# EE30241: Robotics and Autonomous Systems

**Ahizechi Nwankwo (an777)**

Lab Report:

Exercise 1:



Exercise 2:



Exercise 3:

Exercise 1:

Introduction:

For exercise 1, we were tasked with constructing a program that acts as a processing unit (a simple calculator) using C++ or Python, and a "requester" program which will send and receive messages to request of calculations and respond with results. The steps followed are shown below:

A. Create a server that can handle at least a client.
B. The server will function as calculator (operations min: = + - * / ^ sqrt).
C. Implement a client using C# (or Python) that can receive commands from a user.
D. Receive and send commands.
E. Use the GUI process to send commands to the "calculator process".
F. Send back the result and display on GUI.
G. The connection and performances need to be good and reliable, and transparency should be provided (e.g., log of communication -> saved on a txt file).

This exercise was completed using Python in Microsoft Visual Studio 2019.

Explanation:

We were provided with code that connects a client to a server; it was only necessary to restructure the code so that the connection between the client and server was continuous. Once this was done, a flow diagram, shown in figure 1, was created, to establish the chain of events that should occur once the client and server have connected.

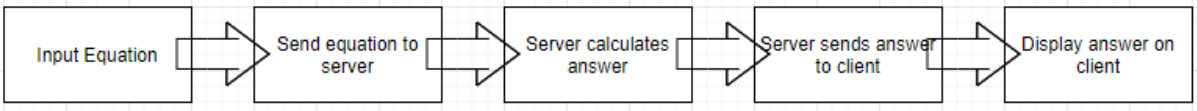


*Figure 1: Flow diagram for the Calculator.*

The user needed to be able to input the operations shown in step B, however both "^" and "sqrt" are not standard functions within python, so it was necessary to develop a way for the user to input and send these operations without turning up an error. This was achieved using .replace():

```
m = m.replace("^", "**")
m = m.replace("sqrt", "math.sqrt")
```

*Figure 2: Code used to replace the input with its respective function.*

The replace() method replaces a specified phrase with another specified phrase. In this case, the inputs "^" and "sqrt" will be replaced with "**" and "math.sqrt" server side, meaning that the operations specified in step B can be used without affecting the outcome. To utilise this method, it was necessary to import the math module so that "math.sqrt" and any other additional functions could work.

```
m = m.replace("^", "**")
m = m.replace("cbrt", "**(1/3)")
m = m.replace("sqrt", "math.sqrt")
m = m.replace("!", "math.factorial")
m = m.replace("e", "math.exp")
m = m.replace("log2", "math.log2")
m = m.replace("log10", "math.log10")
```

*Figure 3: Additional operations that can be used.*

It was also important for the code to be able to distinguish between a valid and invalid equation, so that any system errors or crashes can be prevented. The first step in doing this was to evaluate expressions using eval.

The eval() method does the following:

1. Parse the expression
2. Compile it to bytecode
3. Evaluate it as a Python expression
4. Return the result of the evaluation

Essentially, the eval() function evaluates the "String" like a python expression and returns the result as an integer. If an equation that can be evaluated, a valid equation, is inputted then the answer is calculated and sent back to the client.

If eval cannot evaluate the equation, an error occurs and originally the system would crash. In order to prevent this crash, a try except block was created so that when an invalid equation is inputted, the user is met with a message stating, 'Please enter a valid equation!'.

```
try:
    _vClient.send(bytes(str(eval(m)), "utf-8")) # Evalua
    continue

except:
    _vClient.send(bytes(str(invalidequation), "utf-8"))
    continue
```

*Figure 4: Try and except block.*

The final part was designing the GUI. This was done using Tkinter, an imported module from the standard python library used to create GUIs.

Each of the buttons were coded in a similar way to Figure 5, with each button having its own individual row and column combination.

```
equal = Button(gui, text=' = ', fg='black', bg='red', command=equalpress, height=1, width=7)
equal.grid(row=8, column=3)
```

*Figure 5: Try and except block.*

Figure 5 shows that upon pressing the equal button, the command 'equalpress' is carried out. 'equalpress' is a defined function, which sends the expression to the server, before receiving the answer back and displaying the answer on the GUI. Originally, the calculator was coded in a way such that every value/operation from a button press was sent to the server and 'stored' server side, however this way was extremely inefficient and led to several errors; changing the way the client communicated to the server resulted in less errors, and a much faster calculation time.

The text box on the GUI could then be cleared by pressing the 'Clear' button on the calculator. Figure 6 below shows the code used to achieve this.

```
def clear(): # This fu
    global expression
    expression = ""
    equation.set("")
```

*Figure 6: Code to clear the text box.*

Conclusion:

In order to make the system user friendly, I decided to add some additional features so that the user can understand system processes easier. Once the server receives a client connection, the user is greeted with a welcome message and instructions on how to use the calculator functions. To make user functionality easier, it was coded so that the user could close the system through a button press on the calculator, shown in Figure 7.

