Sean Carda
PID: A59009786
ECE 253 – Image Processing
November 21, 2021
Homework 3

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.

By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

Problem 1:
Code:

```python
# Define a function that accepts as inputs a gratscale image and a threshold and returns an
# image containing the edges of the original.
def compute_canny_edge(im, threshold):
    # Start the smoothing process.
    print('------------------------------------')
    print('---Smoothing---')

    # Grab the dimensions of the image.
    size = im.shape

    # The first step is to smooth the image using the following gaussian kernel.
    K = (1 / 159) * np.array([[2,  4,  5,  4, 2],
                              [4,  9, 12,  9, 4],
                              [5, 12, 15, 12, 5],
                              [4,  9, 12,  9, 4],
                              [2,  4,  5,  4, 2]])



    # Compute the convolution between the kernel and the padded image.
    print('Beginning the convolution for smoothing...')
    #im_smooth = compute_convolution(im, K)
    im_smooth = ndimage.convolve(im, K)

    # Start the gradient calculation process.
    print('------------------------------------')
    print('---Gradients---')

    # Instantiate the necessary kernels to calculate the respective gradients.
    k_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])

    k_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]])

    # Calculate the gradients in the x and y directions.
    print('Computing x-gradient...')
    G_x = compute_convolution(np.float64(im_smooth), k_x)

    print('Computing y-gradient...')
    G_y = compute_convolution(np.float64(im_smooth), k_y)

    # Calculate the magnitude and direction of the gradient.
    G_mag = np.sqrt(G_x**2 + G_y**2)
    G_dir = np.degrees(np.arctan2(G_y, G_x))
```

```python

    # Start the suppression process.
    print('-------------------------------------')
    print('---NMS---')

    # First, correct the degrees
    print('Correcting the degrees matrix...')
    G_dir_round = np.zeros([size[0], size[1]])
    for r in range(0, size[0]):
        for c in range(0, size[1]):
            G_dir_round[r, c] = round_degrees(G_dir[r, c])

    # Pad the gradient magnitude matrix for NMS checks.
    G_mag_pad = np.pad(G_mag, 1, mode='symmetric')

    # For every pixel in the padded image, determine whether or not to suppress its value or
    # to keep it.
    print('Suppressing pixels...')
    #G_mag_NMS = np.zeros([size[0], size[1]])
    G_mag_NMS = np.copy(G_mag)
    for r in range(0, size[0]):
        for c in range(0, size[1]):
            # Store the current pixel in a variable.
            pixel = G_mag_pad[r + 1, c + 1]

            # Grab the gradient angle.
            grad_angle = np.radians(G_dir_round[r, c])

            # Compute the row and columns offsets given by the direction matrix.
            direction_y_offset = int(np.round(np.sin(grad_angle)))
            dirrection_x_offset =  int(np.round(np.cos(grad_angle)))

            # Calculate which pixels are neighboring the current pixel based on the offsets.
            G_north = G_mag_pad[r + 1 + direction_y_offset, c + 1 + dirrection_x_offset]
            G_south = G_mag_pad[r + 1 - direction_y_offset, c + 1 - dirrection_x_offset]

            # If the current pixel is greater than its neighbors, do not suppress it.
            if pixel > G_north and pixel > G_south:
                G_mag_NMS[r, c] = pixel
            else:
                G_mag_NMS[r, c] = 0

    # Start the thresholding process.
    print('-------------------------------------')
    print('---Thresholding---')
    G_mag_NMS = (255 / np.max(G_mag_NMS)) * G_mag_NMS
    G_threshold = np.where(G_mag_NMS > threshold, 255, 0)

    # Show some of the intermediate steps.
    print('-------------------------------------')
    print('---Displaying---')

    # Show the gradient magnitude image.
    G_mag_img = np.uint8(G_mag * (255 / np.max(G_mag)))
    cv2.imshow('Gradient Magnitude', G_mag_img)
    cv2.imwrite('gradient_magnitude.jpg', G_mag_img)
    cv2.waitKey(0)

    # Show the NMS gradient magnitude image.
    G_mag_NMS_img = np.uint8(G_mag_NMS)
    cv2.imshow('NMS Gradient Magnitude', G_mag_NMS_img)
    cv2.imwrite('nms_gradient_magnitude.jpg', G_mag_NMS_img)
    cv2.waitKey(0)

    # Return the final edged image.
    return np.uint8(G_threshold)


# Function to compute the convolution between an image and a given kernel.
def compute_convolution(im, kernel):
    # Grab the sizes of the given image and kernel.
    size_im = im.shape
    size_k = kernel.shape

    # Grab the offset generated by the kernel assuming the kernel is square with odd dimensionality.
    o = int(np.floor(size_k[0] / 2))

    # Pad the image based on the dimensions of the kernel.
    im_pad = np.pad(im, o, mode='symmetric')

    # Compute the convolution between the kernel and the padded image.
    im_smooth = np.zeros([size_im[0], size_im[1]])
    for r in range(o, size_im[0]):
        for c in range(o, size_im[1]):
            im_smooth[r, c] = np.multiply(kernel, im_pad[(r - o):(r + o + 1), (c - o):(c + o + 1)]).sum()

    # Return the convolved image.
    return im_smooth


# Function to round a given degree value to the nearest 45 degrees.
def round_degrees(degree):
    # Correct the degree value first.
```

