# [Fall 2023] ROB-GY 6203 Robot Perception Homework 1

Panayiotis Christou

Submission Deadline (No late submission): NYC Time 11:00 AM, October 04, 2023
Submission URL (must use your NYU account): https://forms.gle/TNrJy7r3smx2t1ed8

1. Please submit the **.pdf** generated by this LaTex file. This .pdf file will be the main document for us to grade your homework. If you wrote any code, please zip all the **code** together and **submit a single .zip file**. Name the code scripts clearly or/and make explicit reference in your written answers. Do NOT submit very large data files along with your code!

2. You don't have to use AprilTag for this homework. You can use OpenCV's Aruco tag if you are more familiar with them.

3. You don't have to physically print out a tag. Put them on some screen like your phone or iPad would work most of the time. Make sure the background of the tag is white. In my experience a tag on a black background is harder to detect.

4. Please typeset your report in LaTex/Overleaf. Learn how to use LaTex/Overleaf before HW deadline, it is easy because we have created this template for you! **Do NOT submit a hand-written report!** If you do, it will be rejected from grading.

5. Do not forget to update the variables "yourName" and "yourNetID".

## Contents

# Task 1 Sherlock's Message (2pt)

Detective Sherlock left a message for his assistant Dr. Watson while tracking his arch-enemy Professor Moriarty. Could you help Dr. Watson decode this message? The original image itself can be found in the data folder of the overleaf project (https://www.overleaf.com/read/vqxqpvbftyjf), named `for_watson.png`

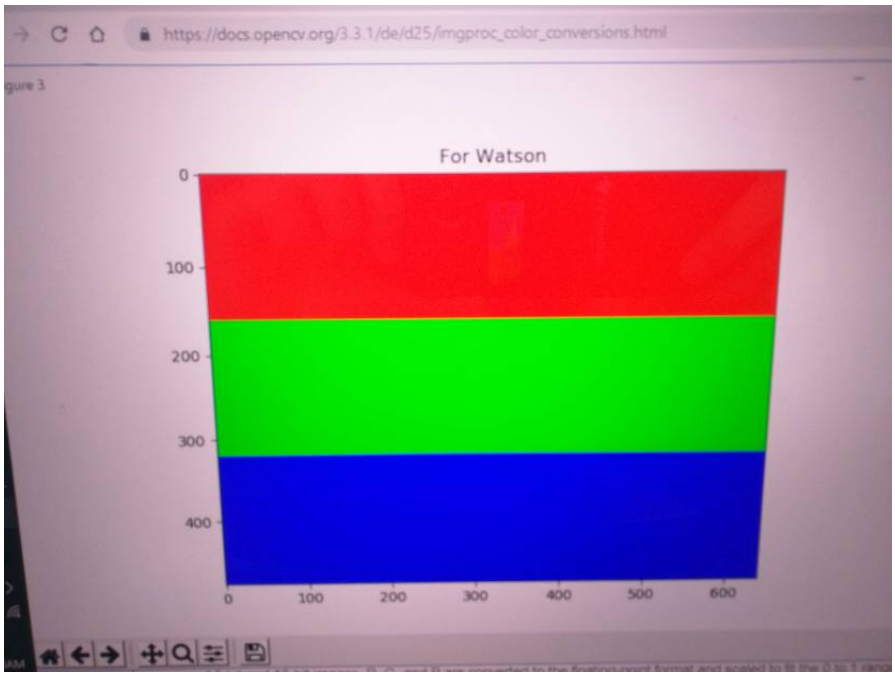



Figure 1: The Secret Message Left by Detective Sherlock

## a) (1pt)

Please submit the image(s) after decoding. The image(s) should have the secret message on it(them). Screenshots or images saved by OpenCV is fine.

**Answers:**

You can use this code snippet to include a picture



Name: Panayiotis Christou, netID: pc2442

## b) (1pt)

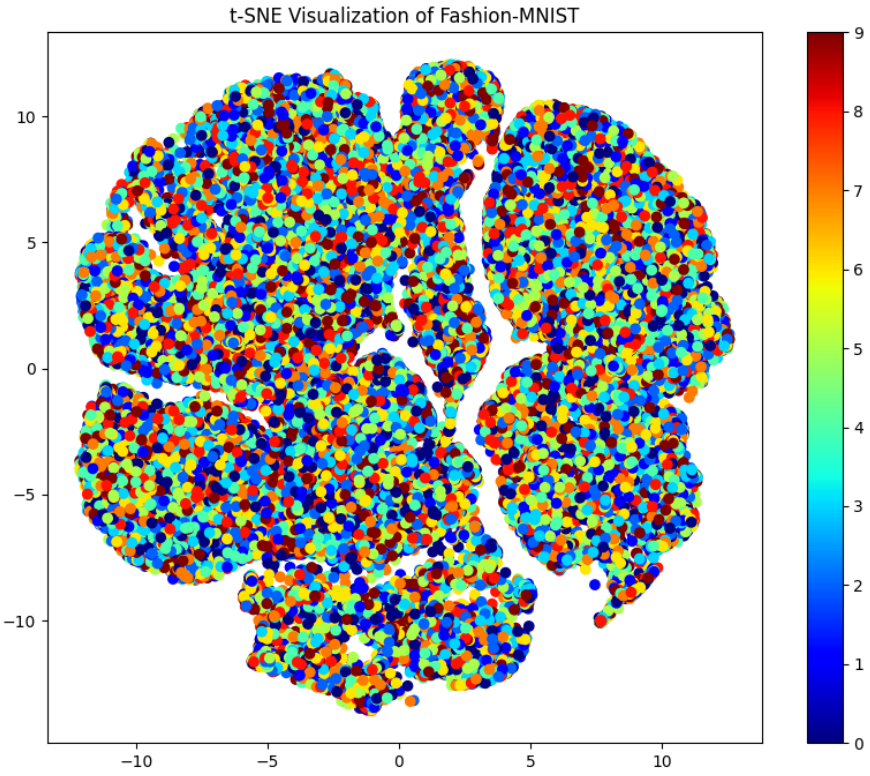
Please describe what you did with the image with words, and tell us where to find the code you wrote for this question.

**Answers:**
First I read the image in using cv2.imread and using the parameter cv2.IMREAD COLOR so that I can read it in with colour. Then I used the cv2.threshold command to threshold the original image in a binary format using the cv2.THRESH BINARY parameter and with the threshold value being 0 and the maximum value of a pixel being 255. For more information you can see the function documentation on https://www.scaler.com/topics/cv2-threshold/. This showed the message which can be seen in the previous image. You can find the github for the code here: https://github.com/trollzero/ROB-GY-6203-Robot-Perception-F23-NYU/tree/main/Homework in homework 1 and in the HW1Q1 notebook (for some reason the direct to link to the notebook is breaking the compilation). I included explanations for each part of the code in the notebook as well.

# Task 2. Low Dimensional Projection (5pt)

Given the Fasion-MNIST dataset, **train** an unsupervised learning neural network that gives you a lower-dimensional representation of the images, after which you could easily use tSNE from `Scikit-Learn` to bring the dimension down to 2. **Visualize** the results of all 10000 images in one single visualization.

**Answers:**
For this question I used pytorch to train a Variational Auto Encoder and then used the t-SNE to reduce the dimension to 2 so that I could visualize all 10000 images in one single visualization.
The visualization I got is the following:




For detailed explanation of each step of the code and the code itself can be found in https://github.com/trollzero/ROB-GY-6203-Robot-Perception-F23-NYU/tree/main/Homework in homework 1 and in the HW1Q2 notebook.
In essence, the dataset is downloaded and transformed into a tensor for further processing. A dataloader is defined with a batch size of 64 and shuffling set to true. Inside the autoencoder we have the encoder and the decoded. In the encoder we first pass the images in a convolutional 2D layer with a stride of 2, kernel of 3 and padding of 1 to reduce the image from 28x28 to 14x14 but also capture enough detail and not lose any of the information, then use a ReLU function and then through an additional convolutional 2D layer to reduce it even further from 14x14 to 7x7, then another ReLU function and then flatten it so that it can be passed to the decoder.
In the decoder we unflatten the image back to the 7x7 shape and then we pass it through a transposed convolutional 2D layer with the same parameters as before and with ouput padding equal to 1 so that we increase the dimension back to 14x14, then through a ReLU function and then another transposed convolutional 2D layer to increase the shape of the image back to the original 28x28 and then pass it through a sigmoid layer for classification.
The network used MSE loss and the adam optimizer for 10 epochs and it finished with a loss of 0.0005253226412070502.
For the t-SNE visualization we needed the lower representation encodings which we found by looping through the images and feeding them to the autoencoder but only getting the encodings and not using the decodings. The data was passed to the TSNE function with number of components = 2 for easy 2D visualization and relatively faster inference and perplexity = 30 which is the default since the recommended is between 5 and 50 and for 300 iterations. You can see the visualization above. For further explanation you can go to the notebook on the link above as explained.