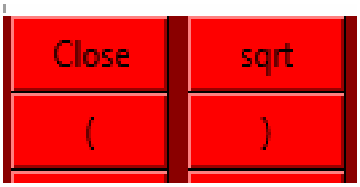


*Figure 7: Pressing Close shuts down the system.*

Additionally, if a valid equation is inputted and an answer is produced, both the answer and the equation are saved to a text file. To allow for the user to understand the text file better, the date and time is saved to the text file everytime the client connects to the server, so that the user can input equations, come back an hour later and carry out more equations whilst differentiating between the two sessions. If an invalid equation/expression is inputted, then an error message is displayed, and the user is prompted to re-enter a new equation.

```
text_file = open("Answers.txt", "a") # o
text_file.write(f'{equation} = {m}\n') #
text_file.close() # close the text file
```

*Figure 8: Saving the answers to a text file.*

Upon testing, we can see that the expression does not clear if an invalid equation is inputted. This was due to personal preference; I preferred to be able to see what was inputted and correct the mistake, however others may prefer the calculator to clear once an invalid equation is inputted, similar to how you can input a new expression without pressing the clear button if a valid equation is inputted. Similarly, the user is required to use brackets when inputting the square root function, as it was easier to visually understand what was occurring when using brackets. If the user wanted to input square root without brackets, the code would be altered to be similar to that of the cube root function.

A recurring issue occurred when the user pressed the 'equals' key without typing anything prior. This would occasionally cause the system to stop working correctly, making it impossible to input anything, forcing you to manually shutdown the system. I am unsure what causes this, but if done again code that recognises when a user has or has not inputted a value will be created to prevent this issue from repeating itself.

From an aesthetics standpoint, Tkinter is not the best GUI framework to; GUIs built with Tkinter look outdated. In a working environment, a more modern sheen may be required, meaning that another framework, such as PySimpleGUI, will be required. However, since the main priority was to design a fully functioning calculator, the appearance was thought of minimally, and Tkinter was chosen as the framework.

Finally, the eval() function, although performing well in this case, is normally "weak" to use. This is because for more complex code, or when coding in a business environment, it is dangerous to use as it is very easy for the user to input malicious code which could then be run, and also makes debugging difficult.

Exercise 2:

Introduction:

This exercise tasked us with building an enhanced ROS version of the calculator coded in exercise 1; creating a ROS package/system with at least 3 nodes. The first node, (#1), was the user input/output interface (text based) that sent commands to the calculation node and log node (#2), and a third node (#3) which provided timestamps of the running time to (#1).

Explanation:

ROS operates differently to the standard ways we have coded before. I decided that the best way to approach this task would be to create a client node and two service nodes. This was achieved by created a .srv file, which allowed me to define the inputs and outputs that would be sent to and from the client (shown in Figure 9). The client node would send information to the service nodes, and the services nodes would compute this information and send the necessary information back to the client before it was outputted and displayed on the GUI. By doing this, node 1 was able to communicate with both node 2 and node 3.

```
string first
float64 tenth
---
string sum
```

*Figure 9: Service file variables.*

Node 3 also needed to advertise a message with the time stamp and running time to node 2, which was achieved using a publisher and subscriber. A Publisher puts the messages of some standard Message Type to a particular Topic. The Subscriber, shown in Figure 10, subscribes to the Topic so that it receives the messages whenever any message is published to the Topic. The publisher was created in Node 3 and the subscriber in Node 2.

```
rospy.init_node('adder') # create a node called
rospy.Subscriber("chatter", String, timer_timer)
rospy.Service('calc', AddTwoInts, callback) # se
```

*Figure 10: Subscriber in Node 2. Carries out a function called timer_timer.*

Each node had their own individual responsibilities, with Node 1 receiving all inputs from the user. Similar to the previous exercise, the user inputted the equation, and the answer was displayed back onto the GUI, with the inputs and any commands entered being logged in a user-friendly manner, making it easy to identify when a certain phrase or equation was inputted.

```
req = AddTwoIntsRequest()
req.first = str(a)
```

*Figure 11: Code showing how the input is sent from Node 1 to Node 2.*

Several commands inputted in node 1, were able to control the functionality of certain aspects in node 2. A command called ‘elements’ was able to count and inform the user about the number of elements within the log file, and ‘history’, shown in Figure 12, was able to display the contents of the log file onto the GUI. The other four useable commands were ‘clear’ which clears the log from node 2, ‘recall’ which recalculates a specific line from the log, ‘time’ which displays the current time to the GUI and ‘close’ which closes the client.

```
if(a == "History" or a == "history"): # if the user inputs h
    checkhistory = open("Answers.txt", "r")
    for x in checkhistory:
        rospy.loginfo(x)
    checkhistory.close()
    text_file = open("AllAnswers.txt", "a")
    text_file.write(f'The command {a} was carried out! \n')
    text_file.close()
    continue
```

*Figure 12: Command to display the History of the log.*

Node 2 handled all calculations and sent the answers back to node 1, whilst notifying node 1 how many operations have been performed. The equation and answer were then logged to a different file from Node 1. Figure 13 displays the incrementor used to count the number of operations. This function was then called in the try-except section of the server, only incrementing when an equation has been solved successfully.

```
def inc():
    global increment
    increment += 1
    return increment
```

*Figure 13: Incrementor.*

Node 3 was used a timekeeper. A start value (defined by the code start = time.time()) was sent from node 1 to node 3 before the system runtime was calculated. This value was then displayed in all three of the nodes. Figure 6 shows the code used to achieve this:

```
end = time.time()
elapsed_time = (end-ans)
timein = timedelta(seconds =elapsed_time)
```

*Figure 14: Code to calculate the System runtime.*

After starting roscore, it was possible to run the 3 nodes individually however to facilitate the process, a launch file was created. Once called, this launches the 3 nodes simultaneously and the inputs/outputs are displayed in a single terminal. It is important to note that the file location changes when using roslaunch, files are read and saved to the '.ros' folder.

Conclusion:

Understanding how the nodes interact with each other was hard to adapt to, however once this understanding was achieved the task became a lot simpler. Communication between nodes was done successfully, and the relevant information was displayed on the GUI or log in a user-friendly manner.

One thing that I initially struggled to get working was recalculating a specific line from the log. The code created was able to read and display the specific line, however it was unable to recalculate the expression because the '=' sign cannot be evaluated using the eval function. Code that removes the server-side information from the line recalled allowed for the line to be correctly recalculated. Additionally, instead of using several if statements to code for the commands, using switch and case would be better, especially if we were going to make this code more complex, as it allows for faster execution compared to if statements.

Exercise 3:

Introduction:

The first task required us to implement two different approaches for robot control using reactive behaviours and sensory-motor control based on the Braitenberg Vehicles approach, and robot control based on the perceptron model. The second task was to implement a 2-layer Multilayer Neural Network or Multiplayer Perceptron (MLP) for the recognition input bits from a 2-bit XOR gate. The MLP was composed of an input, a hidden and an output layer, with this then being implemented in Python without any machine learning library. An MLP for recognition of digits from 0 to 9 and communication via ROS was also implemented.

Explanation:

Activity 1 saw us implement the fear, love, and explorer reactive behaviours in Coppeliasim. We were given code that implements the aggression behaviour (2b), so in order to achieve the fear behaviour (2a), the motors were swapped around so that the sensor-motor connections were no longer crossed. Similar to this, love behaviour (3a) required the same sensor-motor layout as fear behaviour, and explorer behaviour (3b) had the same layout as aggression behaviour. Figure 15 shows how the fear behaviour was achieved within the code:

```
sim.setJointTargetVelocity(motorLeft,vMotors[1])
sim.setJointTargetVelocity(motorRight,vMotors[2])
```

*Figure 15: Code to implement 'fear' behaviour.*

Implementing the fear or explorer code from activity 1 into the wallFollowingWithReactiveBehaviours ttt file allowed the robot to traverse around a maze, speeding up when the sensors detect an object nearby. This is demonstrated in the exercise 3 video linked below.

Activity 3 had us implement a 2-input perceptron model to recognise the input pattern from 2-bits AND, OR and NOR logic gates. In machine learning, the perceptron is an algorithm for supervised learning of binary classifiers. A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class [1]. In order to differentiate between the gates, the user was prompted to input which gate was being used, as shown in figure 16:

```
operator = input('Opertator: ')

X = np.array([[0,0],[0,1],[1,0],[1,1]])

if operator == 'and':
    Y = np.array([0, 0, 0, 1])
elif operator == 'or':
    Y = np.array([0, 1, 1, 1])
elif operator == 'nor':
    Y = np.array([1, 0, 0, 0])
```

*Figure 16: Inputting different gates causes the Y output to differ.*

The weights and bias were then updated using the code below:

```
w[0] = w[0] + delta * lRate * X[j][0]
w[1] = w[1] + delta * lRate * X[j][1]
bias = bias + lRate * delta
print(bias)
```

*Figure 15: Code to implement 'fear' behaviour.*

The code was then ran and designed to stop once the correct output values have been learnt:

```
if global_delta == 0:
    break
```

*Figure 17: The code stops once the outputs have been learnt.*

After several epochs, the code was able to successfully learn and display the correct output for each of the required gates. An example can be seen in Figure 18:

```
epoch  50
nor [ 0 0 ] -> Expected: 1 , Prediction: 1  (w: [-0.31884055] )  (b: [5.02622751] )
nor [ 0 1 ] -> Expected: 0 , Prediction: 0  (w: [-0.31884055] )  (b: [5.02622751] )
nor [ 1 0 ] -> Expected: 0 , Prediction: 0  (w: [-0.31884055] )  (b: [5.02622751] )
nor [ 1 1 ] -> Expected: 0 , Prediction: 0  (w: [-0.31884055] )  (b: [5.02622751] )
```

*Figure 18: After 50 epochs, the correct NOR output is observed.*

In activity 4, the perceptron model was implemented with 3 inputs (R,G,B data), for recognition of red and green colours, allowing the coppeliasim robot to move slowly when the green object is detected and fast when the red object is detected.

```
NO OBJECT DETECTED
NO OBJECT DETECTED
NO OBJECT DETECTED
GREEN BLOCK
GREEN BLOCK
GREEN BLOCK
```

*Figure 19: The display changes and displays the colour when the sensor detects a block.*

To achieve this, the code developed in activity 3 was altered, so that it could recognise green and red colour objects. Coppeliasim and spyder were then run in sequence and, after the learning process, the robot would speed up and slow down after sensing the respective colours. Figure 20 shows how the python code connected to Coppeliasim.

```
clientID=vrep.simxStart('127.0.0.1',19997,True,True,5000,5)
if clientID!=-1:
    print ("Connected to remote API server")
else:
    print("Not connected to remote API server")
    sys.exit("Could not connect")
```

*Figure 20: Connecting the python code to Coppeliasim.*

Activity 1 in part b Implement a 2-layer MLP network to recognise the input bits from an XOR gate. This was achieved by using backpropagation, which is an algorithm for supervised learning of artificial neural networks using gradient descent. Given an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights [2]. Figure 21 shows the results obtained:

```
For input [0, 0] output is 0
For input [0, 1] output is 1
For input [1, 0] output is 1
For input [1, 1] output is 0
```

*Figure 21: XOR outputs.*

Activity 2 was an introduction into activity 3, teaching us how to implement an MLP network using the TensorFlow and Keras Libraries. Activity 3 required us to create an MLP network using the TensorFlow and Keras libraries that can recognise digits from 0 to 9 using the Bath dataset. This firstly required us to path to the correct folder, shown by the code in Figure 22.

```
imageFolderTrainingPath = r'/u/i/an777/Bath_digits_dataset/train'    #
imageFolderTestingPath = r'/u/i/an777/Bath_digits_dataset/validation'
imageTrainingPath = []
imageTestingPath = []
```

*Figure 22: Folder Pathing.*

Once complete, the code was ran to obtain the training and test accuracy:

```
Training accuracy: 94.95
Test accuracy: 95.12
```

*Figure 23: Accuracies obtained after running.*

This code was then implemented into the next activity.

Activity 4 include the require ROS code to publish the predicted digit from your system in order to be received by the calculator (developed with ROS) that was developed in the Coursework Exercise 2 For example, if the user needed to add two numbers A + B, then your calculator should:

1. The first number A will be provided by the user
2. The operator + will be provided by the user
3. The second number B will be recognised and published by your MLP and received by your calculator via ROS.

The results from this activity are discussed in the conclusion.

Conclusion:

This exercise proved to be a lot more difficult than the previous exercises due to my lack of understand of certain aspects. Part 1 was fairly straightforward, and the required tasks were completed correctly however, Part 2 proved to be more of a struggle as the code gradually got more and more complex. Activity 4 was the hardest, and I was unable to correctly implement the code into ROS and connect to my calculator. This was caused by an error that occurred when the publisher attempted to receive the data from the Subscriber. Due to external factors (catching COVID and a power outage), the amount of available time spend on this exercise was greatly reduced; had these issues not occurred, I am positive that the work would have been completed, with activity 4 being implemented correctly.

References:

[1] Perceptron Definition, 10/01/2022, https://en.wikipedia.org/wiki/Perceptron

[2] Backpropagation Definition, 10/01/2022, https://brilliant.org/wiki/backpropagation/

Videos:

Exercise 1: https://www.youtube.com/watch?v=uoxnUZcbW-I&ab_channel=Zechi

Exercise 2: https://www.youtube.com/watch?v=yoTkAdcDs8U&ab_channel=Zechi (Skip to 3:45 to see the calculator process using Roslaunch).

Exercise 3a: https://www.youtube.com/watch?v=Kf4OgQgnpZs&ab_channel=Zechi