, it would iterate though the list, searching for an instance of a closed bracket. If it encounters an open bracket, it will amend the coordinates of the open bracket to the position of the new one. This would ensure that only the highest level set of brackets would be operated on. Once it finds a closed bracket, it would stop iterating and would record its position separately. It would then make a complete copy of the list using copy.deepcopy, and would take the copy and cut it down to the expression inside the brackets, eliminating the brackets themselves. It would then operate on them using Calculator\_Process.Calculate(), inputting the result into the original list whilst replacing the brackets as well. It would continue this until no more open brackets remained, and would return the final result to the Input() function. If the number of open brackets and closed brackets were unequal, it would return a “Parse Error” instead.

#Calculate

def Calculation(e):

print("e:",e)

print("pre-res:", e)

Operators = [

[Indices, O\_Indices, "^", "I"],

[Division, O\_Division, "/", "D"],

[Multiplication, O\_Multiplication, "\*", "M"],

[Addition, O\_Addition, "+", "A"],

[Subtraction, O\_Subtraction, "-", "S"],

]

for m in range(0, 2):

for n in range(0, 5):

if m == 0:

if sum(i.count(Operators[n][2]) for i in e) > 0 and e[0][0] != "ERROR":

print(Operators[n][3])

e = Operators[n][0](e)

print(Operators[n][3] + ":", e)

elif e[0][0] == "ERROR":

return e[0][1]

else:

e = Operators[n][1](e)

if e[0][0] == "ERROR" or isinstance(e[0][0], str):

return e[0][1]

else:

res = e

print("RTRN:", e[0][0])

return e[0][0]

This subroutine uses the fact that one can store function calls in lists, using this fact to iterate through each operator, calling each respective function to move through the BIDMAS operations. At each iteration, if the first element of the expression’s list is “ERROR”, the program will return the second element, which would always provide a descriptor of the error, for example “Syntax Error” or “Divide By Zero Error”. If this doesn’t happen, the first element would be returned after the entire process is finished.

#I - Indices

def Indices(e):

#Inside

for i in range(0, len(e)):

while e[i].count("^") > 0:

try:

ii = e[i].index('^')

e[i][ii] = e[i][ii-1] \*\* e[i][ii+1]

del e[i][ii-1]

del e[i][ii]

except:

e = [["ERROR", "Syntax Error"]]

return e

def O\_Indices(e):

#D - Division

def Division(e):

#Inside

for i in range(0, len(e)):

while e[i].count("/") > 0:

try:

ii = e[i].index('/')

if e[i][ii+1] != 0:

e[i][ii] = e[i][ii-1] / e[i][ii+1]

del e[i][ii-1]

del e[i][ii]

else:

e = [["ERROR", "Divide by Zero Error"]]

except:

e = [["ERROR", "Syntax Error"]]

return e

#M - Multiplication

def Multiplication(e):

#Inside

for i in range(0, len(e)):

while e[i].count("\*") > 0:

try:

ii = e[i].index('\*')

if isinstance(e[i][ii+1], int) or isinstance(e[i][ii+1], float):

e[i][ii] = e[i][ii-1] \* e[i][ii+1]

del e[i][ii-1]

del e[i][ii]

else:

return [["ERROR", "Syntax Error"]]

except:

e = [["ERROR", "Syntax Error"]]

return e

#A - Addition

def Addition(e):

#Inside

for i in range(0, len(e)):

while e[i].count("+") > 0:

try:

ii = e[i].index('+')

e[i][ii] = e[i][ii-1] + e[i][ii+1]

del e[i][ii-1]

del e[i][ii]

except:

e = [["ERROR", "Syntax Error"]]

return e

#S - Subtraction

def Subtraction(e):

#Inside

for i in range(0, len(e)):

while e[i].count("-") > 0:

try:

ii = e[i].index('-')

e[i][ii] = e[i][ii-1] - e[i][ii+1]

del e[i][ii-1]

del e[i][ii]

except:

e = [["ERROR", "Syntax Error"]]

return e

These ten functions each jump to the first instance of their respective operators, performing powering, division, addition or subtraction respectively between the integers or floats directly to their lefts and rights. Then, the values to the right of the operators and the operators themselves are eliminated, leaving only the result that was substituted back into the first value. If this entire process isn’t possible, the system will return a “Syntax Error”, else they will return the altered list.

#F - Functions

def Functions(e):

print("F")

#Inside

Functions = [

["S", math.radians, math.sin],

["C", math.radians, math.cos],

["T", math.radians, math.tan],

["L", math.log10],

["E", math.log1p],

["A", abs],

["!", math.factorial],

]

print(e)

for i in range(0, len(e)):

for f in range(0, len(Functions)):

print(Functions[f][0])

print(e)

while e[i].count(Functions[f][0]) > 0:

ii = e[i].index(Functions[f][0])

for p in range(1, len(Functions[f])):

if Functions[f][0] == "T":

if (e[i][ii+1] - 90) % 90 == 0:

return [["ERROR", "Complex Error"]]

if Functions[f][0] == "E":

e[i][ii+1] = e[i][ii+1] - 1

try:

if Functions[f][0] == "!":

e[i][ii-1] = Functions[f][p](e[i][ii-1])

else:

e[i][ii+1] = Functions[f][p](e[i][ii+1])

except:

pass;

print(str(p) + (" " \* p), end = "")

print(e[i][ii+1], end = "\n")

print(e)

del e[i][ii]

return e

This subroutine acts earlier than the Calculate() function, and utilises the same technique as it as well, storing the function calls in a list to use them later. It stores the codes for the functions in the first index of each of the nested lists and the various functions in subsequent indexes, in the order they are to be called. For example, sin() would use the symbol code “S”, and would call math.radians to convert the value in degrees into radians, and then calls math.sin to pass the new value through the sine wave function. This would be done for each of the functions in turn, with special exceptions programmed through if…then selector statements in the loop.