MTRX5700 - Assignment Task 2 Report

Experimental Robotics

SIDs:
470148329
470416309
480436153

Question 1 - Hand Eye Calibration

Theoretical Analyses

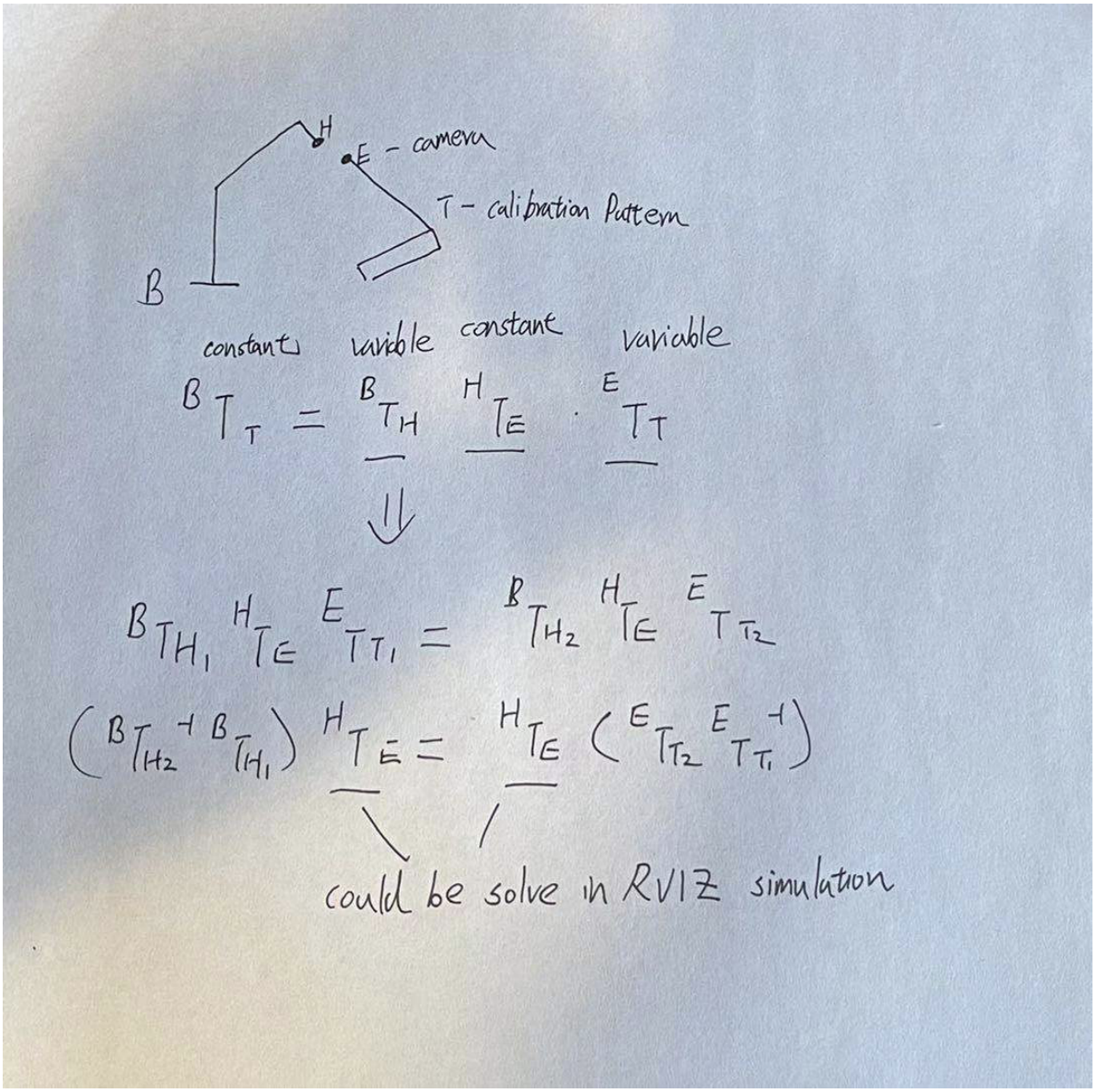


Figure 1 - Theoretical Analysis

Work on RVIZ

Baes on the RVIZ, firstly, setting up the parameters of HandEye calibration and then set UR5e arm in the middle position as shown below and taking a sample:

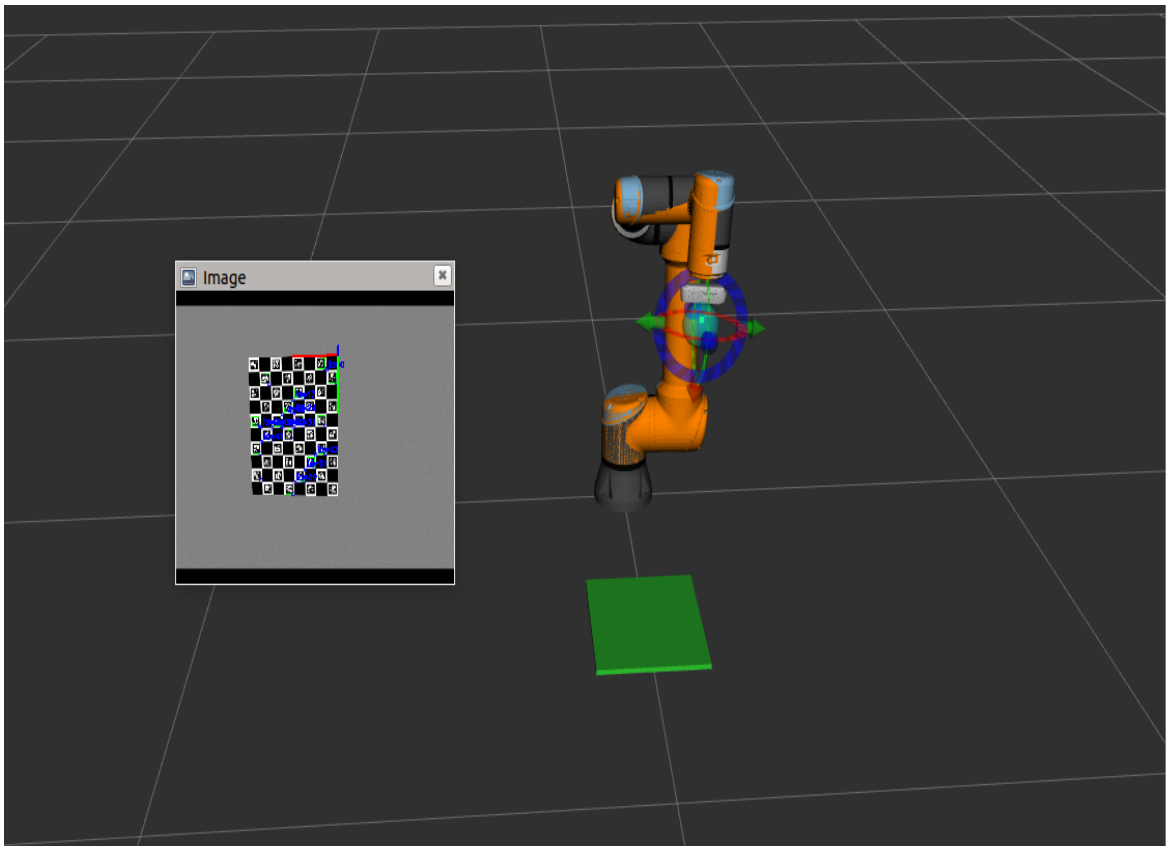
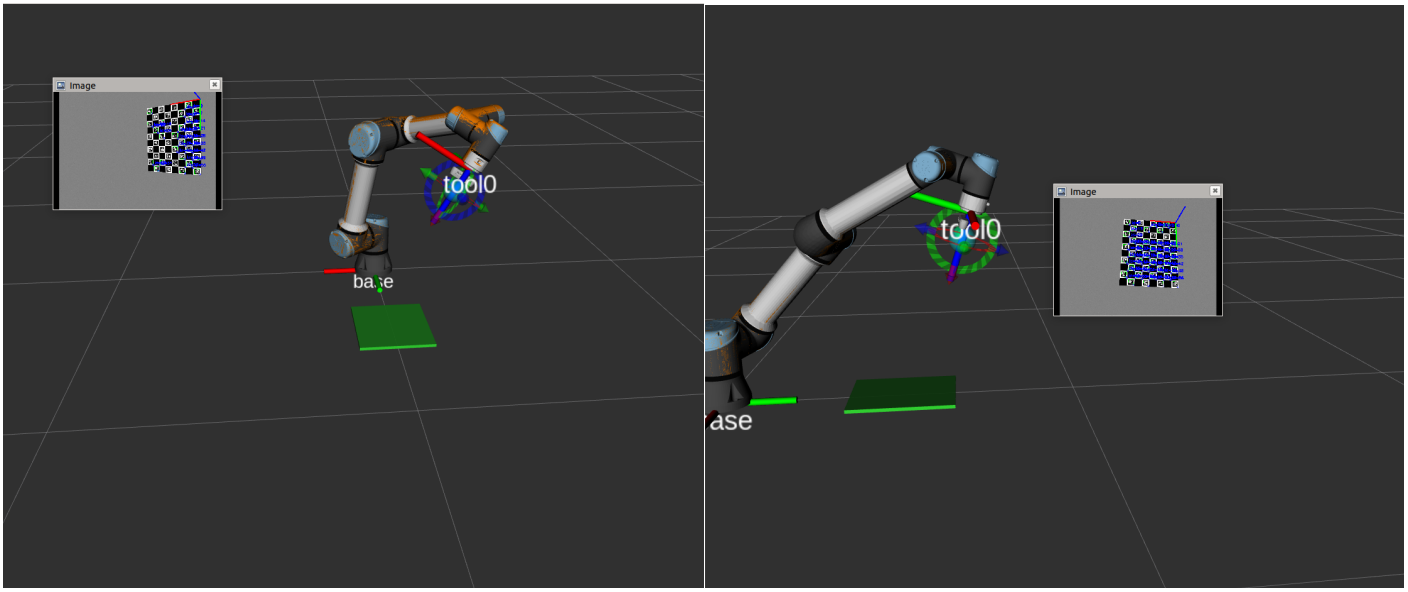


Figure 2 - Located the UR5 Arm in the Middle Position

Second is move the arm in different locations but need to keep the calibration pattern fully observed and some special and key positions are shown below and 30 samples have been taken to increase the accuracy.



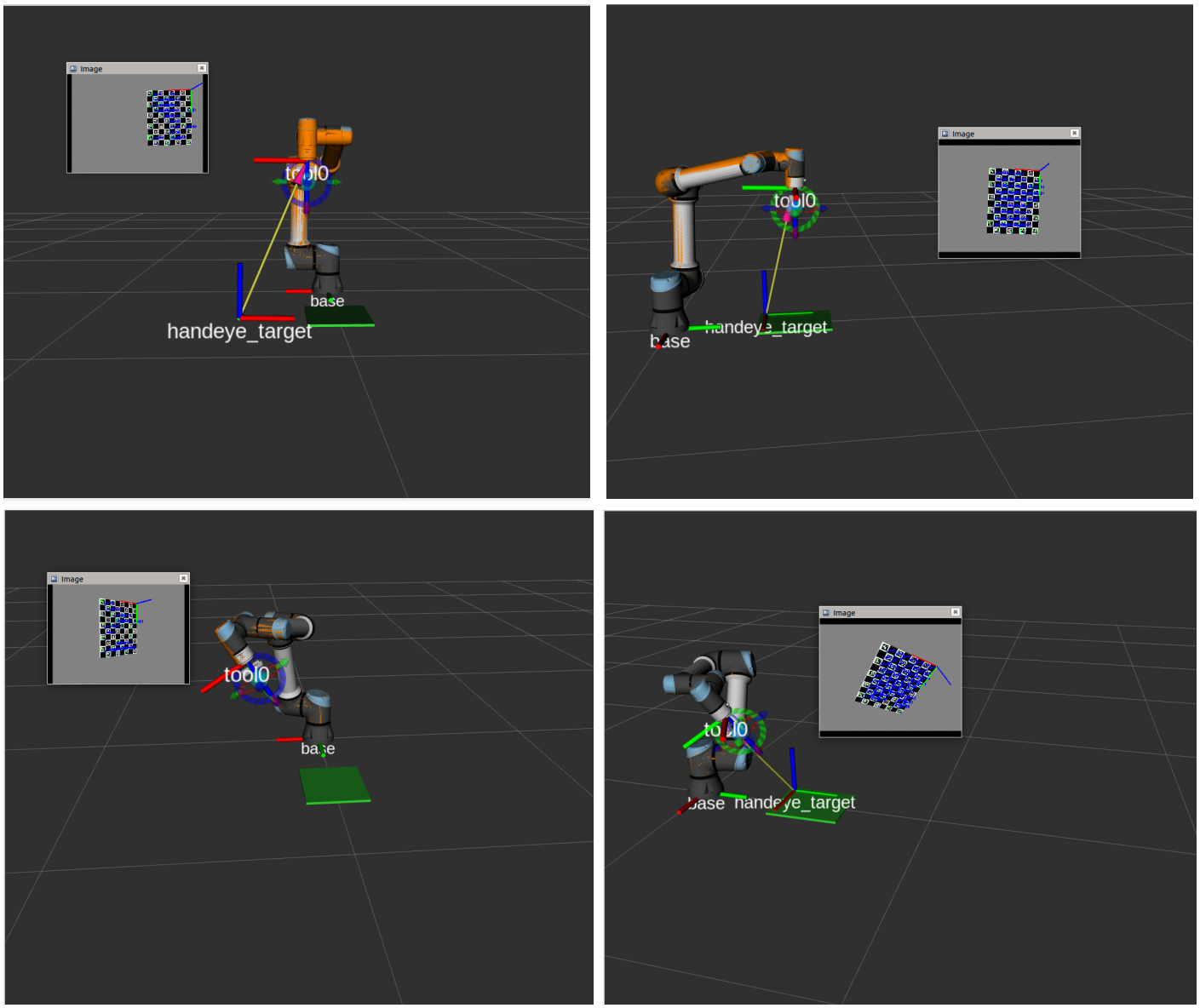


Figure 3 - From the Different to Take Location

Last one is select solve and whis will give the final result of ${}^{H}T_{E}$ and three methods result is shown below:

crigroup/Daniilidis 1999

```
          -1 -0.000267438 -0.000332869    0.0172432
 0.000267732           -1 -0.000883878    0.0126004
-0.000332632 -0.000883967           1    0.0300778
           0            0           0            1
```

crigroup/ParkBryan 1994

```
          -1 -0.000267438 -0.000332869    0.0172432
 0.000267732           -1 -0.000883878    0.0126004
-0.000332632 -0.000883967           1    0.0300778
           0            0           0            1
```

crigroup/TsaiLenz 1989

```
-0.997851 -0.0652883 0.00548093  0.0163231
0.0653405  -0.997814 0.00993659 0.00909533
0.0048202  0.0102734   0.999936   0.035291
        0          0          0          1
```

And the final location coordinate of target and camera are correct as the figure shown below:

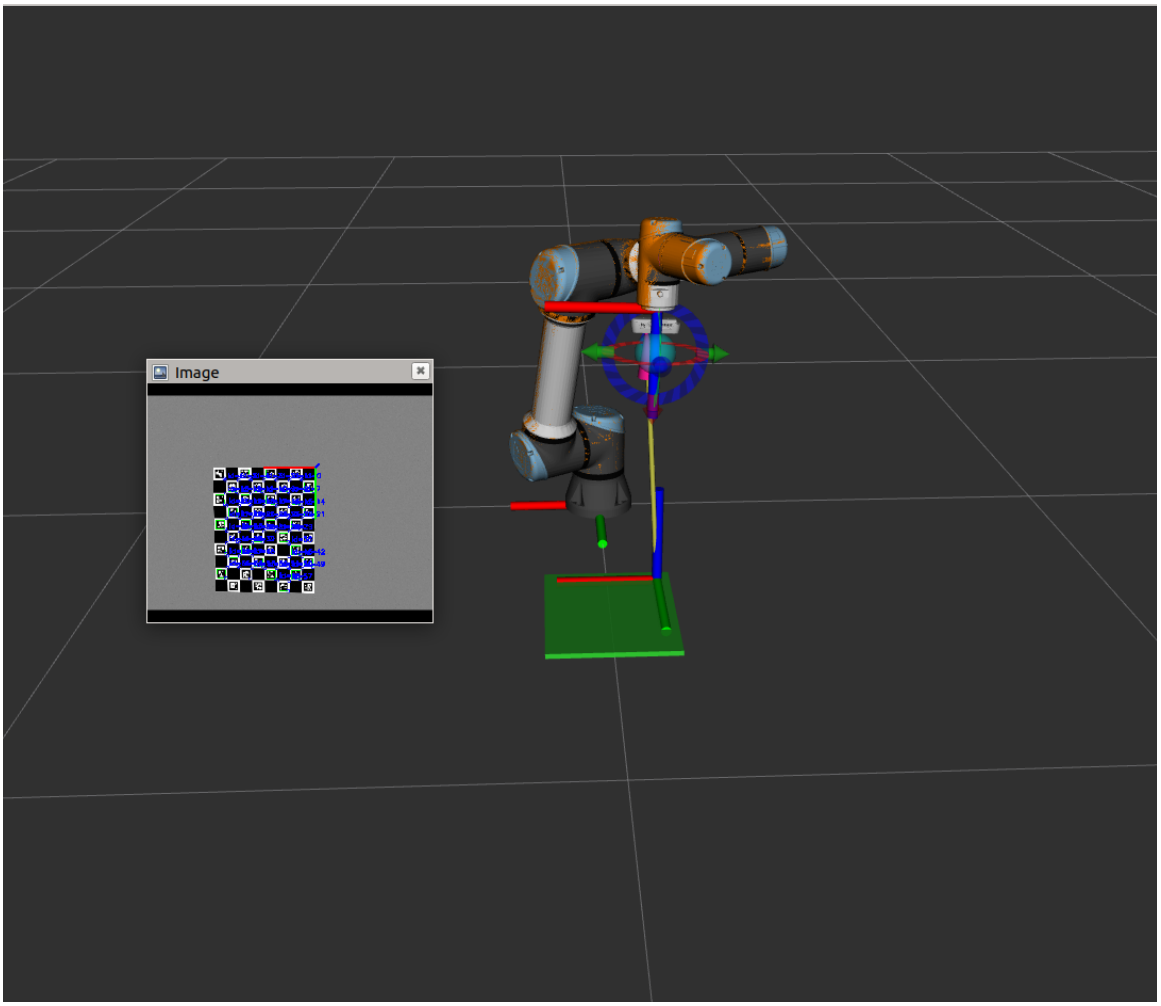


Figure 4 - The Final Coordinate of Target and Camera

Question 2 - Point Cloud Registration

Iterative Closest Point (ICP)

a) Point Cloud Poses

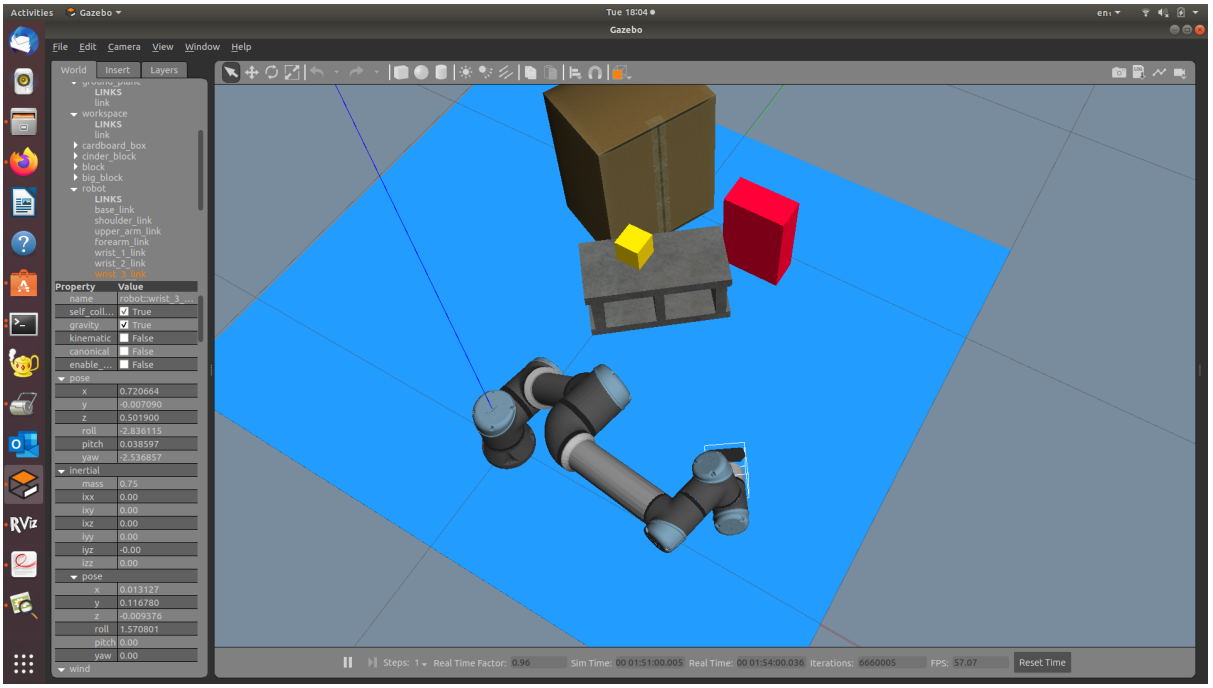


Figure 5 - Pose 1 (End Effector Data)

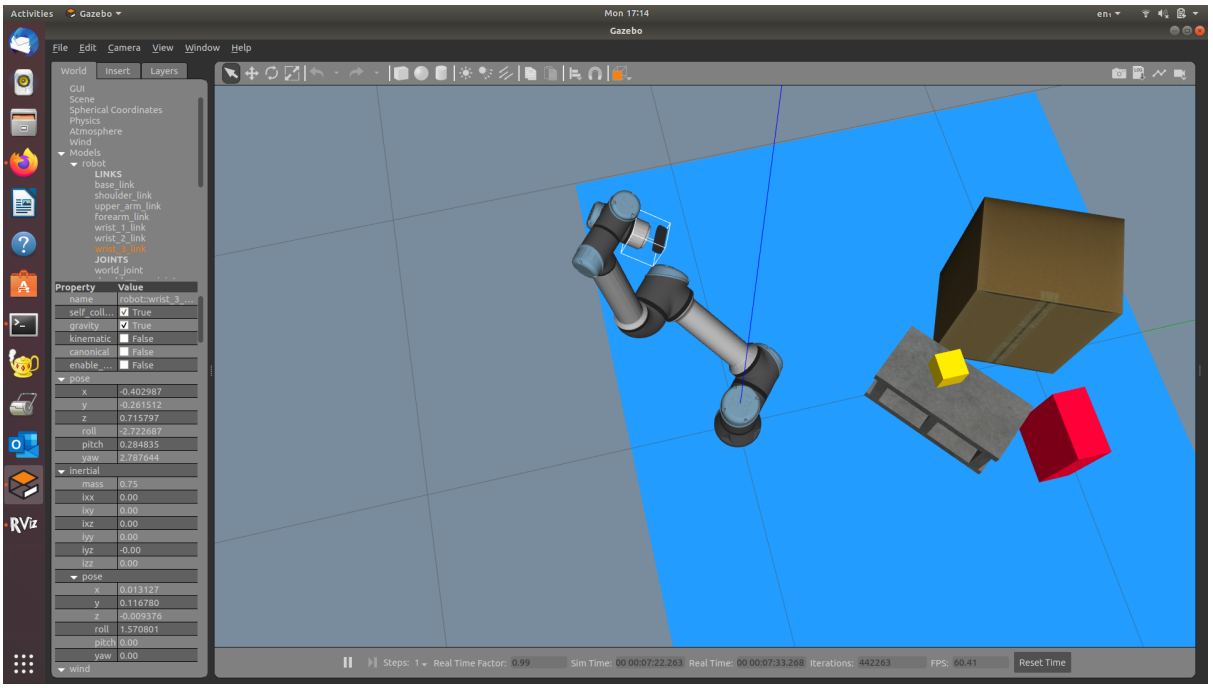


Figure 6 - Pose 2 (End Effector Data)

b) MATLAB Two Point Cloud Visualisation

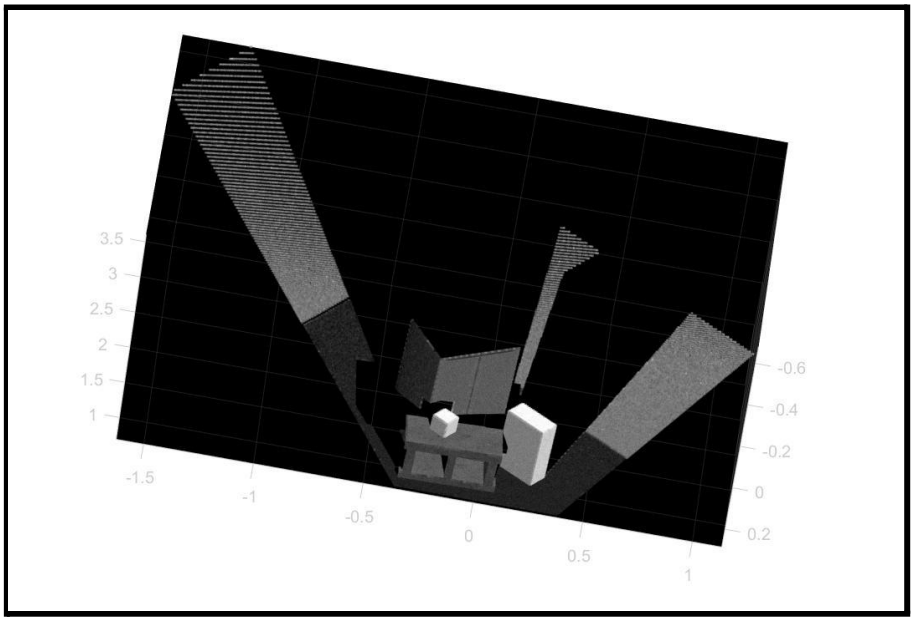


Figure 7 - End-Effector Pose 1 ($T_E^{B^1}$)

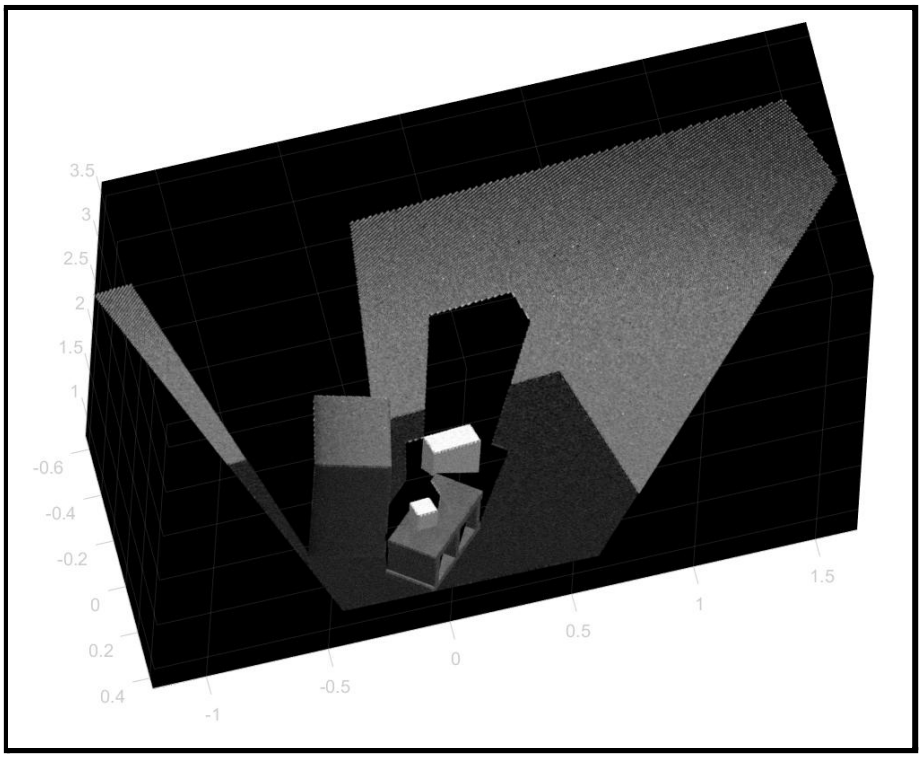


Figure 8 - End-Effector Pose 2 ($T_E^{B^2}$)

c) Results

Motion of the Camera

From Question 1 - it has been assumed that the end-effector to camera transformation obtained is: $T_E^C =$

-0.99785	-0.06529	0.005481	0.016323
0.065341	-0.99781	0.009937	0.009095
0.00482	0.010273	0.999936	0.035291
0	0	0	1

Furthermore, the poses of the end-effector have a defined transformation matrix as follows (respectively **pose one** and **pose two**):

Pose One $T_E^{B^1} =$

0.999	0.0443	0.0007	0.7207
-0.0442	0.9978	0.0495	-0.0071
0.0015	-0.0495	0.9988	0.5019
0	0	0	1

Pose Two $T_E^{B^2} =$

0.9988	-0.04863	0.00497	-0.40299
0.048341	0.997733	0.047499	-0.26151
-0.00265	0.047685	0.99888	0.715797
0	0	0	1

The motion of the camera therefore, can be computed using the following equation:

$$M_2^1 = (T_E^{B^1} \cdot T_C^E)^{-1} \cdot (T_E^{B^2} \cdot T_C^E)$$

Equation 1 - Computation of the Motion of the Camera

This resulting motion of the camera hence is as follows:

0.995692	-0.09286	-0.00349	1.088374
0.092901	0.99005	0.001178	0.400817
0.004968	-0.09627	1.000955	0.191089
0	0	0	1

Point Cloud Registration

The transformation matrix defined utilizing Two Point Cloud Registration within MATLAB (with similar formatting to the above matrices) resulted to be:

0.9727	-0.1886	-0.00639	0.9999
0.09616	0.9729	0.00137	0.2349
0.00843	-0.135	0.9777	0.1609
0	0	0	1

Computation of Errors

Assuming the motion of the camera that was derived from *Equation 1*, is the true expressional representation, the following formula was employed to calculate the error between the true and point cloud registered or ICP derived matrix values:

$$Error \ = \frac{ICP\ Value - True\ Matrix\ Value\ (Motion\ of\ Camera)}{True\ Matrix\ Value\ (Motion\ of\ Camera)} \times 100$$

Furthermore, the following matrix displays the error between M_2^1 and ICP method, accomplished by subtracting the values of the matrix respectively:

- The **angle of rotation error** can be displayed as:

Rotation	X	Y	Z
X	0.022992	0.095737	0.002901
Y	-0.00326	0.01715	-0.00019
Z	-0.00346	0.038732	0.023255

- The **norm of the translation error** can be displayed as:

Translation

0.0884742

0.1659169

0.0301891

$$translation\ error\ =\ 17.37\%$$

Question 3 - Image Classification using Deep Learning

a) Initial Experimenting

i) Number of samples in the German Traffic Signs Dataset: 3000

ii) 1500 of the samples are organized into the training set, while 420 samples are loaded to the validation set and 1080 samples are loaded to the test set.
   a) Training Data: Data which the network can familiarise itself with the information.
   b) Test Data: Data which is used by the network for testing.
   c) Validation Data: Data which is used upon each iteration of the training dataset to investigate the network's performance.

iii) As shown below, there are **five** classes in the dataset, including: *Stop*, *Right*, *Left*, *Forward* and *Roundabout*. In the three datasets, there are in total 643 *Stop* signs, 562 *Right* signs, 485 *Left* signs, 842 *Forward* signs, and 468 *Roundabout* signs.

Figure 9 - Output Traffic Signs

iv) Dimensions of the images in pixels: 32×32 pixels

b) Deep Nerual Network

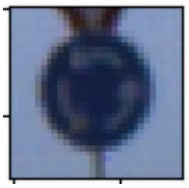
We defined our neural network based on the example provided on the Pytorch Tutorials [1]. During the training loop, we trained the model based on both the training dataset and the validation dataset, with a dataset batch size of 5 and a minibatch of 200, using 5 epochs. The overall performance of the model when experimented on the test dataset reaches an accuracy of 97%.

c) Confusion Matrix

The results of the neural network prediction based on the training dataset is shown below. The model was operated using a batch size of five and a mini-batch of 50. As we can see, the prediction is somewhat accurate apart from several prediction failures. Failed predictions often occur when the image is dark or blurry. The network

cannot predict such images likely because they have a low colour contrast, which is one of the key characteristics during image classification, and hence the network fails to accurately identify their labels.

Table 1 - Confusion Matrix

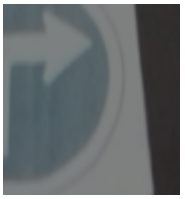
Actual Label	**Predicted Label**	**Image** ***(if it doesn't match)***
Stop	Stop	
Forward	Right	
Forward	Forward	
Forward	Forward	
Stop	Stop	
Stop	Stop	
Forward	Right	
Roundabout	Roundabout	
Right	Right	
Left	Left	
Forward	Forward	
Forward	Roundabout	
Roundabout	Left	

d) External Input

We extracted some traffic signs from the rosbag file given in Question 4, preprocessed the images and inputted to the trained model, and below are some of the outputs. There are two failed predictions, as we can see from the table below, the first row and the last row. We observed that all samples in the training datasets had a medium to low brightness, whereas in this failed case the image has a high brightness compared to most of the samples in the training dataset, and hence the model might not be trained to handle bright pictures. This is the same for the case at the last row, where the image has a high brightness but is also captured at an angle, hence more difficult for the model to predict.

Table 2 -

Actual Label	Predicted Label	Image
Right	Roundabout	
Stop	Stop	
Right	Right	
Roundabout	Roundabout	
Right	Right	

Right	Roundabout	

Question 4 - Perception for Robots

Navigate Duckietown

a) Process of Navigation

The process of identifying the traffic signs through the manufactured model town (Duckietown) was achieved by firstly registering and detecting a **red** line. This line acts as an indicator within the system, highlighting to the program that an oncoming traffic sign on the surface will be present and requires detection. Furthermore, the red line coordinates indicate to the system to trigger the binary **white** colour detection. When this colour is detected, its coordinates can be found using further OpenCV functions and the frame of that traffic sign can be grabbed. This is accomplished alongside the red line disappearing from the frames, triggering the frame capture of the traffic sign to be centralised and fully present in the saving of that particular image/frame.

Finally, this frame must undergo some further processing prior to it being applied to the neural network - this includes transforming the trapezial like traffic sign (since the camera is slightly above the surface and is in a parallel perspective to the surface) to a more square image and reducing the captured image to 32×32 a pixel image.

From this stage the neural network can then detect the label, predict the traffic sign and display the image, label and traffic sign as an output.

Upon testing the program, many alterations were made to improve the performance of the system in terms of efficiency and accuracy - this included reducing the contrast and brightness of the images prior to detection of the red line. It was evident that this reduced the glare present on the red line and hence, the number of frames that are saved will be kept to a minimal (as there will be a more clear indication as to when the line is present verses when it is not - contrast between the red, black and white aspects of the images).

Alike, to reduce the number of frames captured to a minimal or ideally only one frame for each traffic sign, a timer function was planned to be employed but was not due to time restrictions. This would limit the amount of captures for each traffic sign and hence, increase the accuracy of the system especially with Question 5.

Question 5 - Demo

- Please refer to video attachment entitled:
    - *Question 5*

Appendix

Codes for Question 3-4

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import pickle
import matplotlib.pyplot as plt
import cv2
import torch
import time
import PIL
from PIL import Image
import numpy as np
import tensorflow as tf
import torch.nn as nn
import torch.nn.functional as functional

from torch import nn, optim
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
import torchvision
from torchvision import transforms
from torchvision.transforms import ToTensor
from torchvision.transforms import functional as fn



import rosbag
import roslib   #roslib.load_manifest(PKG)
import rospy

from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from scipy.ndimage import filters

font = cv2.FONT_HERSHEY_SIMPLEX

training_file = "/home/ob1-knb/ur5/src/traindata.pkl"
validation_file = "/home/ob1-knb/ur5/src/validdata.pkl"
testing_file = "/home/ob1-knb/ur5/src/testdata.pkl"
xtra_file = "/home/ob1-knb/ur5/src/extradata.pkl"

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)
```

```python
with open(xtra_file, mode='rb') as f:
    xtra = pickle.load(f)


x_train, y_train = train['features'], train['labels']
x_valid, y_valid = valid['features'], valid['labels']
x_test, y_test = test['features'], test['labels']
x_xtra, y_xtra = xtra['features'], xtra['labels']

# Custom Dataloader Class
class TrafficSignsDataset(Dataset):
    def __init__(self, features, labels, transform=None):
        self.features = features
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.features)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        x = self.features[idx]
        y = self.labels[idx]

        if self.transform:
            x = self.transform(x)
            y = y

        return x, y


# Define custom transforms to apply to the images
# -----------------------------------------------
# Here we are applying a normalization to bring all pixel values between 0 and 1
# and converting the data to a pytorch tensor
custom_transform = transforms.Compose([transforms.ToTensor(),
                                       transforms.Normalize((0.5,0.5,0.5 ), (0.5,0.5,0.5))])

# Define the dataloaders
# ----------------------
# Dataloaders help in managing large amounts of data efficiently
```

```
batch_size = 5

train_dataset = TrafficSignsDataset(x_train, y_train,
                                    custom_transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

validation_dataset = TrafficSignsDataset(x_valid, y_valid,
                                         custom_transform)
validation_loader = DataLoader(validation_dataset, shuffle=True)

test_dataset = TrafficSignsDataset(x_test, y_test,
                                   custom_transform)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

xtra_dataset = TrafficSignsDataset(x_xtra, y_xtra,
                                   custom_transform)
xtra_loader = DataLoader(xtra_dataset, batch_size=batch_size, shuffle=True)



classes = ('Stop', 'Right', 'Left', 'Forward', 'Roundabout')

def imshow(img):
    img = img / 2 + 0.5

dataiter = iter(train_loader)
images, labels = dataiter.next()

print("Number of training samples is {}".format(len(train_dataset)))
print("Number of test samples is {}".format(len(test_dataset)))
print("Number of validation samples is {}".format(len(validation_dataset)))

print("Batch size is {}".format(len(images)))
print("Size of each image is {}".format(images[0].shape))

print("The labels in this batch are: {}".format(labels))
print("These correspond to the classes: {}, {}, {}, {}, {}".format(
    classes[labels[0]], classes[labels[1]],
    classes[labels[2]], classes[labels[3]], classes[labels[4]]))

#############################################################################
# Step 2: Building your Neural Network
#-----------------------------------
```

```
class Network(nn.Module):
  def __init__(self):
    self.output_size = 5   # 5 classes

    super(Network, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)             # 2D convolution
    self.pool = nn.MaxPool2d(2, 2)              # max pooling
    self.conv2 = nn.Conv2d(6, 16, 5)            # 2D convolution
    self.fc1 = nn.Linear(16 * 5 * 5, 120)       # Fully connected layer
    self.fc2 = nn.Linear(120, 84)               # Fully connected layer
    self.fc3 = nn.Linear(84, self.output_size)  # Fully connected layer

  def forward(self, x):
    """Define the forward pass."""
    x = self.pool(functional.relu(self.conv1(x)))
    x = self.pool(functional.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = functional.relu(self.fc1(x))
    x = functional.relu(self.fc2(x))
    x = self.fc3(x)
    return x

net = Network()

#########################################################################################################
# Step 3: Defining loss function and optimizer
#--------------------------------------------

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

#########################################################################################################
# Step 4: Main training loop
#---------------------------

for epoch in range(5):
  running_loss = 0.0
  val_loss = 0.0

  for i, data in enumerate(validation_loader, 0):

    inputs, labels = data
    optimizer.zero_grad()
```

```
        predicted_labels = net(inputs)
        loss = criterion(predicted_labels, labels)
        loss.backward()
        optimizer.step()

        val_loss += loss.item()
        if i % 200 == 199:    # print every 200 mini-batches
            print('Epoch: %d, %5d val loss: %.3f' % (epoch + 1, i + 1, val_loss / 50))
            val_loss = 0.0

    for i, data in enumerate(train_loader, 0):

        inputs, labels = data
        optimizer.zero_grad()
        predicted_labels = net(inputs)
        loss = criterion(predicted_labels, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 200 == 199:
            print('Epoch: %d, %5d loss: %.3f' % (epoch + 1, i + 1, running_loss / 50))
            running_loss = 0.0

print('Finished Training.')
torch.save(net.state_dict(), "traffic_signs_classifier.pt")
print('Saved model parameters to disk.')

############################################################################################
# Step 5: Duckietown Experiment
#-----------------------------------------------------------------

model = Network()
model.load_state_dict(torch.load('traffic_signs_classifier.pt'))
model.eval()

bag_file = '/home/ob1-knb/ur5/src/bag1.bag'
bag = rosbag.Bag(bag_file, "r")

bridge = CvBridge()
bag_data = bag.read_messages('/raspicam_node/image/compressed')

for topic, msg, t in bag_data:
    cv_image = bridge.compressed_imgmsg_to_cv2(msg, "bgr8")
```

```python
for topic, msg, t in bag_data:
    cv_image = bridge.compressed_imgmsg_to_cv2(msg, "bgr8")

    dst = cv2.rotate(cv_image, cv2.ROTATE_180)
    original = dst
    alpha = 0.5
    beta = 0
    res = cv2.convertScaleAbs(dst,alpha=alpha,beta=beta)
    HSVImage_imp = cv2.cvtColor(res, cv2.COLOR_BGR2HSV)
    Low_red_range = np.array([140,50,50])
    High_red_range = np.array([180,255,255])
    Low_white_range = np.array([100,43,46])
    High_white_range = np.array([124,255,255])

    mask_red = cv2.inRange(HSVImage_imp,Low_red_range,High_red_range)
    mask_red = cv2.medianBlur(mask_red, 7)

    mask_red, contours2, hierarchy2 = cv2.findContours(mask_red, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

    for cnt2 in contours2:
        (x2, y2, w2, h2) = cv2.boundingRect(cnt2)

        if  (y2>308 and y2<315) and w2 > 3*h2 :

            crop_img = HSVImage_imp[220:300,130:490]
            mask_white = cv2.inRange(crop_img,Low_white_range,High_white_range)
            mask_white = cv2.medianBlur(mask_white, 7)  # 中值滤波
            mask_red1 = cv2.inRange(crop_img,Low_red_range,High_red_range)
            mask_red1 = cv2.medianBlur(mask_red1, 7)  # 中值滤波
            mask = cv2.bitwise_or(mask_red1,mask_white)
            mask_white, contours1, hierarchy1 = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

            for cnt1 in contours1:
                (x, y, w, h) = cv2.boundingRect(cnt1)
                crop_img_1 = crop_img[y:y+60,x:x+120]
                RGB_img_1 = cv2.cvtColor(crop_img_1,cv2.COLOR_HSV2BGR)

            resize_img = cv2.resize(RGB_img_1,(32,32))


            resize_img = resize_img[:, :, [2,1,0]]
            # resize_img = resize_img[:,:,::-1]

```

```
            RGB_img_1 = cv2.resize(RGB_img_1,(100,100))

            cv2.imshow("Image window",RGB_img_1)

            x = ToTensor()(resize_img)
            x = transforms.Normalize((0.5,0.5,0.5 ), (0.5,0.5,0.5))(x)
            x.unsqueeze_(0)

            outputs = model(x)
            sm = nn.Softmax(dim=1)
            sm_outputs = sm(outputs)

            probs, index = torch.max(sm_outputs, dim=1)
            for p, i in zip(probs, index):
                print('{}'.format(classes[i]))
                sign = classes[i]
                cv2.putText(original, '{}'.format(sign), (240,220), font, 2, (0,0,0)
                time.sleep(0.5)

    cv2.imshow("Image window", original)


    cv2.waitKey(3)

##########################################################################################
# Step 6: Evaluate the accuracy of your network on the test dataset
#-----------------------------------------------------------------

dataiter = iter(train_loader)
images, labels = dataiter.next()

fig, ax = plt.subplots(1, len(images))
for id, image in enumerate(images):
  # convert tensor back to numpy array for visualization
  ax[id].imshow((image / 2 + 0.5).numpy().transpose(1,2,0))
  ax[id].set_title(classes[labels[id]])
plt.show()
print(' '.join('%5s' % classes[labels[j]] for j in range(batch_size)))

outputs = model(images)

```

```
sm = nn.Softmax(dim=1)
sm_outputs = sm(outputs)

probs, index = torch.max(sm_outputs, dim=1)
for p, i in zip(probs, index):
    print('True label {0}, Predicted label {0} - {1:.4f}'.format(classes[i], p))

correct = 0
total = 0
with torch.no_grad():
    for data in validation_loader:
        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Correct predictions: {}'.format(correct))
print('Total predicted: {}'.format(total))
print('Number per class: ')
print(stop)
print(right)
print(left)
print(forward)
print(round)
print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

```

Codes for Question 5

#!/usr/bin/env python3
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
import pickle
import matplotlib.pyplot as plt
import cv2
import torch
import time
from PIL import Image
import numpy as np
import tensorflow as tf
import torch.nn as nn
import torch.nn.functional as functional
import sys, time
from torch import nn, optim
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
import torchvision
from torchvision import transforms

```
from torchvision.transforms import ToTensor
from scipy.ndimage import filters
import rosbag
import roslib   #roslib.load_manifest(PKG)
import rospy
from sensor_msgs.msg import CompressedImage
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from scipy.ndimage import filters
from geometry_msgs.msg import Twist
PI = 3.1415926535897


training_file = "/home/max/assignment2/src/traindata.pkl"
validation_file = "/home/max/assignment2/src/validdata.pkl"
testing_file = "/home/max/assignment2/src/testdata.pkl"


with open(training_file, mode='rb') as f:
        train = pickle.load(f)
with open(validation_file, mode='rb') as f:
        valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
        test = pickle.load(f)

x_train, y_train = train['features'], train['labels']
x_valid, y_valid = valid['features'], valid['labels']
x_test, y_test = test['features'], test['labels']

# Custom Dataloader Class
class TrafficSignsDataset(Dataset):
        def __init__(self, features, labels, transform=None):
        self.features = features
        self.labels = labels
        self.transform = transform

        def __len__(self):
        return len(self.features)

        def __getitem__(self, idx):
        if torch.is_tensor(idx):
        idx = idx.tolist()

        x = self.features[idx]
        y = self.labels[idx]

        if self.transform:
        x = self.transform(x)
```

```
        y = y

        return x, y


# Define custom transforms to apply to the images
# -----------------------------------------------
# Here we are applying a normalization to bring all pixel values between 0 and 1
# and converting the data to a pytorch tensor
custom_transform = transforms.Compose([transforms.ToTensor(),
                            transforms.Normalize((0.5,0.5,0.5 ), (0.5,0.5,0.5))])

# Define the dataloaders
# ----------------------
# Dataloaders help in managing large amounts of data efficiently
batch_size = 5

train_dataset = TrafficSignsDataset(x_train, y_train,
                            custom_transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

validation_dataset = TrafficSignsDataset(x_valid, y_valid,
                                    custom_transform)
validation_loader = DataLoader(validation_dataset, shuffle=True)

test_dataset = TrafficSignsDataset(x_test, y_test,
                            custom_transform)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)



classes = ('Stop', 'Right', 'Left', 'Forward', 'Roundabout')


class Network(nn.Module):
  def __init__(self):
        self.output_size = 5   # 5 classes

        super(Network, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)                 # 2D convolution
        self.pool = nn.MaxPool2d(2, 2)          # max pooling
        self.conv2 = nn.Conv2d(6, 16, 5)                # 2D convolution
        self.fc1 = nn.Linear(16 * 5 * 5, 120)   # Fully connected layer
        self.fc2 = nn.Linear(120, 84)           # Fully connected layer
        self.fc3 = nn.Linear(84, self.output_size)  # Fully connected layer

  def forward(self, x):
        """Define the forward pass."""
```

```
        x = self.pool(functional.relu(self.conv1(x)))
        x = self.pool(functional.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = functional.relu(self.fc1(x))
        x = functional.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Network()

#########################################################################
############################

#########################################################################
############################
# Step 3: Defining loss function and optimizer
#-----------------------------------------

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

#########################################################################
############################
# Step 4: Main training loop
#-----------------------------------------

for epoch in range(6): # we are using 5 epochs. Typically 100-200
  running_loss = 0.0
  for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # Perform forward pass and predict labels
        predicted_labels = net(inputs)

        # Calculate loss
        loss = criterion(predicted_labels, labels)

        # Perform back propagation and compute gradients
        loss.backward()

        # Take a step and update the parameters of the network
        optimizer.step()

        # print statistics
```

```
            running_loss += loss.item()
            if i % 300 == 299:       # print every 200 mini-batches
            print('Epoch: %d, %5d loss: %.3f' % (epoch + 1, i + 1, running_loss / 50))
            running_loss = 0.0

print('Finished Training.')
torch.save(net.state_dict(), "traffic_signs_classifier.pt")
torch.save(net, '/home/max/assignment2/cifar_2.pt')
print('Saved model parameters to disk.')

############################################################################
############################
# Step 5: Evaluate the accuracy of your network on the test dataset
#-----------------------------------------------------------------

model = Network()
model.load_state_dict(torch.load('traffic_signs_classifier.pt'))
model.eval()

class image_feature:

            def __init__(self):
            # subscribed Topic
            self.subscriber = rospy.Subscriber('/raspicam_node/image/compressed',
            CompressedImage,self.callback,queue_size=1)

            def callback(self, ros_data):
            velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=1)
            angular_speed = 90*2*PI/360
            relative_angle = 90*2*PI/360
            vel_msg = Twist()
            vel_msg.linear.x = 0.1
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            vel_msg.angular.z = 0
            velocity_publisher.publish(vel_msg)

            font = cv2.FONT_HERSHEY_SIMPLEX
            #### direct conversion to CV2 ####
            np_arr = np.fromstring(ros_data.data, np.uint8)
            image_np = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)  #original image
            image_np = cv2.rotate(image_np, cv2.ROTATE_180)         #rotate
            alpha = 0.7      # change the contrast
            beta = 0

            # eliminate glare
```

```
        R, G, B = cv2.split(image_np)
        output1_R = cv2.equalizeHist(R)
        output1_G = cv2.equalizeHist(G)
        output1_B = cv2.equalizeHist(B)
        equ = cv2.merge((output1_R, output1_G, output1_B))
        res = np.hstack((image_np,equ)) #stacking images side-by-side

        # change the contrast use to test the redline
        adjusted = cv2.convertScaleAbs(image_np, alpha=alpha, beta=beta)
        HSVImage = cv2.cvtColor(adjusted,cv2.COLOR_BGR2HSV) # for redline detection
        HSVImage_imp = cv2.cvtColor(res,cv2.COLOR_BGR2HSV)  #for traffic signal
recognion

        Low_red_range = np.array([140,50,50])
        High_red_range = np.array([180,255,255])
        Low_white_range = np.array([80,35,70])
        High_white_range = np.array([135,230,255])

        mask_red = cv2.inRange(HSVImage,Low_red_range,High_red_range)
        mask_red = cv2.medianBlur(mask_red, 7)  # Median fifler
        contours2, hierarchy2 = cv2.findContours(mask_red, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)

        for cnt2 in contours2:
        (x2, y2, w2, h2) = cv2.boundingRect(cnt2)
        if w2>3*h2 and (y2==308 or y2==307 or y2 == 314): # take one flame each signal
                crop_img = HSVImage_imp[200:310,190:490]
                mask_white = cv2.inRange(crop_img,Low_white_range,High_white_range)
                mask_white = cv2.medianBlur(mask_white, 7)   # Median fifler
                mask_red1 = cv2.inRange(crop_img,Low_red_range,High_red_range)
                mask_red1 = cv2.medianBlur(mask_red1, 7)   # Median fifler
                mask = cv2.bitwise_or(mask_red1,mask_white)
                cv2.imshow('mask_white', mask)
                contours1, hierarchy1 = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)
                for cnt1 in contours1:
                (x, y, w, h) = cv2.boundingRect(cnt1)
                crop_img_1 = crop_img[y:y+60,x:x+120]
                BGR_img_1 = cv2.cvtColor(crop_img_1,cv2.COLOR_HSV2BGR)
                cv2.imwrite('traffic_image_original_1.png',BGR_img_1)
                resize_img = cv2.resize(BGR_img_1,(32,32))
                cv2.imwrite('traffic_image_2.png',resize_img)

                # use for model detect
                resize_img_1 = resize_img[:,:,[2,1,0]]
                x = ToTensor()(resize_img_1)
                x = transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))(x)
                x.unsqueeze_(0)
```

```
            outputs = model(x)
            sm = nn.Softmax(dim=1)
            sm_outputs = sm(outputs)
            _, predicted = torch.max(sm_outputs,dim=1)
            print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                                for j in range(1)))
            sign = classes[predicted]
            cv2.putText(image_np,format(sign), (240,220),font,2, (0,0,0),2, cv2.LINE_AA)

            if sign == "Stop":
            vel_msg.linear.x = 0
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            vel_msg.angular.z = 0
            velocity_publisher.publish(vel_msg)

            elif sign == "Right":
            current_angle = 0
            t0 = rospy.Time.now().to_sec()
            while(current_angle < relative_angle):
            vel_msg.angular.z = -abs(angular_speed)
            velocity_publisher.publish(vel_msg)
            t1 = rospy.Time.now().to_sec()
            current_angle = angular_speed*(t1-t0)

            elif sign == "Left":
            current_angle = 0
            t0 = rospy.Time.now().to_sec()
            while(current_angle < relative_angle):
            vel_msg.angular.z = abs(angular_speed)
            velocity_publisher.publish(vel_msg)
            t1 = rospy.Time.now().to_sec()
            current_angle = angular_speed*(t1-t0)

            elif sign == "Roundabout":
            current_angle = 0
            t0 = rospy.Time.now().to_sec()
            while(current_angle < relative_angle*2):
            vel_msg.angular.z = -abs(angular_speed)
            velocity_publisher.publish(vel_msg)
            t1 = rospy.Time.now().to_sec()
            current_angle = angular_speed*(t1-t0)
    cv2.imshow('cv_img', image_np)
    cv2.waitKey(1)
```

```
def main(args):
        ic = image_feature()
        rospy.init_node('image_feature', anonymous=True)
        try:
        rospy.spin()
        except KeyboardInterrupt:
        print ("Shutting down ROS Image feature detector module")
        cv2.destroyAllWindows()

if __name__ == '__main__':
        main(sys.argv)
```