# Task 3 Camera Calibration (3pt)

Use the pyAprilTag package provided in the class, or other free packages (e.g., OpenCV's camera calibration toolkit) that you may be aware of, to **calibrate** your camera and provide the full K matrix, with the top two distortion parameters k1 and k2.

**Answers:**
For this question I used OpenCVs camera calibration toolkit using a calibration board in this Medium Post https://aliyasineser.medium.com/opencv-camera-calibration-e9a48bdd1844.
For detailed explanation of each step of the code and the code itself can be found in https://github.com/trollzero/ROB-GY-6203-Robot-Perception-F23-NYU/tree/main/Homework in homework 1 and in the HW1Q3-4 CalibrationBoard notebook.
The full K matrix and the top two distortion parameters k1 and k2 can be seen in the figure below taken as a screenshot from the output of the notebook for a clean representation.

```
Camera Matrix (K):
[[2.09862213e+03 0.00000000e+00 1.15512024e+03]
 [0.00000000e+00 2.10149813e+03 7.81509956e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Parameters (k1, k2):
0.024360057477338787 0.6431001054247808
```

In essence, I took 12 pictures holding the calibration board pattern as shown above. The pictures are not in the github repo because I don't want to have my face uploaded but I can provide the google drive link if you need to recreate the results.
The pictures were appended into a list. and then the pattern of the checkerboard was defined. The number of rows was 6 and the number of columns was 9.
The object points and image points lists were initialized to keep the object points (3D) and the image points (2D). Then an array of (rows*cols,3) was initiated to zero to store the 3D points of the pattern used for calibration. Then using the X and Y coordinates a grid is created as 2 2D arrays of n rows x 2 cols to keep the coordinates of the corners of the checkerboard pattern. For more information see the code explanation in the notebbok.
We then go through all the images 1 by 1 and use the cv2.findChessboardCorners function to calculate the corners if ret returns true and append them into the lists initialized above.
We then use the object points and image points found from the images in the cv2.calibrateCamera function to find the rotation vectors, translation vectors, the distortion and the K matrix and then use the distrortion to find k1 and k2.

# Task 4 Tag-based Augmented Reality (5pt)

Use the pyAprilTag package to detect an AprilTag in an image (or use OpenCV for an Aruco Tag), for which you should take a photo of a tag. Use the K matrix you obtained above, to draw a 3D cube of the same size of the tag on the image, as if this virtual cube really is on top of the tag. **Document** the methods you use, and **show** your AR results from at least two different perspectives.

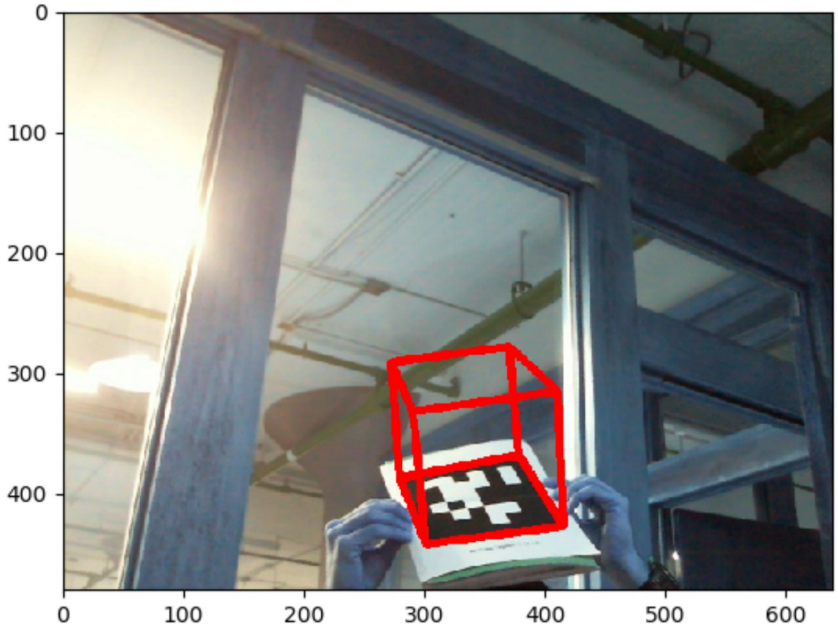



Figure 2: Caption

**Tips**: There are many ways to do this, but you may find OpenCV's `projectPoints`, `drawContours`, `addWeighted` and `line` methods useful. You don't have to use all these methods.

**Answers:**
For this question I used OpenCVs Aruco Tag and took multiple photos with it to test the function. The Aruco Marker I used is shown below:

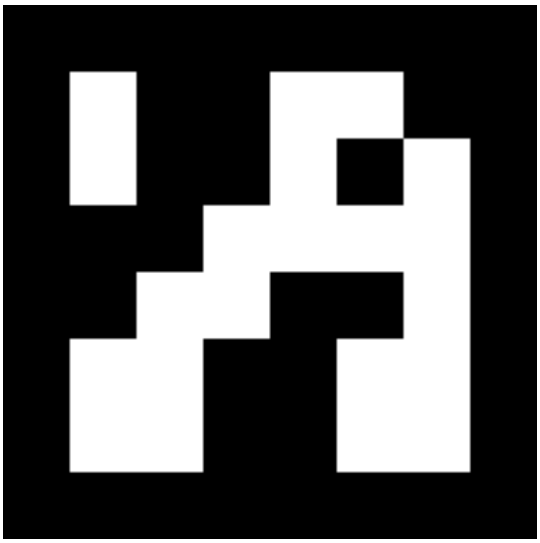


Figure 3: Caption

And these are the 2 pictures with the cube projected on top of the marker:

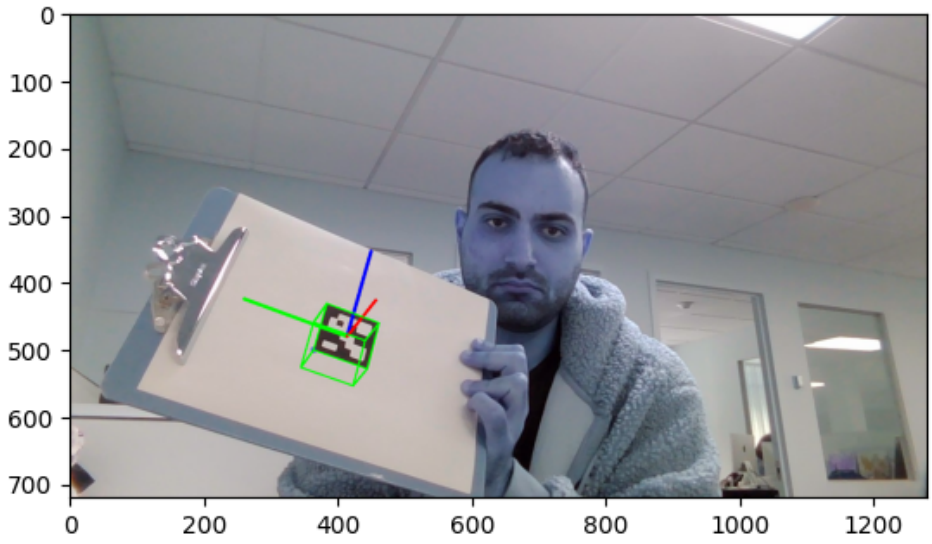



Figure 4: Caption

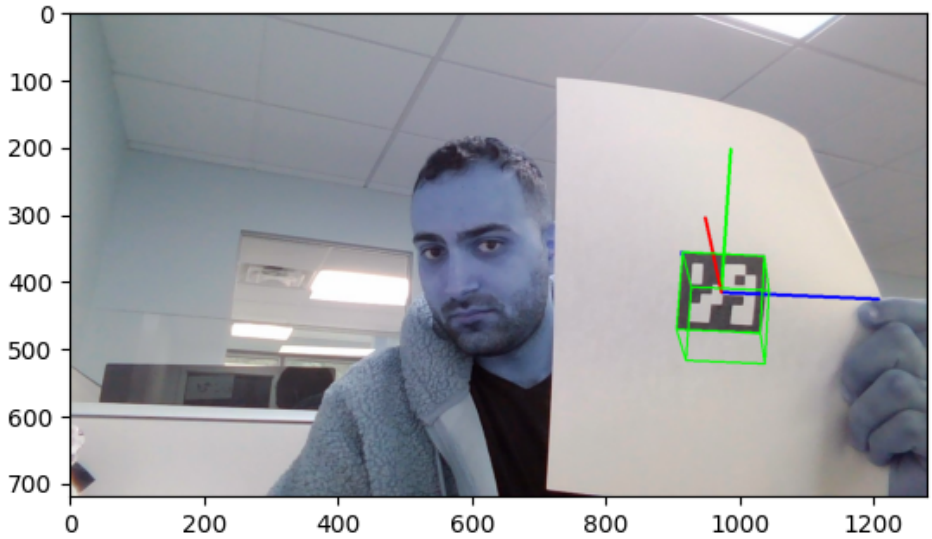



Figure 5: Caption

For detailed explanation of each step of the code and the code itself can be found in https://github.com/trollzero/ROB-GY-6203-Robot-Perception-F23-NYU/tree/main/Homework in homework 1 and in the HW1Q3-4 CalibrationBoard notebook.

In essence, we import cv2 and aruco from cv2. We read in the image of the aruco tag and the we use cv2.aruco.getPredefinedDictiona 6X6 250) to define the dictionary to be used by the aruco function when detecting the markers. We use parameters = cv2.aruco.DetectorParameters() for the same reason. We then get the corners and the marker id using the function corners, ids, unused = cv2.aruco.detectMarkers(image, aruco dict, parameters=parameters).
Then for all markers (in this case 1) we use aruco.drawDetectedMarkers(image, corners) to identify and draw a box around the marker to show it has been identified.

Then we use ret, rvec, tvec = cv2.solvePnP(object points, corners[i], K, distortion) where the function takes the marker corners as image points, the object points defined by the user, the distrotion parameters and the Calibration matrix to get the translation and rotation vectors so it can estimate the pose/orientation of the marker.
We then use cv2.drawFrameAxes(image, K, None, rvec, tvec, 0.1) which takes in the image, the calibration matrix, the rotation and translation vectors and the length of the axis to draw the frame axes on the image centered on the Aruco marker.
We then define the 8 corners of the cube and use cv2.projectPoints(cube points, rvec, tvec, K, distortion) to project them onto the frame axis/ orientation of the Aruco marker. The function takes in the cube corners, the translation and rotation vectors, the distortion parameters and the calibration matrix.
We then define the connections between the vertices as lines to be drawn and then looping through all these lines we use cv2.line(image, start point, end point, (0, 255, 0), 2) which takes the image, the start point, the end point and the colour of the line as well as the thickness to draw the lines. The start and end points are the 0th and 1st element in the tuples in the list called lines we defined above.
We then write the image and use plt.imshow to display it in grayscale because google colab breaks with cv2.imshow.
If you need even more explanation you can find a lot more in the google colab notebook given in the link above and if you need the pictures to recreate the results I can provide a google drive link.
In summary the steps are, calibrate the camera, take a picture with an Aruco Marker, use the dictionary of the marker, find the corners of the marker, find the rotation and translation matrix to find its orientation/pose, draw the frame axes, define the cube vertices, project them in the orientation of the Aruco Marker. Define the vertices to be connected, draw the lines, write the new image to a file and present it. Make sure the line lengths and thickness of the frames, and the cube are appropriate.
PS: functions are not exactly the same as in the google colab notebook because it won't let me use an underscore.