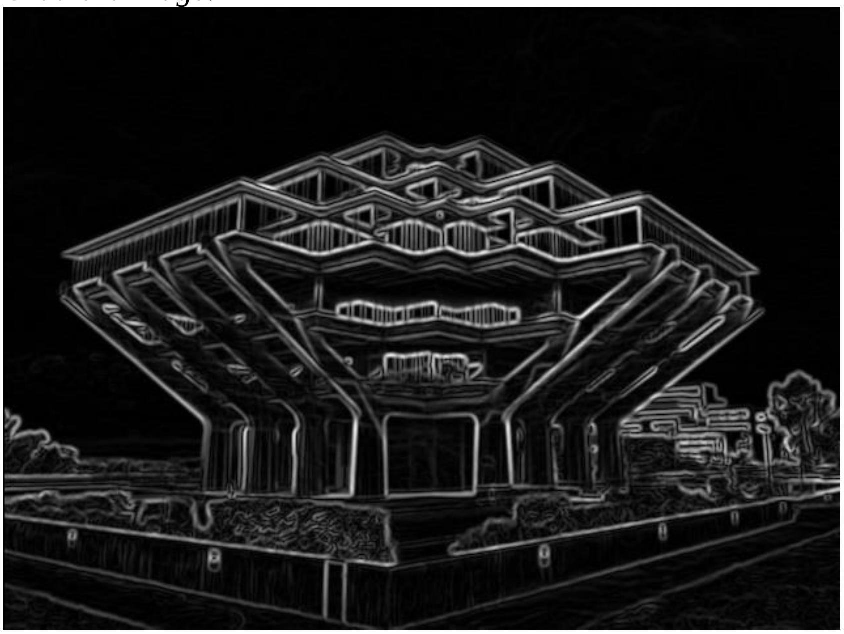
```python
    if degree < 0:
        degree = degree + 360

    # Provide a list of valid degree measurements.
    valid_degrees = [0, 45, 90, 135, 180, 225, 270, 315, 360]

    # Return the corrected degree value.
    return valid_degrees[np.argmin(abs(valid_degrees - degree))]


# Load the geisel.jpg image.
file_path = r"HW3_geisel.jpg"
A = cv2.cvtColor(cv2.imread(file_path), cv2.COLOR_BGR2GRAY)
size = A.shape

# Compute the canny edge image.
A_edges = compute_canny_edge(A, 50)

# Show the original image.
cv2.imshow('Original Image', A)
cv2.imwrite('original_geisel.jpg', A)
cv2.waitKey(0)

# Resize the edged image.
cv2.imshow('Canny Edge Image', A_edges)
cv2.imwrite('canny_edge.jpg', A_edges)
cv2.waitKey(0)
```

Output:

Original Image Color:

Original Image Grayscale:

Gradient Image:



Gradient-NMS Image:

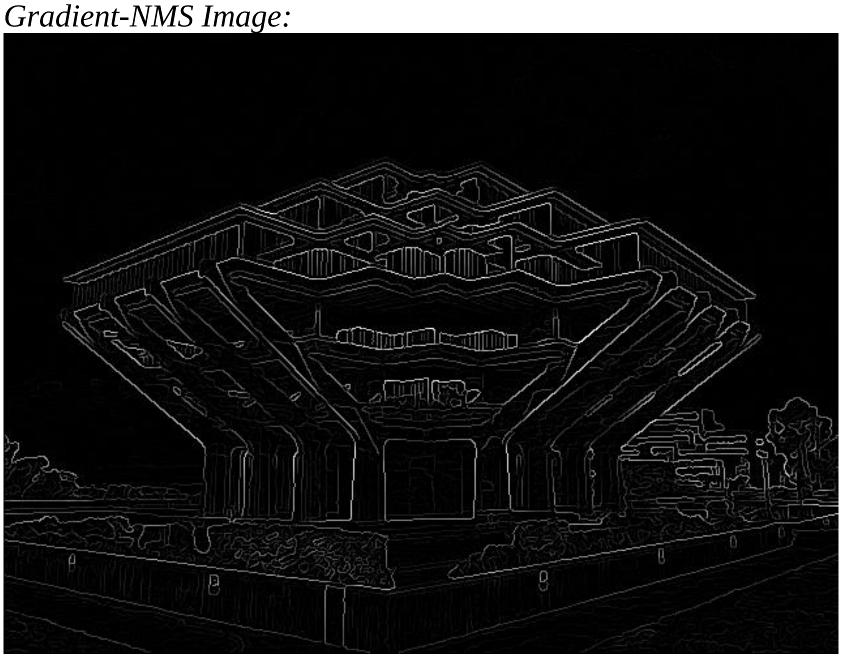
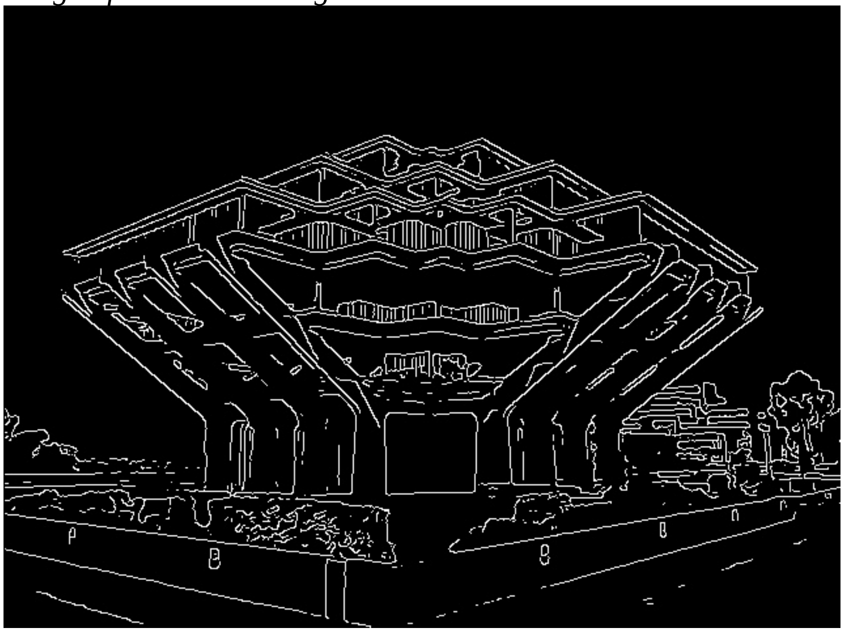


