Question 2.1: Op Amp Buffer

Part 1:

$$\frac{5 - V_{out}}{5} = \frac{V_{out} - 0}{12}$$
$$60 - 12V_{out} = 5V_{out}$$
$$V_{out} = \frac{60}{17} \approx 3.53V$$

Part 2:

$$\frac{5 - V_{out}}{5} = 0.500 + \frac{V_{out}}{12}$$
$$5 - V_{out} = 2.5 + \frac{5V_{out}}{12}$$
$$\frac{5V_{out}}{12} + V_{out} = 2.5$$
$$\frac{17\,V_{out}}{12} = 2.5$$
$$V_{out} = \frac{2.5 \times 12}{17} \approx 1.76\,V$$

Part 3:

$$V_{out} = A\,(V_{in} - V_{out})$$
$$V_{out} = \frac{AV_{in}}{1 + A}$$

Part 4: In Op-Amps the gain is usually higher. That makes $V_{in} = V_{out} \approx 3.53V$.

The current consumption is given by:

$$I = 500mA + \frac{5 - 3.53}{5}$$
$$I = 0.79\,A$$

Total Power of the battery is given by:

$$P_b = 3500 \times 10^{-3} \times 60 \times 60$$
$$P_b = 12600\,Joules$$

Circuit draws the following amount of power in 1 second.

$$P_c = 5 \times 0.79$$
$$P_c = 3.97\,Joules$$

The total time that the circuit can run on this battery is given by:

$$T = \frac{P_b}{P_c}$$
$$T = \frac{12600}{3.97}$$
$$T = 3173.8\,Seconds$$
$$T = 0.88\,Hours$$

The battery will last for approximately **0.88 Hours**.

Part 5: This circuit is practical, but there can be voltage drops while the battery starts to drain.

Question 2.2 : Assembly Code:

Part 1. Based on Nyquist-Shannon sampling theorem or the signal sampling theorem in digital signal processing, the required sampling frequency to capture the signal frequency of ω should at least be twice the frequency of interest.

$$S = 2 \times \omega$$

This rule helps prevent jitters and aliasing in the signals.
It is given that the sampling frequency is *128 kHz.* Therefore, the maximum frequency that can be captured is:

$$S = 2 \times \omega$$
$$\omega = \frac{S}{2}$$
$$\omega = \frac{128000}{2}$$
$$\omega = 64{,}000$$

Therefore, with the given system, we can capture signals of up to ***64kHz*** without any signal loss.

Part 2.

This can be done by following process:

1. Check if the input value is negative
2. Compare the size of the array with the absolute value and check if it is in the limit
3. If negative wrap index to the other side
4. Finally, reduce the index by 25% of the array size

```
;Storing inputs
mov r0, input 1
mov r1, input 2

; Check if input 1 value is negative
and r0,r0 , #0X8000  ; using 0x8000 because it is 0x1000000000000000

;compare of r0 is equal to 0x8000
cmp r0, 0x8000

; if r2 is negative
bne positive ;branch if not equal
; Take absolute value of input 1 if it is negative
not r0, r0 ; Invert the bits of b0
add r0, r0, #1 ; add 1 to the inverted value and store it in r0

positive:
; Check if absolute value is greater than or equal to array size
cmp r0, r1
bge reduce ; branch if greater or equal
```

```
; If not, wrap index to other side of array
sub r0, r1, r0
b end

reduce:
; Reduce index by 25% of array size
mov r3, r1 ; store r1 in r3
sr r3, r3, #2 ;shift right by two bits
sub r0, r0, r3 ;subtract r3 from r0 and store in r0
end:
```

Part 3:

The above assembly program requires the following number of cycles.

- 2 cycles for moving input 1 and input 2 to registers r0 and r1
- 1 cycle for the *and* instruction to check if *r0* is negative
- 1 cycle for the *cmp* instruction to compare the result of the *and* instruction with 0x8000
- 1 cycle for the *bne* instruction to move to the positive label if *r0* is positive
- 1 cycle for the *not* instruction to take the absolute value of *r0* if it is negative
- 1 cycle for the *add* instruction to add 1 to the absolute value of *r0*
- 1 cycle for the *cmp* instruction to check if the absolute value of *r0* is greater than or equal to *r1*
- 1 cycle for the *bge* instruction to jump to the *reduce* function if the absolute value of *r0* is greater than or equal to *r1*
- 1 cycle for the *sub* instruction to wrap the index to the other side of the array if the absolute value of *r0* is less than *r1*
- 1 cycle for the *mov* instruction to store *r1* in register r3
- 1 cycle for the *sr* instruction to shift the value in register r3 to the right by 2 bits
- 1 clock cycle for the *sub* instruction to reduce the index by 25% of the array size.

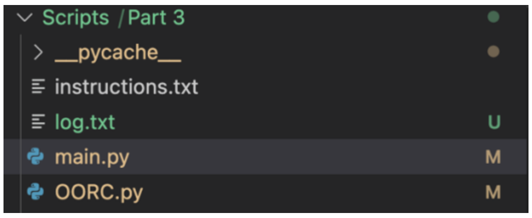
In total, this function takes 12 clock cycles to execute.

a. Total number of available clock cycles at *128kHz is* 128000 per second
b. *12* cycles are required for the assembly function
c. *34* cycles for DAC
d. 48 cycles for ADC
e. 805 cycles for other DSP

Total number of clock cycles is : $805 + 48 + 34 + 12 = 899$
Since 899 is way smaller than the maximum time cycle per second, the assembly program should not cause any problems to the system. However, the maximum frequency of sampled signal could drop by a bit.

<u>Question 3.1: Object Oriented Robot Control</u>

Code for this problem is implemented as two different ***.py*** files. These files are available in the *Part 3* sub folder of *Scripts* folder in the attached zip. This folders tree is as shown below.



```
Scripts / Part 3
  > __pycache__
  instructions.txt
  log.txt          U
  main.py          M
  OORC.py          M
```

1. *instructions.txt* contains all the instructions for the robot
2. *log.txt* contains the log of all robot actions based on the given instructions
3. *main.py* reads the instructions from the *instructions.txt* and performs operations on the robot by using *OORC.py* module and generates the *log.txt* file
4. *OORC.py* contains *Robot* module that *drives or stops* the robot and sets all the required features of the robot

Run this script by using following command:

python3 main.py

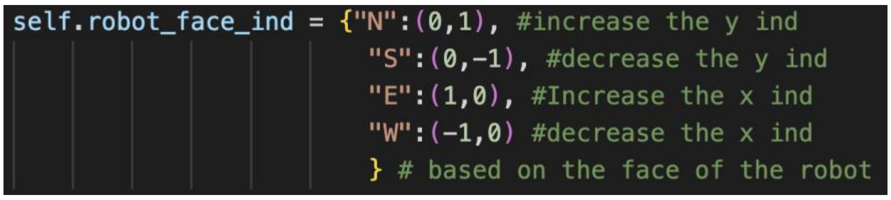
In this script, the robot starts at (0,0) facing North . It is assumed that the north direction is same as positive y direction. If a robot is moving in the north direction, it makes sense to only increment the y index of the robot position. For example, if the robot took one step in the north direction, then the robot position changes from (0,0) to (0,1). If the robot is at (1,2) and took one step in the north direction, then it's new position will be (1,3).

Similarly, if the robot is moving in the south direction, its y index will be reduced by one. For example, if the robot moved by one step in south direction from (3,2), then its new position will be (3,1).

If the robot is moving in the east direction, which is nothing but positive x axis, then its x index will be increased by 1. For example, is the robot moved one step in east from (0,0), its new position will be (1,1).

Someway, is the robot is moving in west direction , its x index is decreased by one. For example, if the robot is moving in west direction by a step from (2,3), its new position is (1,3)

This procedure can be generalized by the following dictionary.



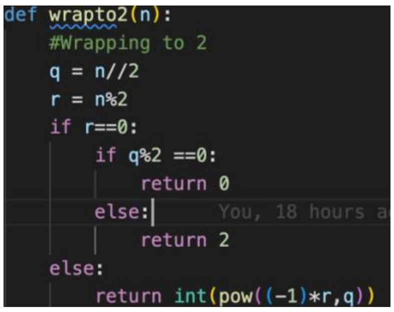
```
self.robot_face_ind = {"N":(0,1), #increase the y ind
                        "S":(0,-1), #decrease the y ind
                        "E":(1,0), #Increase the x ind
                        "W":(-1,0) #decrease the x ind
                        } # based on the face of the robot
```

In the positive rotating order, the directions can be initialized as "NWSE"

Whenever there is a rotation of n steps, this string can be rotated, and the first index of this string represents the robot facing direction.

For example, if the robot is facing “N” and turned by one step in the counterclockwise direction, the string changes to “WSEN”, and the robot’s facing direction will be “W”. Same way, if the robot is facing “E”, and turned by 1step in the clockwise direction , the its new facing direction is will be “S” , because the string changes from “ENWS” to “SENW” .

And the turning steps are wrapped by 2. When the rotation is given as 3 steps, it is same as turning 1 step in the clockwise direction. Similarly, if the rotation is given as 5 steps, it is same as turning 1 step in the counterclockwise direction. Implementation of this is given by the following function.


```
def wrapto2(n):
    #Wrapping to 2
    q = n//2
    r = n%2
    if r==0:
        if q%2 ==0:
            return 0
        else:        You, 18 hours a
            return 2
    else:
        return int(pow((-1)*r,q))
```
For driving the robot, the incrementing factor for indices is taken based on the robot’s facing direction. And at each iteration, the possibility of moving to new goal is checked. If the goal is not reachable, then the robot is turned by 1 step in the counterclockwise direction.

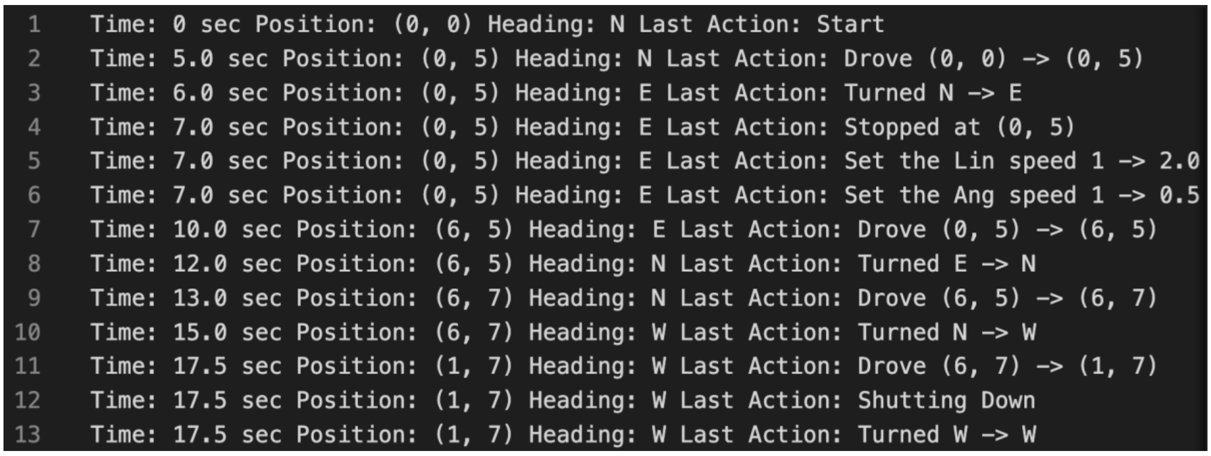
Time for each action is computed based on the linear speed for driving, and angular speed for turning. This time is appended at each step.

For the given following example instructions:

```
Sample instructions.txt:


D5
T-1
UL10.4
S
SS2.0
ST0.5
D6
T1
D2
UL0.9
D5
ES
T4
```

The following is the log data:



```
Time: 0 sec Position: (0, 0) Heading: N Last Action: Start
Time: 5.0 sec Position: (0, 5) Heading: N Last Action: Drove (0, 0) -> (0, 5)
Time: 6.0 sec Position: (0, 5) Heading: E Last Action: Turned N -> E
Time: 7.0 sec Position: (0, 5) Heading: E Last Action: Stopped at (0, 5)
Time: 7.0 sec Position: (0, 5) Heading: E Last Action: Set the Lin speed 1 -> 2.0
Time: 7.0 sec Position: (0, 5) Heading: E Last Action: Set the Ang speed 1 -> 0.5
Time: 10.0 sec Position: (6, 5) Heading: E Last Action: Drove (0, 5) -> (6, 5)
Time: 12.0 sec Position: (6, 5) Heading: N Last Action: Turned E -> N
Time: 13.0 sec Position: (6, 7) Heading: N Last Action: Drove (6, 5) -> (6, 7)
Time: 15.0 sec Position: (6, 7) Heading: W Last Action: Turned N -> W
Time: 17.5 sec Position: (1, 7) Heading: W Last Action: Drove (6, 7) -> (1, 7)
Time: 17.5 sec Position: (1, 7) Heading: W Last Action: Shutting Down
Time: 17.5 sec Position: (1, 7) Heading: W Last Action: Turned W -> W
```

Question 1.1 : Motor Specs and Transmission Design

Part 1. **Speed vs Torque**

Relation between speed and torque is inversely proportional. At torque equals to zero, the rpm is called No-load rpm , and it is given as 4610 RPM. And at zero rpm , the torque is called stall torque , and it is given as 2.05 ft-lb .

The torque-speed relation is nothing but a straight line between $(0, \omega_0)$, and $(\tau_s, 0)$

And equation of this straight line is given by :

$$\omega = -\frac{\omega_0}{\tau_s}\tau + \omega_0$$

Current vs Torque

Relation between Torque and current is a straight-line between $(0,0)$ and (τ_s, I_s)

$$I = \frac{I_s}{\tau_s}\tau$$

Power vs Torque

$$P_M = \tau\omega$$

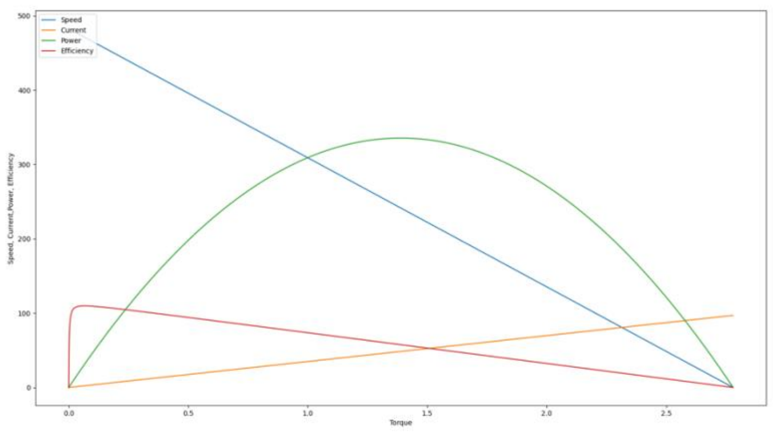
Here P_M, is mechanical power output. Electrical power input can be computer as $P_E = VI$.

Efficiency:

$$E = \frac{P_M}{(P_E + I_f^2 R_a)} \times 100$$

Here $I_f^2 R_a$ Represents the losses

Plots:



Script for this problem can be found in Part 1 folder

Part 2. The maximum torque required for the application is 22.3 ft-lbs. But the DC motor can only generate a max torque of 2.05 ft-lb.

To solve this problem, we can design a drive train with a gear ratio of $\frac{22.3}{2.05} = 10.87 \sim 11$

Part 3. Back emf of the dc motor is computed by :

$$\mathrm{E_b} = V_t - I_a R_a$$

Terminal voltage V_t is given as 12V. The current at maximum efficiency is called free current. And it is $2.4\ A$

$$E_b = 12 - 2.4 * 0.12$$
$$E_b = 11.712$$

Part 4: