Analysis

The problem

Graphing calculators are legal in a level maths and further maths exams, however their price makes them unaffordable for many students, however, they provide a massive advantage for those students who are lucky enough to own these devices. This is because they are capable of removing much of the memorisation needed of different graphs and their shapes and properties in the exam, and can give numerical solutions to many graph based questions in order to check graph based questions. Graphing calculator emulators are available from casio, one of the companies that make graphing calculators for the exam, however these require expensive licenses. Apart from being useful In exams, I hope that this software may be useful in many stem classrooms as an all in one replacement for many different applications. Teachers can show their calculator working out live on the board, and plot any subsequent graphs as needed, making it an easy to follow experience for the students, and painless for teachers. That being said, my target audience is students if this program was to be approved by the exam boards, and teachers to assist with their class teaching by centralising all the resources they need in one place.

Products that graph already include Desmos:

Chart

Description automatically generated

This website is an amazing tool for looking at and understanding functions encountered while studying, it allows the user to see the graph and also see key points while mousing over it. However, it doesn’t have traditional calculator feel or looks, so is slightly less applicable when doing demonstrations, in addition, it lacks some features like ability to solve quadratics for analytical solutions. this means that roots of quadratics will be given as imprecise decimals, rather than surds like a regular calculator. This is insufficient for some problems in the a level. Further to this point, statistics are lacking from Desmos.

Casio official emulator

Need to get this when home

Research:

I have decided to great the program in python with pyqt5 providing the front end. I did this because I am familiar with the programming language and matplotlib, the library I will use to plot graphs and render tex (code to render mathematical notation) in order to make the app user friendly and easy to use. It will also be use this to plot the graphs when the user goes to that screen and enters an equation to be plotted. Another consideration is that the program should evaluate the calculations entered in accordance with BIDMAS order of operations. This is very important as it will be key functionality of the app. Upon researching this problem further, I discovered this can be done by converting the strings to post fix in the shunting yard algorithm. This ensures that all operations go in the correct order.

Need to do

Design:

Pseudo code for complex parts of the program:

**System used to compute mathematical expressions inputted by the user:**

It is important that the equations are computed according to BIDMAS order of operations (Brackets, Indices, Division, Multiplication, Addition and subtractions) for this reason the expressions need to be parsed to be in the correct form. There were many algorithms I researched that could do this, but I decided on reverse polish notation and the shunting yard algorithm, because they were the most simple, to implement because it is designed only for mathematical expressions, whereas others, like lexical trees, are designed to translate High Level Languages and therefore need to be more adaptable.

The first challenge is to tokenise the data inputted by the user

PSEUDO CODE:

SUBROUTINE tokenise(exp)

exp🡨exp.replace(‘ ‘,’’) //replace the white space in the expression

precedence🡨{'+': 0, '-': 0, '\*': 5, '/': 5, '%': 10, '^': 15, 'sin': 20, 'cos': 20, 'tan': 20, 'acos': 20, 'asin': 20, 'atan': 20, 'ln': 20, 'log': 20}

newExp🡨’’

tokens🡨['+', '-', '\*', '/', '%', '^', 'sin', 'cos', 'tan', 'acos', 'asin', 'atan', 'ln', 'log']

FOR token IN tokens

FOR char in exp

IF char = token

newExp+=’ {char} ‘

ELSE

newExp+=char

ENDIF

ENDFOR

Exp🡨newExp

newExp🡨’’

ENDFOR

Tokenised🡨exp.split(‘ ‘)

newTokenised🡨[]

FOR i 🡨0 TO LEN(tokenised)

t🡨tokenised[i]

IF t!=’’

newTokenised.append(t)

ENDIF

ENDFOR

tokenised🡨newTokenised

newTokenised🡨[]

// at this point minus signs kept crashing the program, for example -5- -6 would break it all

skip🡨False

minus🡨False

IF tokenised[0] ==’-‘

//minus signs aren’t a problem if the computer can interpret them as negative numbers, therefore they need to be made one element in the array as to not cause issues in the shunting yard algorithm misunderstanding them

tokenised.pop(0)

t=tokenised.pop(0)

tokenise.insert(0,’-‘+t)

ENDIF

FOR i🡨0 TO LEN(tokenised[:-1])

T🡨tokenised[i]

IF skip = True

Skip 🡨 False

CONTINUE

ELIF t=’-‘ AND tokenise[i+1]

newTokenised.append(t)

skip🡨True

minus🡨true

ELIF t NOT IN tokens AND minus = true

newTokenised.append(‘-‘+t)

minus🡨false

ELSE

newTokenised.append(t)

ENDIF

ENDFOR

IF newTokenised[-1] == ‘-‘ AND newTokenised[-2] IN tokens

newTokenised.pop()

newToeknised.append(‘-‘ +tokenised[-1])

ELSE

newTokenised.append(tokenised[-1]

return newTokenised

[Algorithm based on and adapted from this implementation in Java <https://www.andreinc.net/2010/10/05/converting-infix-to-rpn-shunting-yard-algorithm> accessed 10/11/22]

SUBROUTINE createRPN(exp)

tokens🡨['+', '-', '\*', '/', '%', '^', 'sin', 'cos', 'tan', 'acos', 'asin', 'atan', 'ln', 'log']

tokenised🡨tokenise(exp)

s🡨Stack()

output🡨[]

FOR t IN tokenised  
 if t =’(‘

s.push(t)

ELIF t =’(‘

WHILE s.peak()!=’(‘

x=s.pop()

if x!=’(‘

output.append(x)

ENDIF

ENDWHILE

s.pop()

ELIF t IN tokens

WHILE not s.isEmpty() AND s.peak() IN precdecence.keys()

IF precedence[t]<=precedence[s.peak()]

X🡨s.pop()

Ouput.append(x)

ELSE

BREAK

ENDIF

s.push(t)

ENDWHILE

ELSE

Output.append(t)

ENDIF

ENDFOR

WHILE s.isEmpty()=False

X🡨s.pop()

Output.append(x)

ENDWHILE

RETURN output

These algorithm’s job is to turn the human entered strings to an array that the computer can easily parse into infix notation. Mathematical functions with higher precedence gets done first, this is quantified by the dictionary. A dictionary was the best datastructure to use for this task because the key is the element being checked, and the value is its precedence value, which is a numerical value which is easy to compare with the precedence value that went before

**System used to find solutions for systems of simultaneous equations entered by the user inputted by the user:**

for the a level, students only need to solve up to 3 equation simultaneous equations, so I added this functionality. I used matrices to achieve this. Bellow are summarised approaches and the pseudo code required to make the computer calculate them

this is the base class that defines the other matrix classes

CLASS Matrix

SUBROUTINE \_\_init\_\_(self)

Self.data=[]

SUBROUTINE addEquation(self,eq)

Self.data.append(eq)

SUBROUTINE dot(self,mat2)

Result =[]

FOR i🡨0 TO LEN(self.data)

Result.append([])

Row🡨self.data[i]

For j🡨0 TO LEN(mat2.data[0])

Col🡨[mat[j] for mat IN mat2.data]

Total🡨0

FOR v1,v2 IN ZIP(row,col)

Total🡨total +v1\*v2

ENDFOR

Result[i].append(total)

ENDFOR

ENDFOR

RETURN results

**system of 2 equations:**

these require a 2x2 matrix

steps:

1. Inverse the matrix (m^-1)
   1. Find determinant
2. Dot the inverse matrix with the constants in the equations

CLASS Matrix2x2(Matrix)

SUBROUTINE \_\_init\_\_(self)

SUPER().\_\_init\_\_()

SUBROUTINE getDeterminatant(self)

A🡨 self.data[0][0]

B🡨 self.data[0][-1]

C🡨self.data[1][0]

D🡨 self.data[1][-1]

return a\*d - b\*c # formula of a determinant

SUBROUTINE invert(self):

det 🡨self.getDeterminant()

a 🡨self.data[0][0]

b 🡨= -1\*self.data[0][-1]

c 🡨-1\*self.data[1][0]

d 🡨self.data[1][-1]

self.data[0][0] 🡨d

self.data[-1][-1] 🡨a

self.data[0][-1] 🡨b

self.data[1][0] 🡨c

self.scalar(1/det)

**solve system of 3 equations:**

uses 3x3 matrices

class Matrix3x3(Matrix):

SUBROUTINE \_\_init\_\_(self):

super().\_\_init\_\_()

SUBROUTINE getDeterminant(self)

element <--self.data[0][0]

# first minor

minor <--matrix2x2()

minor.addEquation([self.data[1][1], self.data[1][-1]])

minor.addEquation([self.data[2][1], self.data[2][-1]])

a <--minor.getDeterminant()\*element

# second minor

element <--self.data[0][1]

minor <--matrix2x2()

minor.addEquation([self.data[1][0], self.data[1][-1]])

minor.addEquation([self.data[2][0], self.data[2][-1]])

b <--minor.getDeterminant()\*element

# third minor

element <--self.data[0][2]

minor <--matrix2x2()

minor.addEquation([self.data[1][0], self.data[1][1]])

minor.addEquation([self.data[2][0], self.data[2][1]])

c <--minor.getDeterminant()\*element

return a-b+c

SUBROUTINE invert(self):

# step 1:

# find deteriment

det <--self.getDeterminant()

# step 2

# matrix of minors

m33 <--matrix3x3()

for x 🡨0 TO 3:

currentRow = []

for y 🡨0 TO 3:

if x = 0 THEN

x1 = 1

x2 = 2

if x = 1 THEN

x1 = 0

x2 = 2

if x = 2 THEN

x1 = 0

x2 = 1

if y = 0 THEN

y1 = 1

y2 = 2

if y = 1 THEN

y1 = 0

y2 = 2

if y = 2 THEN

y1 = 0

y2 = 1

m <--Matrix2x2()

m.addEquation([self.data[x1][y1], self.data[x2][y1]])

m.addEquation([self.data[x1][y2], self.data[x2][y2]])

# step 3

# +-+ matrix

if (x+y) % 2 = 1 THEN

# all even sums of x,y make their element negative

currentRow.append(-1\*m.getDeterminant())

else

# otherwise it is just a positive element

currentRow.append(m.getDeterminant())

ENDIF

m33.addEquation(currentRow)

currentRow = []

ENDFOR

self.data <--m33.data

# step 4

# transpose the matrix

transposedMatrix = [[], [], []]

for x🡨 0 TO 3:

row <--self.data[x]

transposedMatrix[0].append(row[0])

transposedMatrix[1].append(row[1])

transposedMatrix[2].append(row[2])

self.data <--transposedMatrix

# step 5:

# finally 1/det \* transposed

self.scalar(1/det)

res <--matrix3x3()

for i 🡨0 TO 3

res.addEquation(transposedMatrix[i])

ENDFOR

return res

**System used to compute the answers to the expressions entered by the user inputted by the user:**

this function computes RPN with values

class Calculate:

SUBROUTINE \_\_init\_\_(self, config: dict, vars=None)

if vars = None THEN

self.vars<--{}

else THEN

self.vars<--vars

ENDIF

SUBROUTINE computeExpression(self, expression)

r<--RPN(expression, self.vars)

return self.computeRPN(r)

SUBROUTINE computeRPN(self, RPN)

paramStack<--Stack()

for item in RPN.rpn THEN

# print(item)

if item not in precedence THEN

paramStack.push(item)

else

# need to computer whatever operator is in here

if item != 'sin' and item != 'cos' and item != 'tan' THEN

x<--paramStack.pop()

y<--paramStack.pop()

res<--self.computeOp(x, y, item)

paramStack.push(res)

else

x<--paramStack.pop()

res<--self.computeTrig(item, x)

paramStack.push(res)

ENDIF

ENDIF

return paramStack.pop()

SUBROUTINE findInput(self, op1)

# gets value out of saved vars if it is not a number

try

op1<--float(op1)

except Exception

if op1 in self.vars THEN

op1<--float(self.vars[op1])

elif op1.find('x')!=-1 THEN

i=op1.find('x')

coef<--float(op1[:i])

x=self.findInput('x')

op1=x\*coef

else

op1<--None # error, the var isnt defined

ENDIF

return op1

SUBROUTINE computeOp(self, op1, op2, symb)

op2<--self.findInput(op2)

if symb = '+' THEN

return op1+op2

elif symb = '\*' THEN

return op1\*op2

elif symb = '-' THEN

return op2-op1

elif symb = '/' THEN

return op2/op1

elif symb = '^' THEN

return op2\*\*op1

ENDIF

SUBROUTINE computeTrig(self, func, x)

x<--self.findInput(x)

if func = 'sin' THEN

return math.sin(float(x))

elif func = 'cos' THEN

return math.cos(float(x))

else:

return math.tan(float(x))

ENDIF

SUBROUTINE retrieveVariable(self, var)

val<--utils.readSetting(var)

return val

Diagram

Description automatically generated