Team 2

MEGA PROJECT REPORT

Names:

- Youssef Okab
- Heidi Gad
- Youssef Fayed

<u>Overall Index:</u>

1. Electrical System:

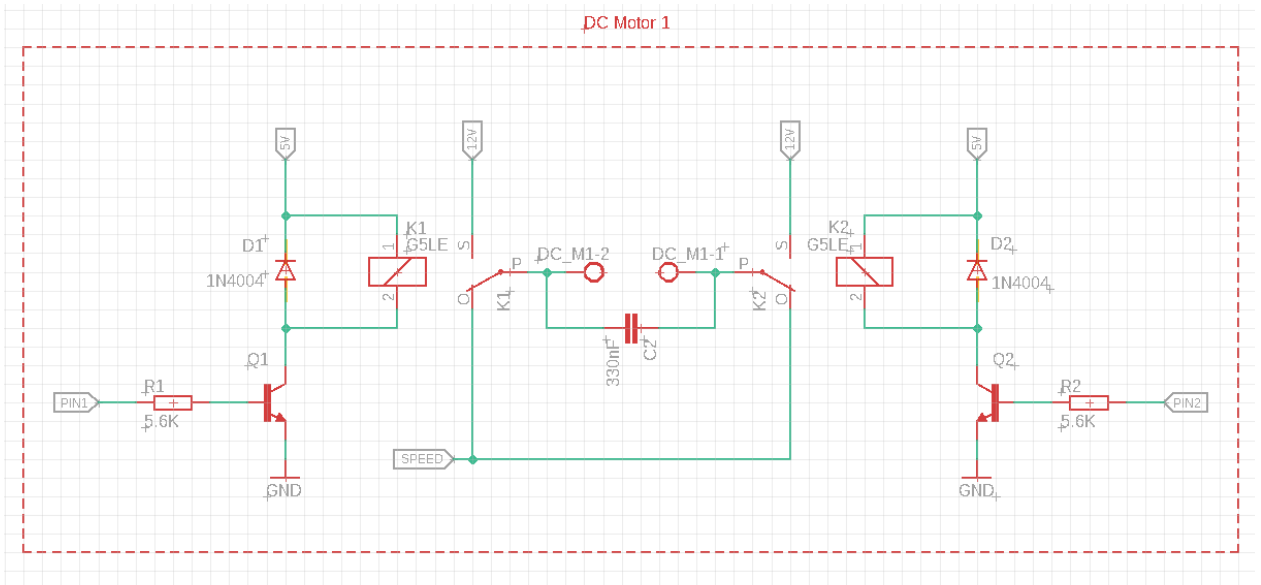
PCB Schematic

We can divide our PCB into 3 categories for simplicity as follows:

1. Motors control
2. Voltage regulation
3. Sensors

Motors Control

- DC Motors



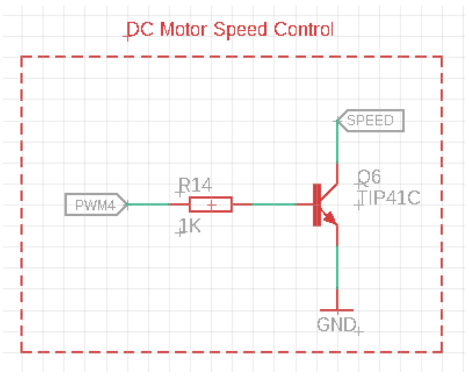
It is required to use 2 relays for each motor in order to be able to move the motor in 2 directions. Relays are divided into 2 parts internally:

1. Coil: it is controlled through a transistor working as switch that takes its digital signal from Arduino. The transistor should have a base connected resistor to control the current flow from collector to emitter, its maximum value is calculated as follows

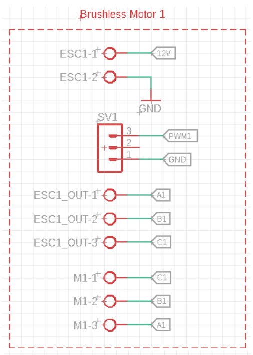
   (Input Voltage- V_{ce}) / (collector current/DC current gain)

   As the coil draws 70mA from 5v supply, 2N2222 transistor is chosen as it holds up to 80 mA, 5.6KΩ resistor is connected to its base, also a fly back diode is connected backwards across the load to suppress the voltage spikes (back EMF) generated when turning devices off.

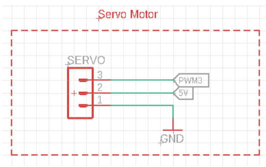
2. Switch: Poles of relays are the static parts; it should be connected to the motor leads, the normally open terminal connected to 12V to power up the motors only when the coil is energized, the normally closed terminal are connected to the collector of a fast flickering transistor to control the speed of the motor. The stream of on/off voltage happening through a PWM signal with Tip41c transistor, that is chosen as it holds up to 6A, is perceived by the motor as an intermediate voltage. A 1K base resistor is connected as declared before and the emitter is grounded. Finally, a 330nF ceramic capacitor is connected at the output to smooth the voltage.



- Brushless thrusters
  ESCs are used to control it, thus 2 input terminal block is needed to deliver 12V to the ESC, two of a 3 input terminal blocks are used to connect ESC to thruster terminals, and pin header is used to connect PWM from Arduino to ESC.

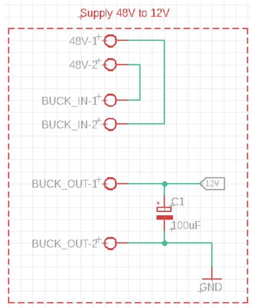


- Servo motor
  Pin header is used to deliver 5V and connect PWM from Arduino to motor.

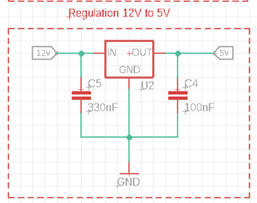


Voltage regulation

- Buck Converter is used to step down from 48V to 12V in order to supply the motors and Arduino. Three of 2 input terminal blocks are used. The first is the input of 48V supply, the second is the input of the buck converter which is connected with the 48v, the third is the output 12V from the buck converter. 100uF decoupling capacitor is used to filter the output noises due to buck converter.

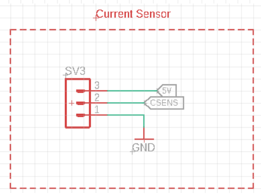


- Voltage regulator 7805 is used to step down from 12V to 5V in order to supply other components. 330nF capacitor filters the input, and 100nF capacitor filters the output.



Sensors

- Current Sensor
  In order to measure current, 12V buck converter output should be connected to terminal block on the current sensor and the other wire connected back to the PCB.
  Pin header is used to connect analog signal between ACS712 and Arduino, power sensor with 5v and last pin is grounded

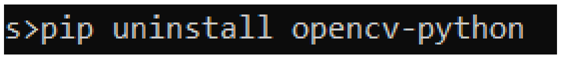


  lesa el equation
- Voltage Sensor

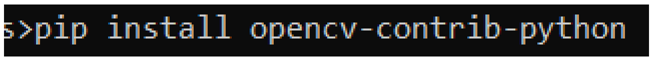
- Leakage Sensor

2. Software Tasks Algorithms:
    a. Stitching.

First of all for this algorithm to perform perfectly, you should have the contrib version of opencv 4.5.3 as we are using the “cv.xfeatures2d.SIFT_create()” function which is considered a non free algorithm and it doesn’t exist in normal version of python opencv 4.5.3. So to set it up, we get the CMD started and do as follows.



```
s>pip uninstall opencv-python
```



```
s>pip install opencv-contrib-python
```

Done? Great. Now to the next step which is specifying the exact code files for the algorithm:

stitch4.py : Main code that is used in the GUI.

Stitch3.py : Starting code to make sure the algorithm works (found inside “”).

Now, let’s talk about the algorithm itself. We are supposed to take an input of randomly arranged horizontally 5 images which needs stitching and output the main wide image.

- We start by placing the resized 5 images (for further use) in an array of images.
- We run the method join.
- Join:
    - which loops over the array of images, compares the first image array with the rest and checks if 2 images can be stitched together by doMatch.
    - If they don’t match, we continue til we reach a match but what if we reached the end of our array with no match? We interchange the first and last images in the array but that means one of two conditions:
        1. “Left out” condition (will be discussed later).
        2. An error image that has nothing to do with the others is added.
    - So, what happens afterwards is that the loop iterates exactly five times and that should be enough to have the final image in the array whether it is the only image there or it has some “left out” or “error” images aside.

- doMatch:
    - Boolean function that takes an input of 2 images and returns whether match is made or not by returning checkMatch, remove images from array if there is match, append the output (so that size decreases by one in each iteration) and break out of the join loop (and it does that whether they are in right order from left-to-right or not).
- checkMatch:
    - runs stitch and returns whether 2 images are stitchable.

- Stitch:
    - Takes as input 2 images and runs detectAndDescribe on both then runs matchKeypoints, returns None if matchKeypoints returns None (which lets checkMatch returns False to detect that images does not match), then we run myWrapPerspective. If we went this far in the method now we have the resulted merge of 2 images. Then, we run drawMatches which has 2 missions (postponed). So now we've got 3 conditions:
        1. The images match but are in wrong order or they aren't supposed to match but they did due to some noise. In this condition we call doMatch again but reverse indices to make the correct stitching after recuring once. We stop recuring after this one time recursion and return None as we now made sure we are in the "noise" condition. So, we return None.
        2. The images match but they are in the "left out" condition. This takes place when 3 consecutive images have very high matches that the first and the third one can output a 100% correct output without the need to link with the second one. So this means that if back in the join loop, if the second image faced the output one, they will match highly but still we cannot link the second image to them. So, in this case, we return None
        3. If we passed the last 2 conditions this means that our output image is 100% correct. So we return it at last.
- detectAndDescribe:
    - We use the SIFT algorithm mentioned before to get the features of both images (that is why we run it twice in stitch) and descriptors. The descriptors indicate the identity of feature such that if 2 features are alike they should ideally have the same descriptor and it can be called "key". So, now we return the keys and the features as well.
- matchKeypoints:

    - This function takes the keys and the features and outputs the matches between them. Matching features together is actually a fairly straightforward process. We simply loop over the descriptors from both images, compute the distances, and find the smallest distance for each pair of descriptors. Since this is a very common practice in computer vision, OpenCV has a built-in function called cv2.DescriptorMatcher_create that constructs the feature matcher for us. The "BruteForce" value indicates that we are going to *exhaustively* compute the Euclidean distance between *all feature vectors* from both images and find the pairs of descriptors that have the smallest distance.
    - We get the top 2 matches for every set of matches so that we can apply the David Lowe Ratio Test which helps us in filtering matches and just keeps the most accurate matches. All it does is ensure that the distance between descriptors of matched features is within a certain ratio (0.75 in our case).
    - Now we check the number of matches found and return None if they are less than 10 as in our case it is logical to assume that we will find much more than 10 matches if the 2 images actually do match.
    - If we reached that far it means that images match. Then, we construct a set of points that indicate where our matches are in both images.
    - We pass these sets of points to the built-in function `findHomography` by reprojection of 4 and we choose the RASNAC algorithm which chooses the top 4 matches in each set of points and creates a homography transformation 3x3 matrix which transforms one image to fit the other. The funny thing is we don't need the homography matrix in our algorithm. So why if this function lying there?! The thing is the function doesn't just return the4 homography matrix. It returns an array that detects the perfect matches in the corresponding points to eachothers and places one in their corresponding index in the array otherwise zero. This array is called status. This will help us later in debugging, in myWrapPrespective.
- Now, we are back to stitch from matchKeyPoints and now we are running drawMatches.
- drawMatches:
    - Takes both images, both keys, match and status. It has 2 missions as mentioned before.
        1. Output both images drawing a green line between each 2 matches to visualize exactly what is happening. We used this a lot while debugging but surely we have no need for it in the actual task
        2. It gives 3 VIP outputs which we will be used when running myWrapPrespective later on as well as "left out" condition detection:
            - The leftmost match in left picture (bpl)
            - The leftmost match in the right picture (bpl2)
            - The rightmost match in the left picture (bpr2)

- Now, we are back to stitch from drawMatches and now we are running myWrapPerspective.
- myWrapPerspective:
    - It takes the 2 images as input and outputs the result of stitch, How? As follows:
        1. We make an image having height of one of them and width of first image from its leftmost pixel til bpl plus width of the second image from bpl2 to its rightmost pixel and fill it exactly as mentioned to get a perfect stitch. In this step, we assumed that all images have same height and it only needs width adjustment (which was shown in the two examples given).
- Now we are back from myWrapPerspective to stitch to checkMatch to doMatch to join to our main code.
- The next step now that we have the result in our array is to filter the array and get it out and we can easily do that as our resulted image has the maximum width in the images left in the array. So we run maxWidth which is our final used method.
- maxWidth:
    - outputs image of maximum width (Address: SOD Outputs\Output.jpg).

<u>Assumptions Made:</u>

1. images that belong together will definitely have more than 10 matches using our algorithms.
    - Even if they did the output will be neglected if its width is only 150 pixels more than image with maximum width.
2. Images have same height and just needs width adjustment.
3. Images entered should be 5. They don’t need to match though. It will match what can be matched.

b. Object Detection:

Code File:

objDet2.py

- Our input is the stitching output.
- We blur the image to get rid of noise.
- We then convert the image colour to hsv so that we can detect the objects by colours. We try to get each desired colour alone as follows:
    - Blue: for Sea Star.
    - Yellow: for coral fragment
    - Pink: for coral colony
    - Black: for coral colony
    - White: for sponge and coral colony
    - Green: for sponge
- We get a binary image of these colours using the mentioned method in the code getMask and using the corresponding ranges for each colour.
- getMask:
    - The method takes 6 integers representing the upper and lower RGB ranges.
    - Outputs a binary image where the desired colour parts are in white and the rest of the image is black.

- Then we make the following additions to get the 4 images of the components that we want so we can forward to the next step.
    - Colony = pink + white + black
    - Sponge = white plus green
- Next step is that each of the 4 images needs dilation except for the coral fragment which is actually a square. So, it is the easiest object to be detected.
- Post dilation, we then pass each of the 4 images to the method comp:
- Comp:
    - Takes 5 parameters: binary image, lower are threshold, upper area threshold, colour of detection, name of object.
    - We use the connectedComponentsWithStats method to detect the object we want in each image as we loop over the components it produces and when the loop gets an object within the area threshold and append it to the skip array. Why so? What does the skip array do?
    - If the area of label is within the area range, this means we found our detected object. But in some cases our detected object has some missing parts right beside it but at the same time not considered a part of the component. So, we will just be detecting part of the object. So, we do the following trick:
        - Now that we found our object, we loop over the components again to search for any component of area less than our object and at the same time within 30 pixels from it or less. Once found, we add it to our component.
    - We then place the coloured frame around its dimensions and write the boxed text right above it.
    - We do this change in the variable of the image itself so that it will be updated frequently and in the end we get our output (SOD Outputs/Final Output).
    - Skip array: sometimes the object breaks into two for some reason and each of the 2 objects is within area. So once we get an object and it finds its mate and links with it, we add the index of its mate to skip array to prevent double detection.

Assumptions:

1. Each object will always have the same colour in all test cases.
2. Each object will never get out of the area ranges in code
3. No object will be linked with another object of same colour (whether they are 2 object or an object and a frame).
4. No displacement between 2 parts of the same object will be more than 30 pixels.

3. <u>GUI Features:</u>

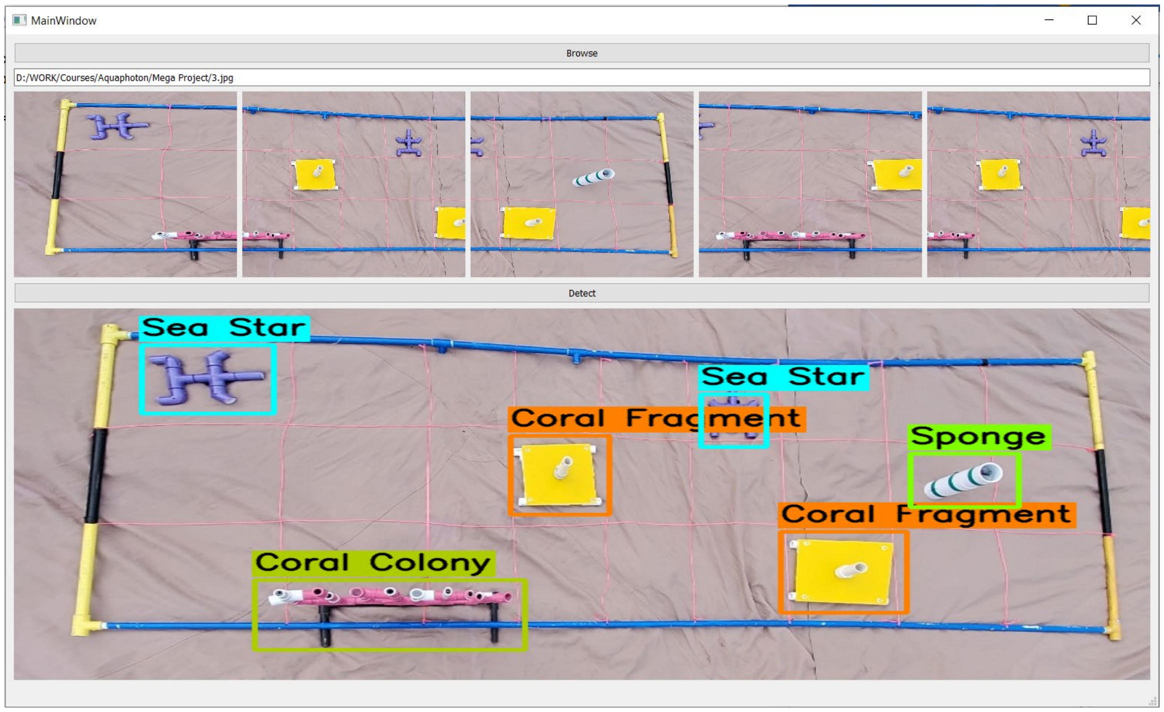
   <u>SubWindows:</u>

   a. Stitching and object detection option:

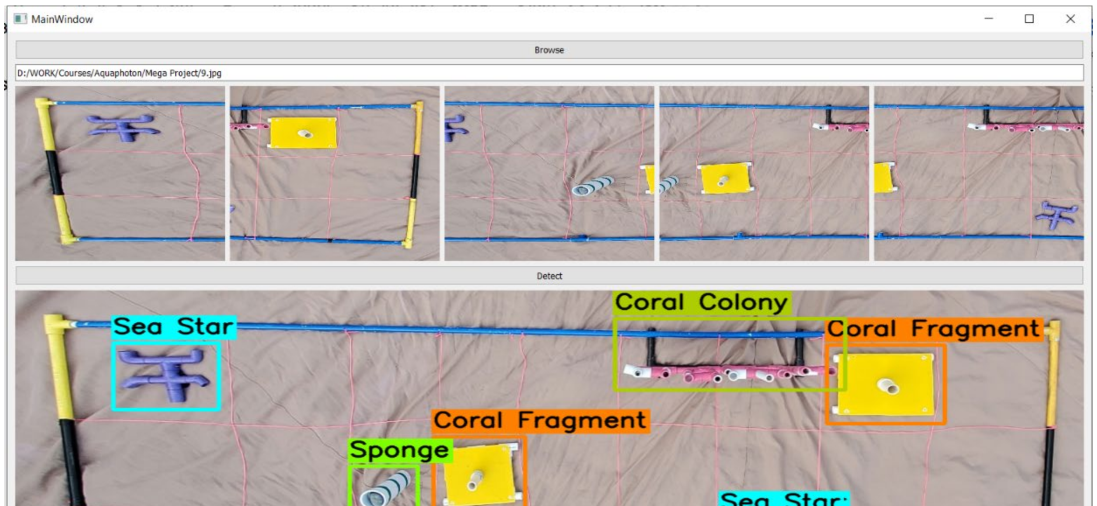
Code file:

SODGUIbd.py

- We placed 5 labels in which we browse to get the desired 5 images and the 5 images are shown. Once the 5th image is in there, the detect button in enabled. Once pressed on it. The final output of stitching and object detection appears.



D:/WORK/Courses/Aquaphoton/Mega Project/3.jpg



D:/WORK/Courses/Aquaphoton/Mega Project/9.jpg

Assumptions:

1. The user will only need to access the stitching and object detection option once per program run as labels don't get overwrited.

PS:

For some odd reason that we haven't quite figured out yet, this sub window in the main program doesn't output the image we want. It does save it in "SOD Outputs/Final Output.jpg"

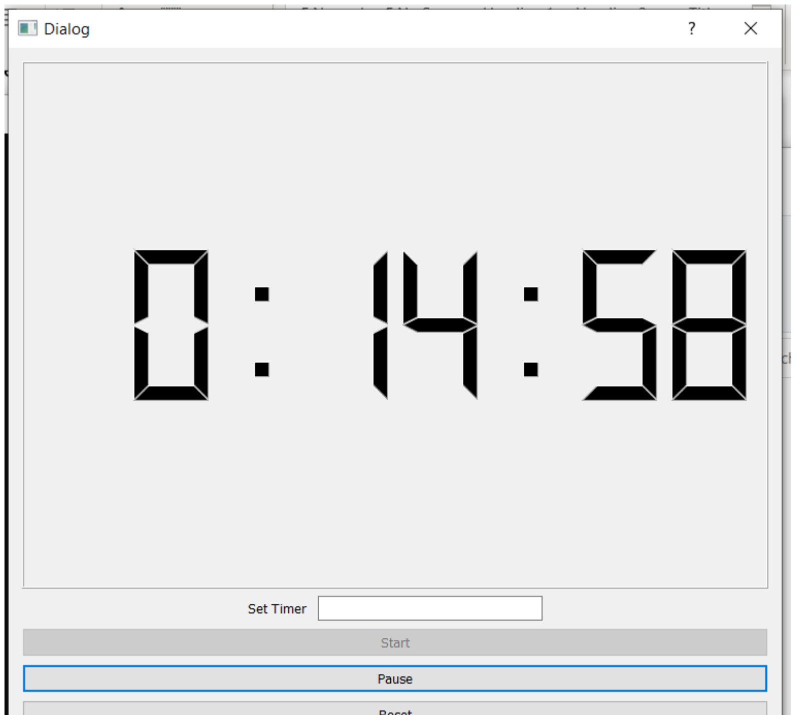
SODGUIbd.py

and if we run the code the GUI of the sub window works perfectly. So, the only explanation is that this algorithm hates to be dealt with as a sub window as it is meant to be the main window 😊.

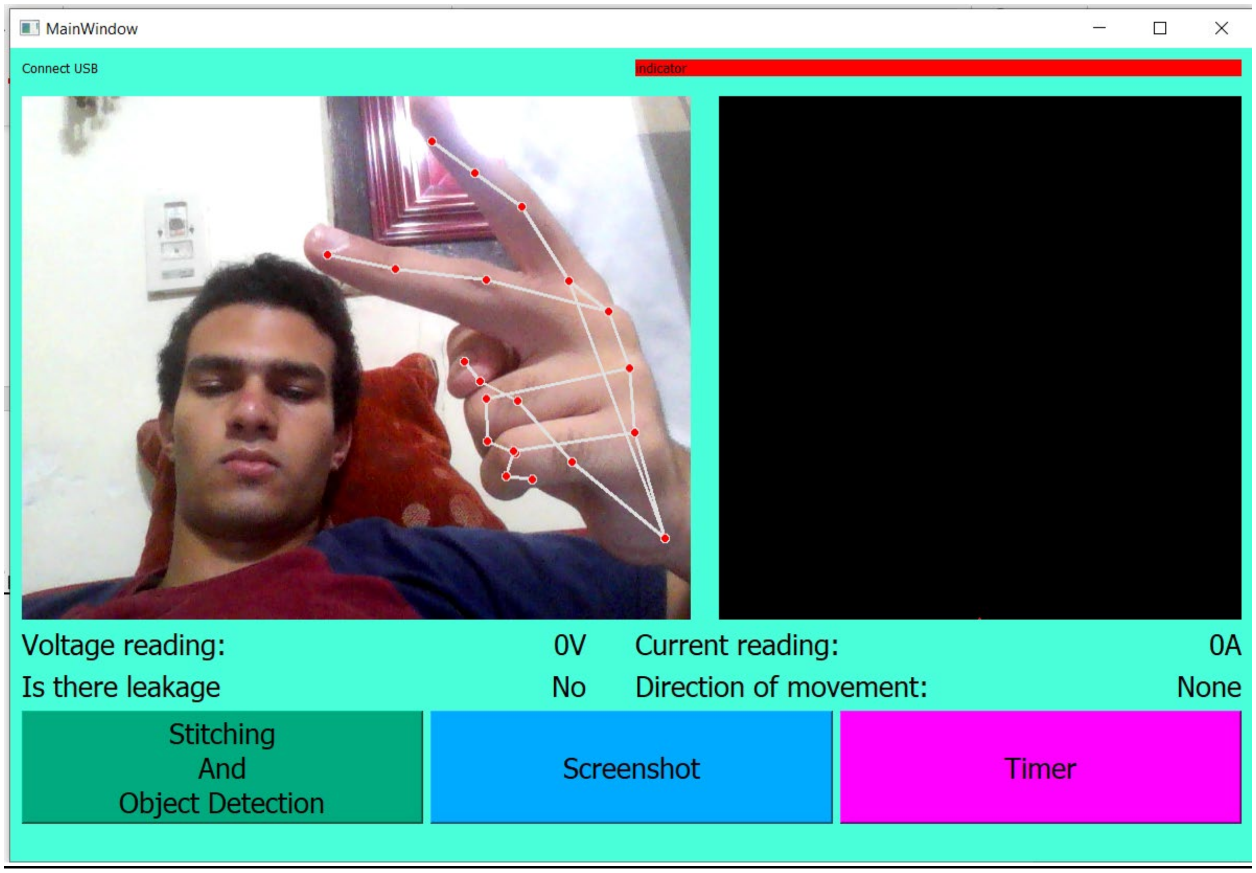
Timer:

call_timer.py

- We designed a simple timer that takes the time in seconds as input in the text box. Then, press start to start timer. Reset button resets timer to 0:00:00, pause simply stops timer.



Main Window:



Voltage reading: 0V
Current reading: 0A
Is there leakage No
Direction of movement: None

- As shown in figure, we've got the live feed front camera:
    - It represents the hand detection task which means it outputs the number of fingers that are raised.
    - We decided to make it a part of the main program which is kind of cool that our program directly opens a front camera and you can take a screenshot anytime you like while the window is running. However the drawback to that is that it made the mapping task relatively slow ☹
- We've got the screenshot button as we mentioned before. Once its pressed a pop up window appears havin the screenshot and it is saved in the folder "screenshots".

- We’ve got also the mapping which is kind of weird because it is part of the main window but for some reason it wont work unless the pop up of the map we’ve made is on the top of the window stack. Then what will happen in the pop up window will be reflected in the map in the main window and it doesn’t work anyother way. So the best way to do this is to have the pop up be synced exactly on the map in the main window so that it looks like it the image in the label is changing. Also, the label next to the direction of movement text alternates between (up, down, left, right) according to the keyboard key we’re pressing on). Goes back to None if no key is pressed.
- We’ve got buttons for Timer and Stitching and Object Detection.
- The label “Connect USB” which has red colour beside it changed to “USB CONNECTED!” with green colour.
- We also have labels to register current, voltage and leakage reading which takes place when USB is connected.