Image after Thresholding:



Here, a thresholding intensity value of **50** seemed sufficient to adequately display the obvious structure edges. It was difficult to strike a balance between removing noise produced by foliage while also preserving more obvious, thin edges such as the edge between the grass and the sidewalk. However, with this value, it is still clear that there exists some edge between the grass and concrete, and most of the noisier foliage has been removed from the image.

Problem 2:

Code:

```python
#----------------------------------------------------------------------------------------------------------------
#-----------
# PROBLEM 1
#-----------
cv2.destroyAllWindows()


# Define a function that will automatically return the log-shifted fourier transform of a
# given image.
def compute_fft(im):
    # First, pad the image to a size of 512 x 512 pixels.
    size = im.shape
    im_pad = np.zeros([512, 512])
    im_pad[0:size[0], 0:size[1]] = im

    # Now, compute the fft of the padded image.
    im_fft = np.fft.fft2(np.uint8(im_pad))

    # Shift the fft.
    im_fft_shift = np.fft.fftshift(im_fft)

    # Take the log of the magnitude of the image.
    im_fft_shift_log = np.log(np.abs(im_fft_shift))

    # Return the image.
    return [np.uint8(im_fft_shift_log), im_fft_shift]


# Function which accepts the parameters n, d0, and center corrdinates u and v to generate the
# Butterworth notch filter mask of given size.
def generate_butterworth(size, n, D0, U, U_k, V, V_k):
    # Generate an initial mask of all ones.
    mask = np.ones([size[0], size[1]])
    for u in range(0, size[0]):
        for v in range(0, size[0]):
            # Calculate the distances of the current point (u, v) to the specified centers of
            # the Butterworth filter.
            Dk_pos = np.sqrt((U[u, v] - U_k + 0.01)**2 + (V[u, v] - V_k + 0.01)**2)
            Dk_neg = np.sqrt((U[u, v] + U_k + 0.01)**2 + (V[u, v] + V_k + 0.01)**2)

            # Calculate the two terms in the product.
            term_1 = 1 / (1 + (D0 / Dk_pos)**(2*n))
            term_2 = 1 / (1 + (D0 / Dk_neg)**(2*n))

            # Calculate the product for every center specified.
            mask[u, v] = np.prod(term_1 * term_2)

    # Return the calculated mask.
    return mask


# Define the information necessary for both images.
file_path = [r"Car.tif", r"Street.png"]
u_k_dictionary = {0: [-85, -85, -85, -85],
                  1: [-166, 0]}

v_k_dictionary = {0: [-171, -85, 85, 171],
                  1: [0, 166]}

n_dictionary = {0: 3,
                1: 3}

D_0_dictionary = {0: 15,
                  1: 20}

# For both the Car and Street images, filter them with the Butterworth notch filter.
for n in range(0, 2):
    print('Computing information for ' + file_path[n] + '...')

    # Load the specified image.
    A = cv2.imread(file_path[n])
    A = A[:, :, 0]
    image_size = A.shape

    # Show the original image.
    plt.figure(1)
    plt.imshow(A, cmap='gray')
    plt.colorbar()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Original Image for ' + file_path[n])

    # Compute the fourier transform.
    print('Compute fft...')
    [A_fft_log, A_fft] = compute_fft(A)
    print('Done!')

    # Show the fft of the image.
    plt.figure(2)
    plt.imshow(15 * A_fft_log, cmap='gray')
    plt.colorbar()
    plt.xlabel('u')
    plt.ylabel('v')
    plt.title('2D Log-Magnitude DFT Image for ' + file_path[n])
```

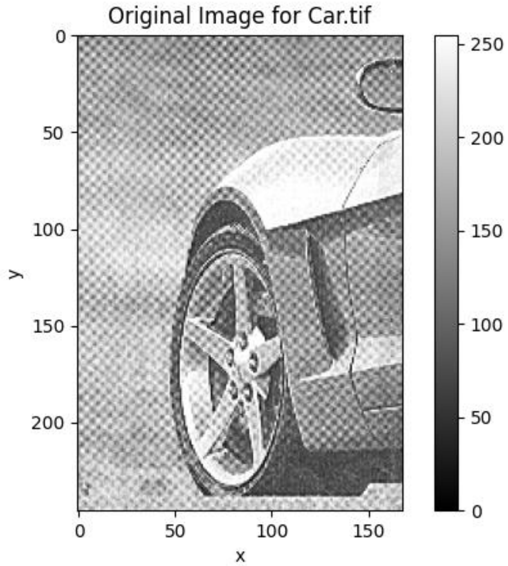
```python

    # Generate the u and v values for the Butterworth filter.
    x_axis = np.linspace(-256,255,512)
    y_axis = np.linspace(-256,255,512)
    [u,v] = np.meshgrid(x_axis,y_axis)

    # Generate the approximate (u, v) coordinates at which the impulses are located.
    u_k = u_k_dictionary[n]
    v_k = v_k_dictionary[n]

    # Generate the butterworth mask for the image.
    print('Generating mask...')
    mask = generate_butterworth([512, 512], n_dictionary[n], D_0_dictionary[n], u, u_k, v, v_k)
    print('Done!')

    # Show the generated mask.
    plt.figure(3)
    plt.imshow(np.uint8(255 * mask), cmap='gray')
    plt.colorbar()
    plt.xlabel('u')
    plt.ylabel('v')
    plt.title('Butterworth Notch Filter Mask for ' + file_path[n])

    # Apply the mask to the DFT of the image.
    print('Filtering with mask...')
    A_fft_masked = A_fft * mask

    # Compute the filtered image.
    A_filtered = np.real(np.fft.ifft2(np.fft.ifftshift(A_fft_masked)))
    A_filtered = A_filtered[0:image_size[0], 0:image_size[1]]

    # Threshold the image by 255 such that pixels over this value are reduced to 255.
    # This prevents artifacts in the uint8 version of the image.
    A_filtered = np.where(A_filtered > 255, 255, A_filtered)
    print('Done!')

    # Show the mask over the DFT.
    plt.figure(4)
    plt.imshow(np.uint8(15 * (A_fft_log * mask)), cmap='gray')
    plt.colorbar()
    plt.xlabel('u')
    plt.ylabel('v')
    plt.title('Butterworth Notch Filter Mask Overlay for ' + file_path[n])

    # Show the filtered image.
    plt.figure(5)
    plt.imshow(np.uint8(A_filtered), cmap='gray')
    plt.colorbar()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Butterworth Filtered Image for ' + file_path[n])
    plt.show()
```

Output:

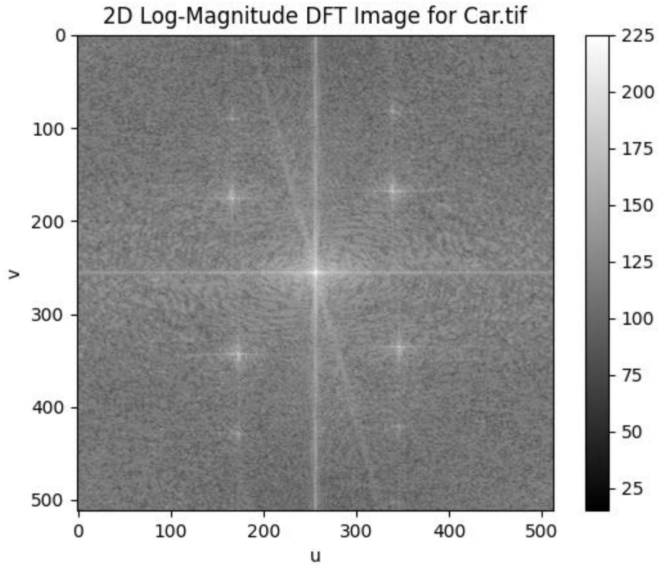
Part i:

Table of Parameters	
n	3
D_0	15
(u_1, v_1)	$(-85, -171)$
(u_2, v_2)	$(-85, -85)$
(u_3, v_3)	$(-85, 85)$
(u_4, v_4)	$(-85, 171)$

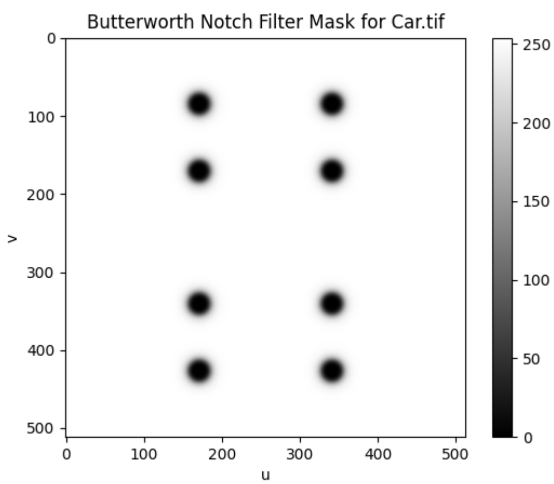
Original Image:



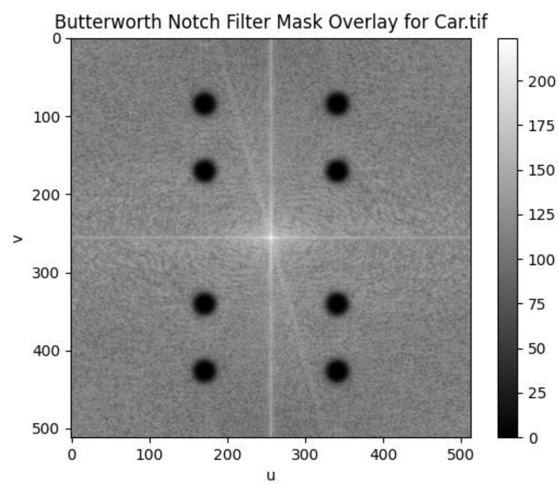
DFT Image:



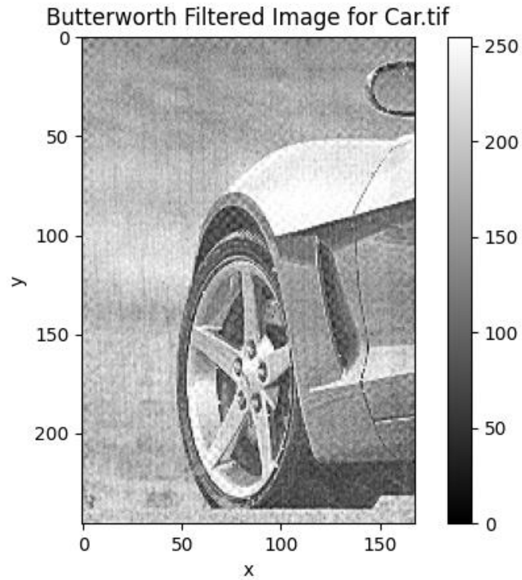
Filter Mask



Filter Mask Overlayed on DFT



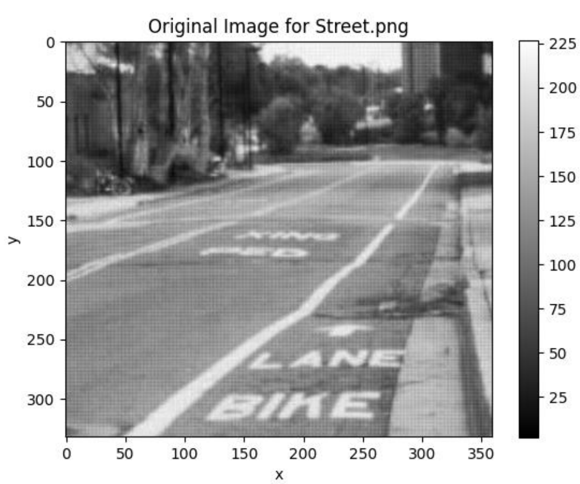
Filtered Image



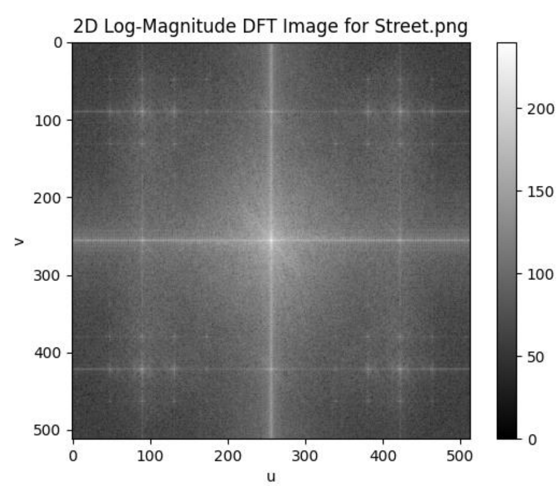
Part ii:

Table of Parameters	
n	3
D_0	20
(u_1, v_1)	$(-166, 0)$
(u_2, v_2)	$(0, 166)$

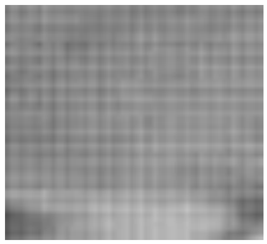
Original Image:



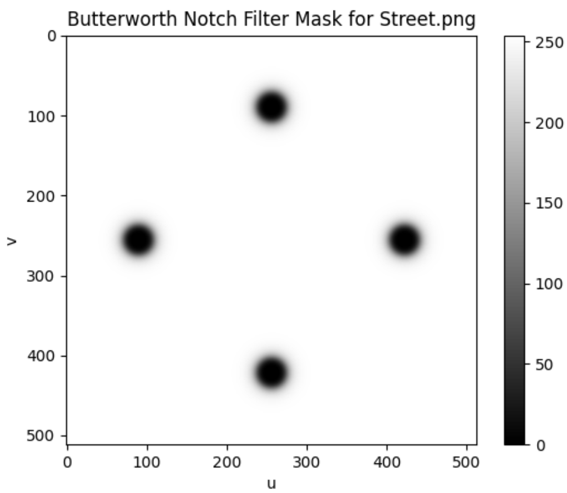
DFT Image:



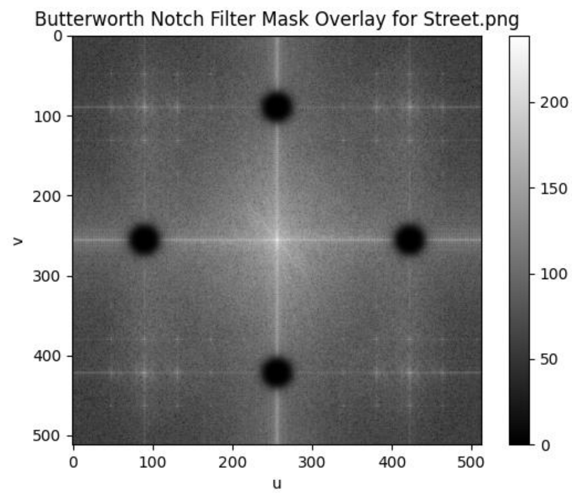
Zoomed-in:



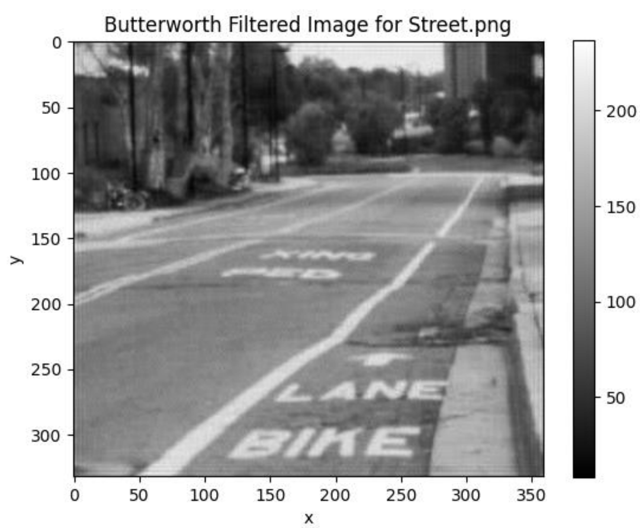
Filter Mask



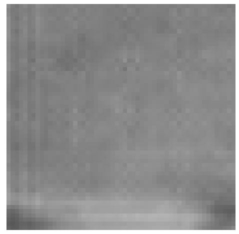
Filter Mask Overlayed on DFT



Filtered Image



Zoomed-in:



Problem 3:

Code:

```python
#----------------
# Prep work.
#----------------

# Define the transform from image data to tensor data.
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Answer question ii.
print('Question ii\n-----------')
print('Here, we are normalizing the images. However, we are not necessarily normalizing the images')
print('\ndirectly. We are converting the images in the dataset to Tensors, then we are normalizing')
print('\nthe Tensors to [-1, 1].')

# Define the batch size.
batch_size = 4

# Define the training set for the classifier.
trainset = torchvision.datasets.CIFAR10(root='./data',
                                        train=True,
                                        download=False,
                                        transform=transform)

# Answer question i.
print('\nQuestion i\n-----------')
print('There are ' + str(len(trainset)) + ' training images in the CIFAR10 dataset.')
print('Since we are using a batch size of ' + str(batch_size) + ', we have ' +
      str((int(len(trainset) / batch_size))) + ' batches for training.')


# Load the training set for the classifier.
trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          num_workers=2)

# Define the testing set for the classifier.
testset = torchvision.datasets.CIFAR10(root='./data',
                                       train=False,
                                       download=False,
                                       transform=transform)


# Print how many testing images there are.
print('There are ' + str(len(testset)) + ' testing images in the CIFAR10 dataset.')


# Load the testing set for the classifier.
testloader = torch.utils.data.DataLoader(testset,
                                         batch_size=batch_size,
                                         shuffle=True,
                                         num_workers=2)

# Define the class identifiers for the classifier.
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

#----------------
# Showing some
# traning images.
#----------------

# Define an function to show a specified image.
def imshow(img):
    # Unnormalize the image.
    img = img / 2 + 0.5

    # Convert the Tensor to a numpy array.
    npimg = img.numpy()

    # Show the image.
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Grab some random training images.
dataiter = iter(trainloader)
images, labels = dataiter.next()

# Show the random images.
imshow(torchvision.utils.make_grid(images))
print(' '.join('%5s' % classes[labels[j]] for j in range(batch_size)))

#----------------
# Define a CNN
#----------------

# Define a new class which will instantiate a CNN module.
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
```

```python
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Define a new CNN from our specified class.
net = Net()
print('CNN creation successful!')

#----------------
# Define a loss
# function and
# optimizer.
#----------------

# Define our loss criterion.
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
print('Criterion established!')

#----------------
# Train the
# network.
#----------------

print('Starting to train...')
loss_data = []
for epoch in range(2):
    # Instantiate our current loss.
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # Get the inputs where the data is a tuple of input images and labels.
        inputs, labels = data

        # Zero the parameter gradients.
        optimizer.zero_grad()

        # Forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Print the current statistics for the classification.
        running_loss += loss.item()
        if i % 2000 == 1999:
            loss_data.append(running_loss / 2000)
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training!')

#------------------
# Plot the data
# loss.
#------------------

# Question iii.
batch_count = [2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000]
batch_labels = [2000, 4000, 6000, 8000, 10000, 12000, 2000, 4000, 6000, 8000, 10000, 12000]
batch = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
epoch_labels = [0, 2]

figure, axes = plt.subplots()
axes.set_xticks(batch)
axes.set_xticklabels(batch_labels)
axes.set_xlabel('Batch Count')
axes.set_ylabel('Data Loss')

epoch_axis = axes.twiny()
epoch_axis.set_xlim(0, 4)
epoch_axis.set_xticks(epoch_labels)
epoch_axis.set_xticklabels([1, 2])
epoch_axis.set_xlabel('Epochs')

axes.plot(batch, loss_data, '-bo')
plt.title('Training Loss for the Network')
plt.show()
plt.rcParams['figure.figsize'] = [15, 8]

#----------------
# Save the
# trained model.
#----------------
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

```python

#-----------------
# Test the
# network on the
# loaded test
# data.
#-----------------

# Instantiate a iterator for the loaded test data.
dataiter = iter(testloader)
images, labels = dataiter.next()

# Print some of the test images.
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))

# Load a CNN with the trained model.
net = Net()
net.load_state_dict(torch.load(PATH))

# Generate the outputs.
outputs = net(images)
_, predicted = torch.max(outputs, 1)
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]] for j in range(4)))

# Calculate the accuracy of the network.
correct = 0
total = 0
print('Calculating the accuracy of the network...')
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct / total))

# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

print('Calculating accurate predictions per label...')
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1

print('Printing accuracy for each label...')
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                         accuracy))

# Instantiate a new loader which will only load one image at a time.
single_loader = torch.utils.data.DataLoader(trainset,
                                            batch_size=1,
                                            shuffle=True,
                                            num_workers=2)

# Define a imshow method for a single layer in the Conv2 output.
def imshow_layer(img):
    # Unnormalize the image.
    img = img / 2 + 0.5

    # Convert the Tensor to a numpy array.
    npimg = img.numpy()

    # Show the image.
    plt.imshow(npimg, cmap='gray')
    plt.show()

# Since the data is shuffled, grab the first random image.
for index, data in enumerate(single_loader, 0):
    temp = data[index]
    imshow(torchvision.utils.make_grid(temp))
    out = F.relu(net.conv1(temp))
    break

# For every layer in the Tensor, plot it as a grayscale image.
[blank, layers, r, c] = out.shape
for i in range(0, layers):
    print('Feature Map Layer ' + str(i + 1))
    imshow_layer(out[0, i, :, :].detach())
```

Output:

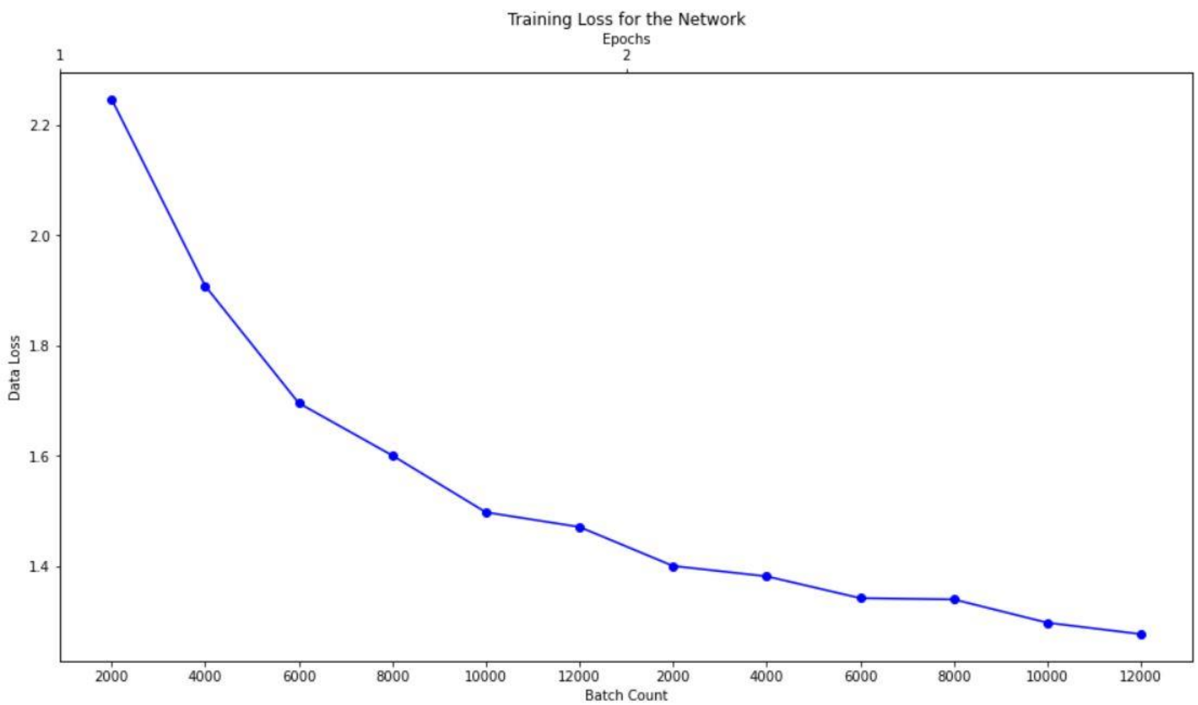
Question i:

```
Question i
-----------
There are 50000 training images in the CIFAR10 dataset.
Since we are using a batch size of 4, we have 12500 batches for training.
There are 10000 testing images in the CIFAR10 dataset.
```

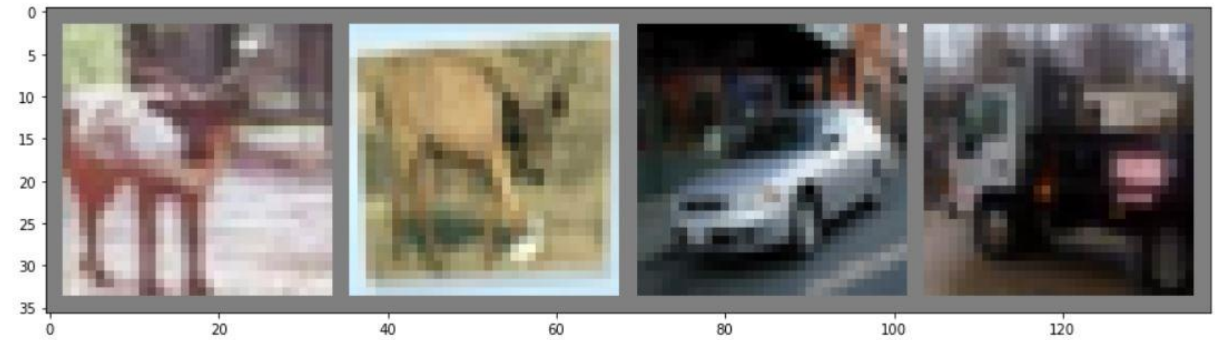
Question ii:

```
Question ii
-----------
Here, we are normalizing the images. However, we are not necessarily normalizing the images
directly. We are converting the images in the dataset to Tensors, then we are normalizing
the Tensors to [-1, 1].
```

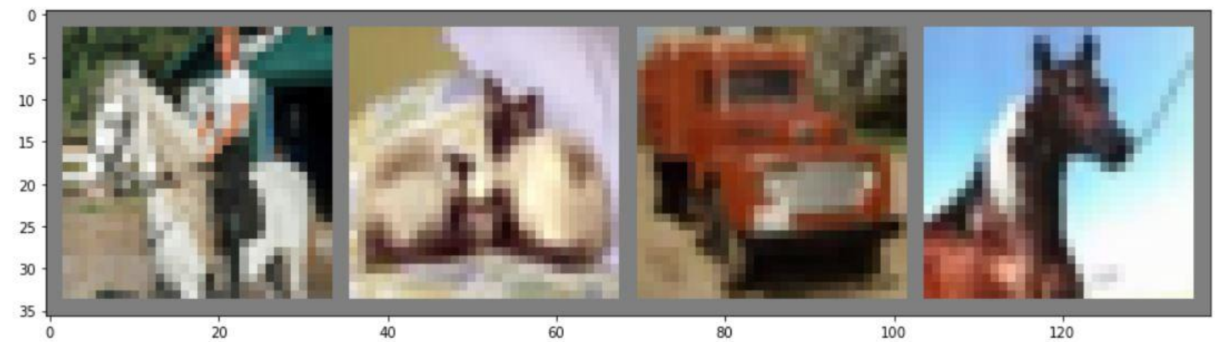
Question iii:



Question iv:




```
GroundTruth:   deer  deer   car truck
Predicted:   deer  bird   car truck
```

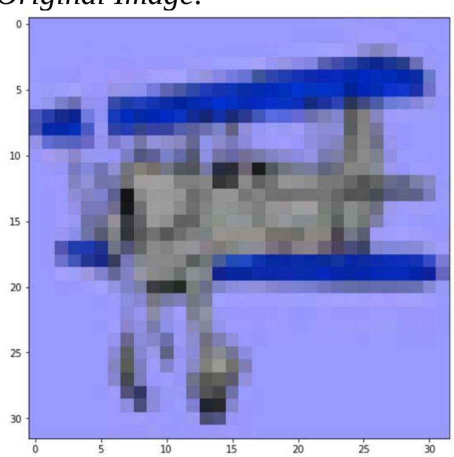



```
GroundTruth:  horse   cat truck horse
Predicted:    cat horse  frog horse
```

Here, we can see that the predicted category for the given image does not always match with the ground truth (i.e., the label assigned to the image before passed through the network). For example, in the second row, the first image was predicted to be a cat, when the image is actually of a horse. It might be likely that this discrepancy is a result of someone appearing to be riding the horse.

Question v:

Original Image:



Feature Maps Produced by the First Convolution: