Homework 2

EEE482- Computational Neuroscience

Efe Acer
21602217

Bilkent University, CS

Contents

Question 1

a) A prevalent approach to solve non-homogeneous differential equations is *the method of undetermined coefficients*. The approach is based on the observation that the general solution of a non-homogeneous differential equation is the sum of the homogeneous solution and the particular solution.

We are given the non-homogeneous differential equation:

$$\tau_m \frac{dv}{dt} = -v + RI(t) \tag{1}$$

that models the sub-threshold membrane-potential dynamics for the leaky integrate-and-fire neuron.

We can solve for v in (1) using *the method of undetermined coefficients*. We assume a DC input current $I(t) = I_0$ and an initial voltage value of $v(0^-) = 0$:

$$\tau_m \frac{dv}{dt} + v = RI_0 \tag{2}$$

As our first step we obtain the homogeneous solution by solving:

$$\tau_m v^{'} + v = 0 \tag{3}$$

The characteristic polynomial of (3) is:

$$\tau_m r + 1 = 0$$

which has the root: $r = -1/\tau_m$.

Hence, the homogeneous solution is:

$$v_h(t) = Ce^{-t/\tau_m}, \text{ where C is an arbitrary constant} \tag{4}$$

We proceed by finding the particular solution. The particular solution is expected to be a constant multiple of the right hand side of (2). Thus, we assume that the particular solution is $v_p(t) = ARI_0$, where A is the constant multiple. We plug $v_p(t)$ in (2) to find the value of A:

$$\tau_m \frac{d(ARI_0)}{dt} + ARI_0 = RI_0$$
$$\tau_m.0 + ARI_0 = RI_0 \implies A = 1$$

Using the value of A, we can write the particular solution as:

$$v_p(t) = RI_0 \tag{5}$$

Since we have the homogeneous and the particular solutions in our hands, we can write the general solution of the non-homogeneous equation (2) as their sum:

$$v(t) = v_h(t) + v_p(t) = Ce^{-t/\tau_m} + RI_0 \tag{6}$$

What remains is to take the initial condition $v(0^-) = 0$ into consideration and solve for C:

$$t = 0^- \implies v(0^-) = Ce^{-0^-/\tau_m} + RI_0 = 0$$
$$C + RI_0 = 0 \implies C = -RI_0$$

We plug the value of C into (6) and obtain:

$$v(t) = -RI_0\ e^{-t/\tau_m} + RI_0 = RI_0\ (1 - e^{-t/\tau_m}) \tag{7}$$

(7) is the general solution, given the initial condition.

b) *Euler's method* is a particularly useful method when one needs to find an approximate solution to a differential equation [1]. Recall that the discrete counterpart of a differential equation is a difference equation. *Euler's method* simply transforms the differential equation to a difference equation using an appropriate temporal step size.

Let h denote the temporal step size, then using *forward difference method* we can approximate $v^{'}(t)$ as:

$$v^{'}(t) \approx \frac{v(t+h) - v(t)}{h} \tag{8}$$

Using this approximation, (2) can be rewritten as:

$$\tau_m\ \frac{v(t+h) - v(t)}{h} + v(t) = RI_0$$
$$\tau_m v(t+h) + (h - \tau_m)v(t) = hRI_0$$

If we denote $v(t+h) = v_{i+1}$ and $v(t) = v_i$, then we transformed the differential equation (2) to the difference equation:

$$v_{i+1} = \frac{\tau_m - h}{\tau_m}\ v_i + \frac{hRI_0}{\tau_m}, \text{ for } i = 0, 1, 2, ... \tag{9}$$

Note that we can translate the initial condition of (2) to (9) as:

$$v(0^-) = 0 \rightarrow v_0 = 0$$

(9) can be easily implemented as an iterative `Python` script as follows:

```
# Constants
TAU_M = 10e-3 # 10 milli seconds
R = 1e3 # 1 kilo ohm
I_0 = 2e-3 # 2 milli ampere
TIME_INTERVAL = (0, 100e-3) # 0 to 100 milli seconds
NUM_STEPS = 10000 # number of bins in the time interval
H = (TIME_INTERVAL[1] - TIME_INTERVAL[0]) / NUM_STEPS # step_size

v_numerical = np.zeros(NUM_STEPS) # in volts
t = np.zeros(NUM_STEPS) # in seconds

# Computing values in the first order difference equation
for i in range(0, NUM_STEPS - 1):
    v_numerical[i + 1] = ((TAU_M - H) * v_numerical[i] + H * R * I_0) / TAU_M
    t[i + 1] = (i + 1) * H
```

The script basically iterates over an array, which has a first element of 0, and repeatedly substitutes the most recent $v(t)$ value for v_i to calculate the next value, v_{i+1}. The calculation is performed according to the first order difference equation (9) and the constants are set as they are given in the problem statement. The step size h is taken as 10 μs. Note that `Numpy` should be imported as `np` for the script to run.

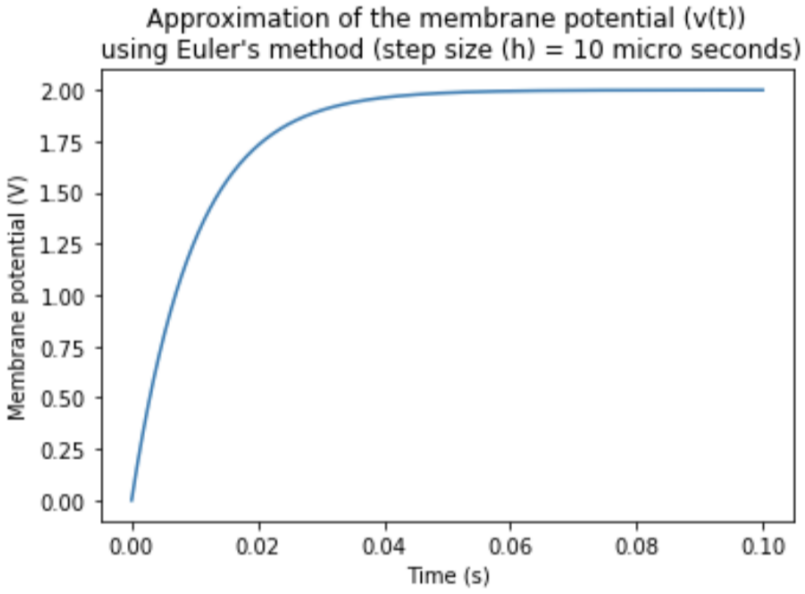

Figure 1: Approximation of membrane potential using *Euler's method*

Using `plot` function from `matplotlib`'s `pyplot` package, the discretized values of $v(t)$, v_i's, stored in the `v_numerical` array are plotted against the respective time values stored in the `t` array. The resulting plot is given at the end of the previous page.

The analytical solution in (7) is also implemented as a `Python` function and the values it generates for the same time values, `t` array, used in *Euler's method* is plotted. The function and the plot are given below:

```
1  def analytical_solution(t):
2      """
3      Given a time value, computes and returns the value of
4      membrane potential calculated using the analylitical
5      solution of the model involving the differential equation.
6      (Uses globally defined constants)
7      Args:
8          t: The specified time value
9      Returns:
10         v(t): The membrane potential calculated according
11             to the analytical solution.
12     """
13     return R * I_0 * (1 - np.exp(-t / TAU_M))
```

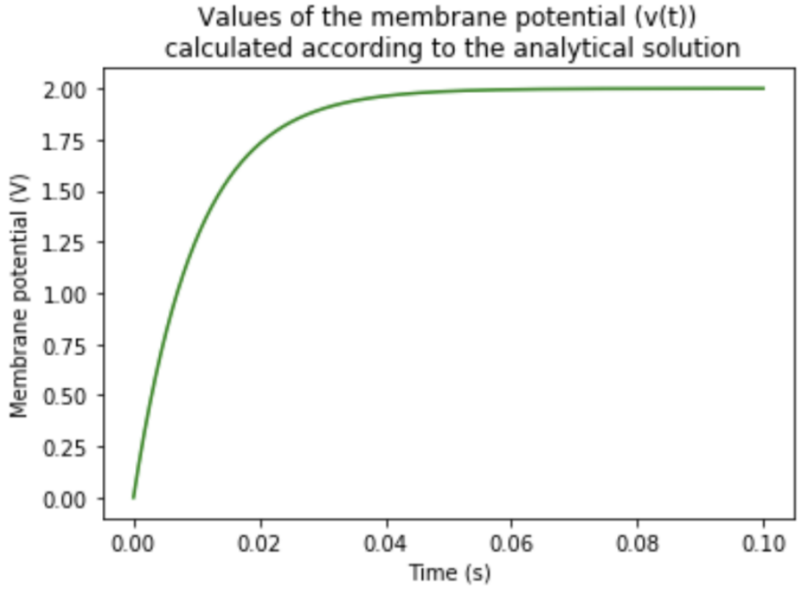

Figure 2: Values of membrane potential calculated using the analytical solution

The plots in Figure 1 and Figure 2 look indistinguishable. Hence, mean squared error (MSE) is defined as the error metric to measure the approximation's precision:

$$\text{MSE} \coloneqq \frac{1}{|T|} \sum_{t \in T} (v_{analyical}(t) - v_{numerical}(t))^2$$

In the definition above; the set T gathers the time steps used in the calculations, $v_{analyical}(t)$ denotes a voltage value obtained using the analytical solution and $v_{numerical}(t)$ denotes a voltage value obtained using the *Euler's method*.

MSE can be calculated and printed using the lines below:

```
MSE = np.sum((v_analytical - v_numerical) ** 2) / NUM_STEPS
print('Mean Squared Error (MSE) of the numerical approximation: %.2g' % MSE)
```

The code prints:

```
Mean Squared Error (MSE) of the numerical approximation: 2.5e-08
```

MSE gives the second moment of the error, which relates to the bias and the variance of the approximation. A MSE value of 2.5×10^{-8} V^2 is extremely small compared to the scale of the voltage values which range from 0 V to 2 V. Thus, we have a sufficiently good numerical approximation.

c) In order to incorporate the phenomenon of spike emission into our model, we define a threshold voltage level θ. Then, we define the spike emission time t_s where the voltage level equals the threshold, $v(t_s) = \theta$. Lastly, we reset the voltage to a certain value v_{reset} right after the spike emission time. The procedure can be mathematically summarized as follows:

$$\lim_{\Delta t \to 0} v(t_s + \Delta t) = v_{reset}, \text{ where } v(t_s) = \theta \tag{10}$$

This procedure can be easily implemented in `Python` on top of the code that uses *Euler's method* to solve (9). We should only add an `if` statement that checks for the instances where the voltage level exceeds θ, and reset the voltage to v_{reset} at those instances. Such an implementation is given below where the threshold voltage level equals 1 V and the reset voltage level is 0 V:

```
# Constants
THETA = 1 # a threshold value of 1 V
V_RESET = 0 # the reset voltage value

v_threshold_numerical = np.zeros(NUM_STEPS) # in volts

```

```
# Computing values in the first order difference equation
for i in range(0, NUM_STEPS - 1):
    voltage = ((TAU_M - H) * v_threshold_numerical[i] + H * R * I_0) / TAU_M
    v_threshold_numerical[i + 1] = V_RESET if voltage >= THETA else voltage
```

The values of $v(t)$ where spike emission is incorporated into the model is stored in the array named `v_threshold_numerical` after we run the code above. We plot the values in the array against the time interval and expect to see regular spikes:

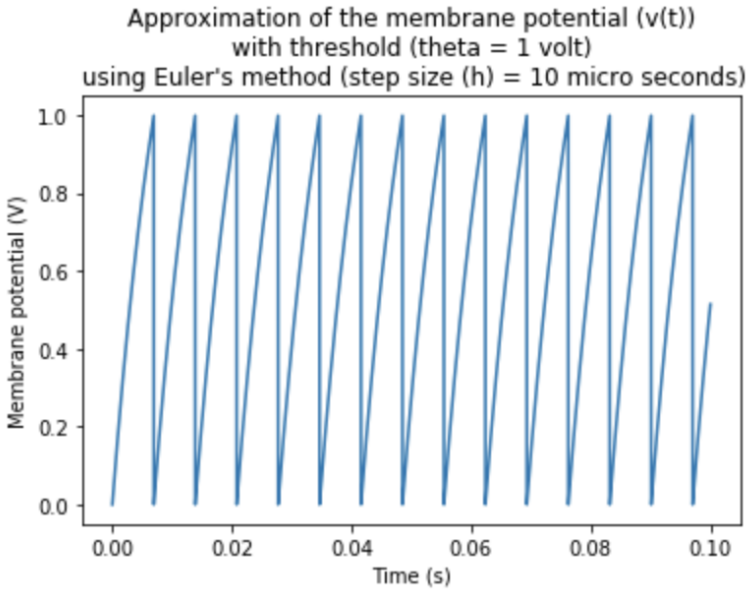

Figure 3: Approximation of membrane potential with spike emission using *Euler's method*

We observe 14 spikes that are occur with a uniform interspike interval. The plot is still far away from representing the behavior of an actual neuron but it is a well representation of an *in vitro* experiment where a non-noisy DC input current is applied to the experiment's subject.

It is again convenient to construct the plot of the same model using the analytical solution. This can be done by finding the spike emission times that make the analytical solution compute a value above the threshold, and shift these values together with the values that follow them to the initial time point. The analytical solution is then computed again at the shifted time values. The `Python` code implementing this approach follows:

```
v_threshold_analytical = np.zeros(NUM_STEPS) # in volts
t_threshold = np.array(t)

```

```
4 for i in range(NUM_STEPS):
5     voltage = analytical_solution(t_threshold[i])
6     if voltage >= THETA:
7         t_threshold[i:] -= t_threshold[i]
8         voltage = analytical_solution(t_threshold[i])
9     v_threshold_analytical[i] = voltage
```

The `if` statement in the code above checks for the spike emission times where the voltage exceeds θ. When such a time value is encountered, that value and the proceeding values are shifted back to the starting time point. The analytical solution is applied again to the shifted time values. The results stored in `v_threshold_analytical` is plotted and the following figure is obtained:

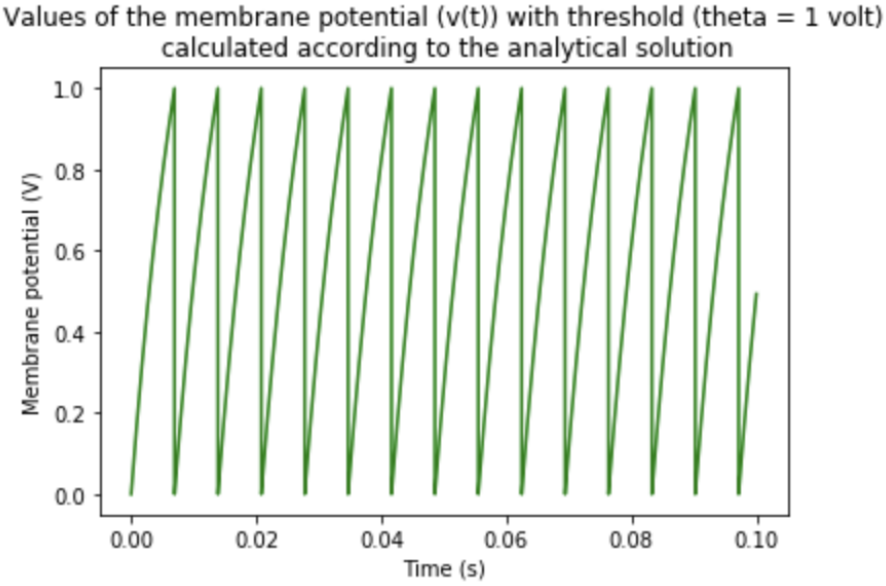

Figure 4: Values of membrane potential with spike emission calculated using the analytical solution

The plots Figure 3 and Figure 4 are again very similar. MSE is used once more as the error metric to measure how well the approximation is:

```
1 MSE = np.sum((v_threshold_analytical - v_threshold_numerical) ** 2) / NUM_STEPS
2 print('Mean Squared Error (MSE) of the numerical approximation '
3       '(threshold case): %.2g' % MSE)
```

The code prints a MSE value of 0.01 which is worse than the MSE value obtained in part **b** but still tolerable, since it is about 100 times smaller than a membrane potential value.

d) The firing rate refers to the frequency of the spike occurrences. Hence, it is the inverse of the interspike interval:

$$\text{firing rate} = \frac{1}{\text{interspike interval}} \tag{11}$$

Notice again that the interspike intervals in our model are uniform, so it suffices to calculate the very first interspike interval and compute the firing rate as the inverse of it. The following script computes the firing rates for DC input currents ranging from 2 mA to 10 mA:

```
I_VALUES = np.arange(2e-3, 10e-3 + 1e-5, 1e-5) # current values in milli amperes

interspike_intervals = np.zeros(np.size(I_VALUES))

for i, I in enumerate(I_VALUES):
    v = np.zeros(NUM_STEPS)
    for j in range(0, NUM_STEPS - 1):
        v[j + 1] = ((TAU_M - H) * v[j] + H * R * I) / TAU_M
        if v[j + 1] >= THETA:
            interspike_intervals[i] = (j + 1) * H
            break

firing_rates = 1 / interspike_intervals
```

The resulting firing rates are plotted against the input current values:

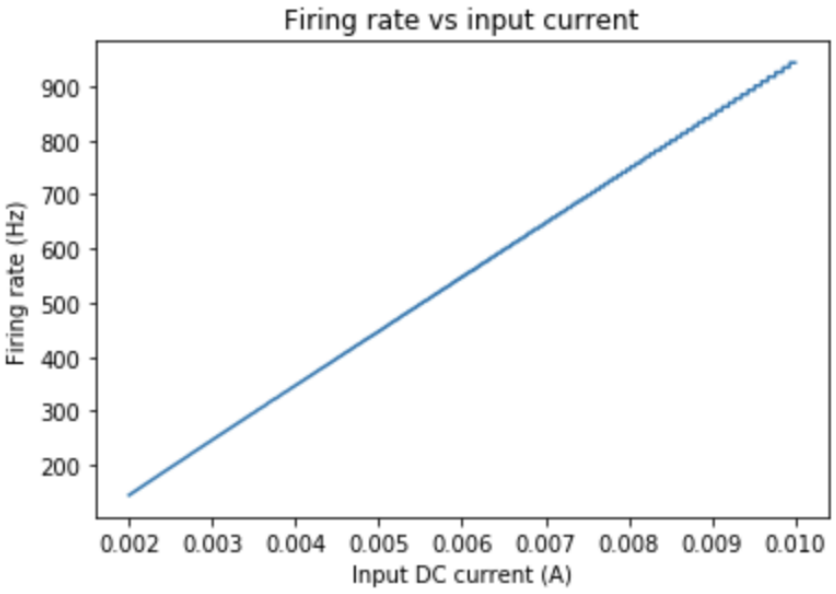

Figure 5: Firing rate vs input current

Looking at Figure 5, we can say that the firing rate and the DC input exhibits a liner relationship where the firing rate increases with the input current. This result is an expected one since the input current appears as a scalar multiplier in (7), meaning that the increase in the input current translates to a linear increase in the membrane potential. The result can be understood better if we take a look at the derivative of (7):

$$v^{'}(t) = \frac{RI_0}{\tau_m}\, e^{-t/\tau_m} \tag{12}$$

(12) implies that the rate of increase in the membrane potential is linearly proportional with the input current I_0. An input current, say I_1, that is 10 times greater than I_0 will make the voltage level increase 10 times faster. The 10 fold increase in the voltage level will make the interspike intervals 10 times smaller because of the fact that the voltage will reach to the threshold value 10 times quicker. Hence, it is obvious that the firing rate will be 10 times greater. Dayann *et al.* also agrees with the result [2].

Note that the interspike intervals are computed using the numerical approximation of (2). This is because the numerical approach is usually more convenient and there is no significant difference between the numerical solution and the analytical one, as it is shown in the previous parts.

e) So far, we assumed a perfectly uniform DC input current. However, this is only possible for controlled *in vitro* experiments. In the *in vivo* experiments, meaning the experiments that are conducted in a living organism, noisy input is inevitable. In order to make our model more realistic, we link an additional noise value to the DC current. The noise, denoted by $n(t)$, is sampled from a Gaussian with 0 mA mean and 4 mA standard deviation:

$$n(t) = N(\mu = 0, \sigma^2 = 16)\ \text{mA}$$

After this noise is added on top of the 2 mA DC input, the input current becomes:

$$I_0 = 2 + n(t)mA = 2 + N(\mu = 0, \sigma^2 = 16)\ \text{mA}$$

This new value of I_0 is incorporated into the numerical solution code in part **c** to approximate the membrane potential where the input current includes some noise. The temporal step size is increased from 10 μs to 50μs since small values of step size makes it harder to observe the effect of the noise. The resulting voltage values are again plotted against time. The code and the plot are given in the beginning of the next page:

```python
# Constants
GAUSSIAN_MEAN = 0
GAUSSIAN_STD = 4e-3 # 4 milli amperes
NUM_STEPS = 2000
H = (TIME_INTERVAL[1] - TIME_INTERVAL[0]) / NUM_STEPS

v_noisy = np.zeros(NUM_STEPS) # in volts
t_noisy = np.zeros(NUM_STEPS)

# Computing values in the first order difference equation
for i in range(0, NUM_STEPS - 1):
    noise = GAUSSIAN_STD * np.random.randn() + GAUSSIAN_MEAN
    voltage = ((TAU_M - H) * v_noisy[i] + H * R * (I_0 + noise)) / TAU_M
    v_noisy[i + 1] = V_RESET if voltage >= THETA else voltage
    t_noisy[i + 1] = (i + 1) * H
```

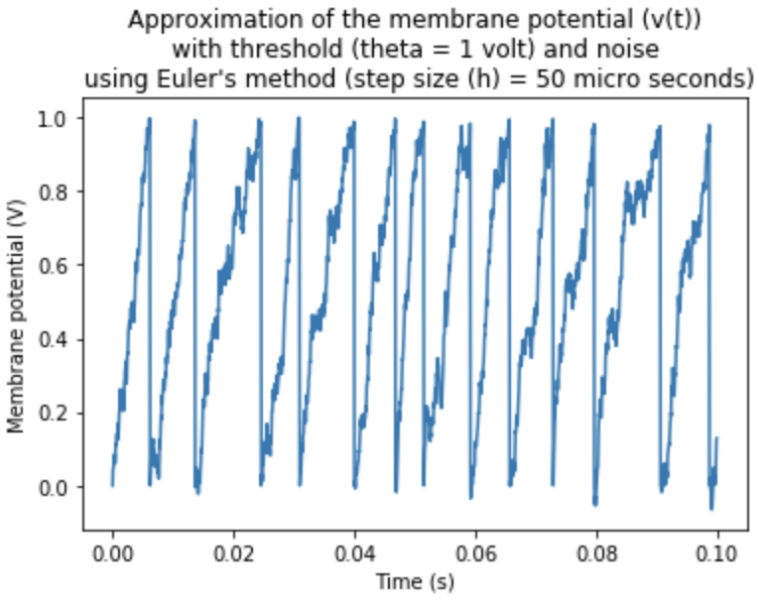

Figure 6: Approximation of membrane potential with spike emission and noise using *Euler's method*

The noise caused irregularities in the membrane potential. However, it did not change the overall behavior of the neuron. There are almost 14 spikes although there are slight variances in the interspike intervals.

In the case of part `d`, interspike intervals were uniform. Howbeit, as it can be seen from Figure 6 the interspike intervals vary when we introduce noise. As a result of this, we should calculate the firing rate as the inverse of the average interspike intervals:

$$\text{firing rate} = (\overline{\text{interspike interval}})^{-1} \tag{13}$$

The code that computes the firing rates for noisy versions of DC current inputs used in part **d** is the following:

```
last_pair = (0, 0)
interspike_intervals = np.zeros(np.size(I_VALUES))

for i, I in enumerate(I_VALUES):
    intervals = [] # holds the individual interspike intervals
    for j in range(0, NUM_STEPS - 1):
        noise = GAUSSIAN_STD * np.random.rand() + GAUSSIAN_MEAN
        v_noisy[j + 1] = ((TAU_M - H) * v_noisy[j] + H * R * (I + noise)) / TAU_M
        if v_noisy[j + 1] >= THETA:
            v_noisy[j + 1] = V_RESET
            interval = (j + 1) * H
            # subtract the end point of the previous time interval if needed
            if last_pair[0] == i:
                interval -= last_pair[1]
            last_pair = (i, ((j + 1) * H))
            intervals.append(interval)
    interspike_intervals[i] = np.mean(interval) # average interspike interval

firing_rates_noisy = 1 / interspike_intervals
```

The code computes the average interspike interval for each noisy current input and then takes the inverse of these values to find the respective firing rates. While computing the average interspike interval for a current input, a variable called `last_pair` references to the end point of the previous interval to be able to derive the length of the current interval.

The resulting firing rates are stored in the `firing_rates_noisy` array. The values in this array are scatter plotted against the input current values. Then the best fit line of the scatter plot is obtained using the `polyfit` and `poly1d` functions of `Numpy`. The code that generates the plot and the plot itself follow:

```
plt.figure(figure_num)
figure_num += 1
plt.scatter(I_VALUES, firing_rates_noisy,
            s=2, color='m', label='average firing rates')
```

```
plt.plot(I_VALUES, np.poly1d(np.polyfit(I_VALUES, firing_rates_noisy, 1))
        (I_VALUES), color='y', linewidth=2, label='best fit line')
plt.legend()
plt.title('Firing rate vs noisy input current')
plt.xlabel('Input noisy DC current (A)')
plt.ylabel('Firing rate (Hz)')
x1, x2, y1, y2 = plt.axis()
plt.axis((2e-3, 10e-3, y1, y2))
plt.show(block=False)
```

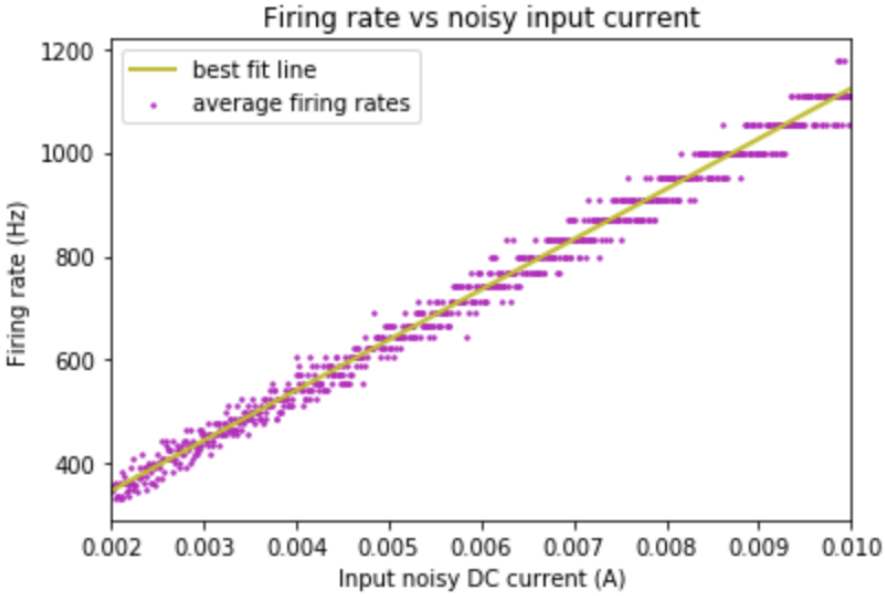

Figure 7: Firing rate vs noisy input current

Figure 7 shows that a noisy input current results in fluctuated firing rates, still the best fit line indicates that the linearly proportional relation between the input current and the firing rate is preserved. It can be observed that the fluctuations of the firing rates increase as the DC input increases. This is an expected behavior. Recall from (12) that the increase in the input current directly translates to an increase in the firing rate. Since we add a stationary variance to the input current, we expect this variance to deviate the firing rate. Clearly, the variance of larger input currents will deviate the firing rates more than the variance of smaller input currents.

Question 2

a) The *spike triggered average* (STA) is a useful tool to characterize a neuron's response behavior. Given a time-varying stimulus and corresponding spike rates, STA approximates the average stimulus before a spike. To compute the STA, we set a fixed length time window, slide it through the time-varying stimulus and calculate a weighted average of the stimuli covered by the window. The weights in the averaging procedure are the corresponding spike rates. In other words, we take a fixed length stimulus before each time we observe a positive spike rate and multiply that stimulus with the respective spike rate, at the end we compute the average of these stimuli. The resulting STA helps us to infer what the neuron is selective for.

Kara *et al.* give the spike trains from lateral geniculate nucleus (LGN) neurons in cat. The LGN neurons have a layered structure and they are responsible for projecting the input they receive from the eyes to the primary visual cortex (V1) [3]. LGN neurons have center-surround receptive fields similar to those of the retinal cells. An LGN neuron may be sensitive to orientation, direction or luminous intensity.

The spike trains given by Kara *et al.* are in `.mat` format. To be able to use the spike trains in the `Python` environment we use the `loadmat` function from `SciPy`'s `io` package. We import the `Numpy` and `Matplotlib` libraries for our computations. We load the spike train data as follows:

```
1 data = loadmat('c2p3.mat')
2 counts = data['counts'].flatten()
3 print('Dimension of counts:', np.shape(counts))
4 stim = data['stim']
5 print('Dimension of stim:', np.shape(stim), '\n')
```

The code prints:

```
Dimension of counts: (32767,)
Dimension of stim: (16, 16, 32767)
```

As it can be seen from the shape of the `stim` array, we have 32767 stimuli recorded at different time steps where each stimulus is a 16x16 grayscale image. The corresponding spike counts are stored in the `counts` array. A time step is 15.6 ms. Hence, `stim(x, y, t)` gives the stimulus presented at the coordinate `(x, y)` at time (15.6 t) ms. Similarly, `counts(t)` gives the number of spikes elicited by the LGN cell at time (15.6 t) ms.

We calculate the STA images for each of the 10 time steps before each spike to infer what the neuron is selective for. A `Python` function is implemented to calculate these STA images using the procedure explained in the previous page:

```
def STA(counts, stim, num_steps):
    """
    Given spike and stimulus data, performs Spike Triggered
    Averaging (STA). STA is performed by computing a weighted
    averaging "num_steps" individual intervals of the stimulus.
    Returns the averages after the computation.
    Args:
        counts: The spike counts
        stim: The stimulus data
        num_steps: Number of individual intervals, in which STA
            will be computed
    Returns:
        averages: The resulting averages.
    """
    averages = np.zeros((np.shape(stim)[0], np.shape(stim)[1], num_steps))
    for i in range(np.size(counts)):
        for j in range(num_steps):
            if i > j:
                averages[:,:, j] += stim[:,:, i - 1 - j] * counts[i]
    averages /= np.sum(counts)
    return averages
```

The STA images are calculated and stored in an array as follows:

```
STAs = STA(counts, stim, NUM_STEPS)
```

In the code above, `NUM_STEPS` has a value of 10, since we are interested in the 10 time steps before each spike. We plot the STA values stored in the `STAs` array using the following code:

```
# Find the range of pixels that cover all STAs
min_pixel = np.min(STAs)
max_pixel = np.max(STAs)
figure_num = 1
for i in range(NUM_STEPS):
    plt.figure(figure_num)
    figure_num += 1
    plt.imshow(STAs[:,:, i], cmap='gray', vmin=min_pixel, vmax=max_pixel)
    step_or_steps = 'steps' if i != 0 else 'step'
    plt.title('STA %d %s before a spike' % ((i + 1), step_or_steps))
    plt.show(block=False)
```

Notice that the minimum and maximum pixel values of 10 STA images must be given as

arguments to the `imshow` function. This is necessary since the function draws each image individually resulting in a different pixel range for each STA image.

The output of the code is given below:

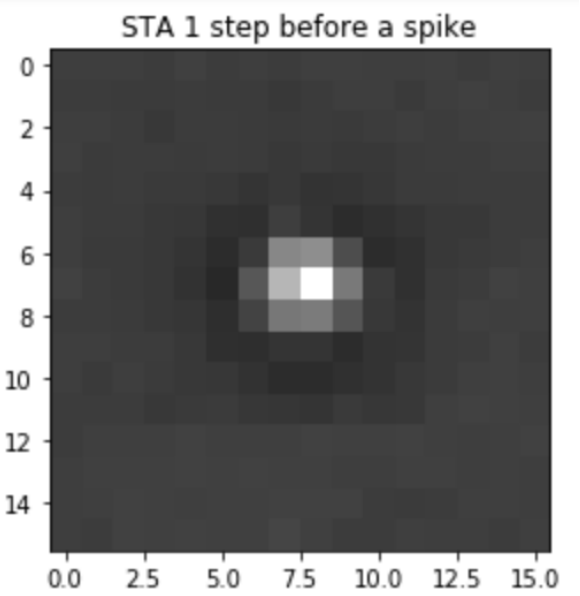

Figure 8: STA image 1 time step before a spike

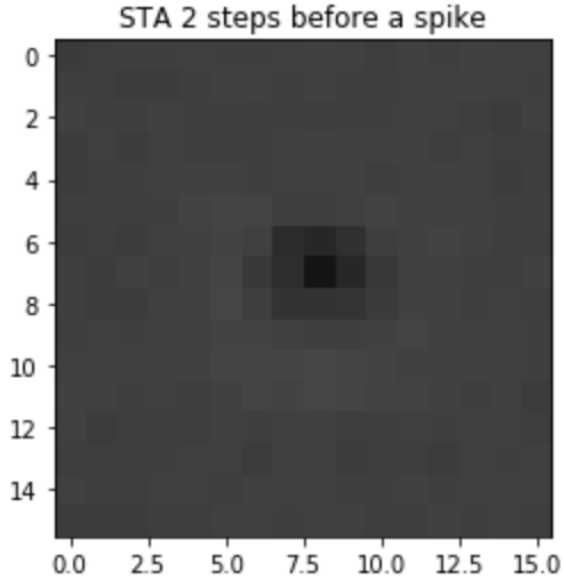

Figure 9: STA image 2 time steps before a spike

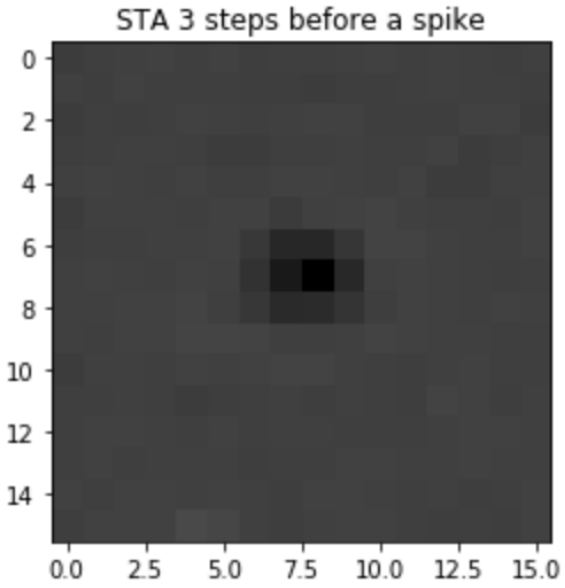

Figure 10: STA image 3 time steps before a spike

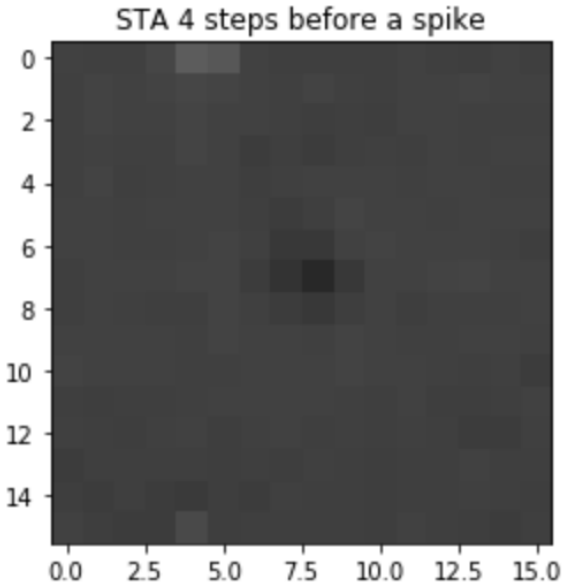

Figure 11: STA image 4 time steps before a spike

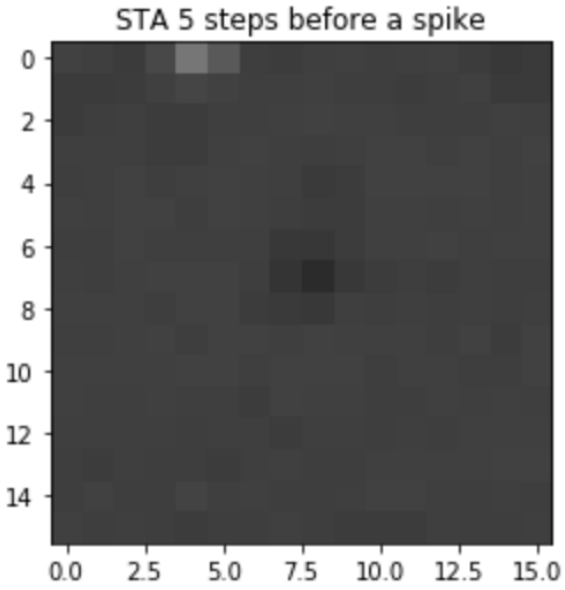

Figure 12: STA image 5 time steps before a spike

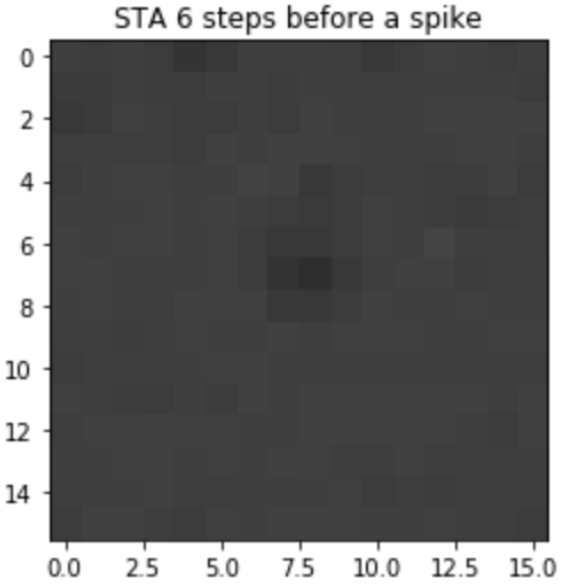

Figure 13: STA image 6 time steps before a spike

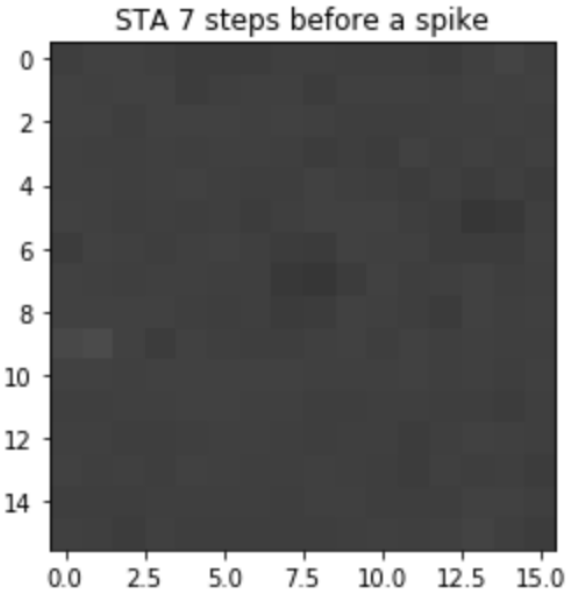

Figure 14: STA image 7 time steps before a spike

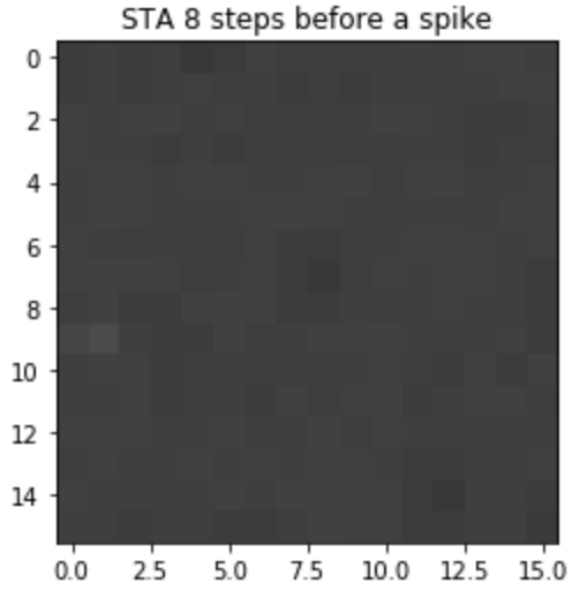

Figure 15: STA image 8 time steps before a spike

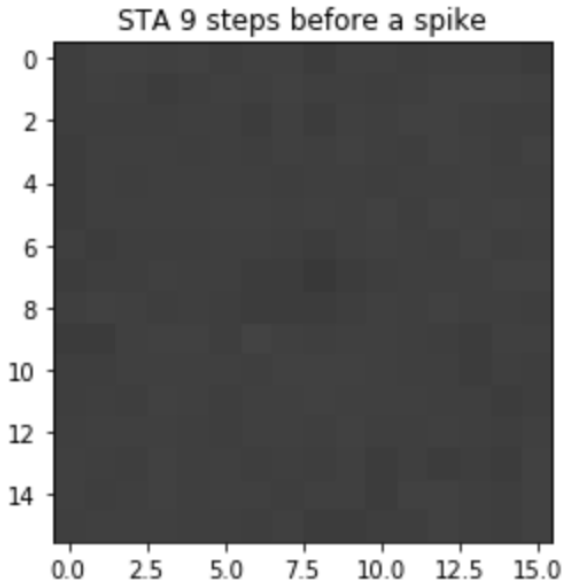

Figure 16: STA image 9 time steps before a spike

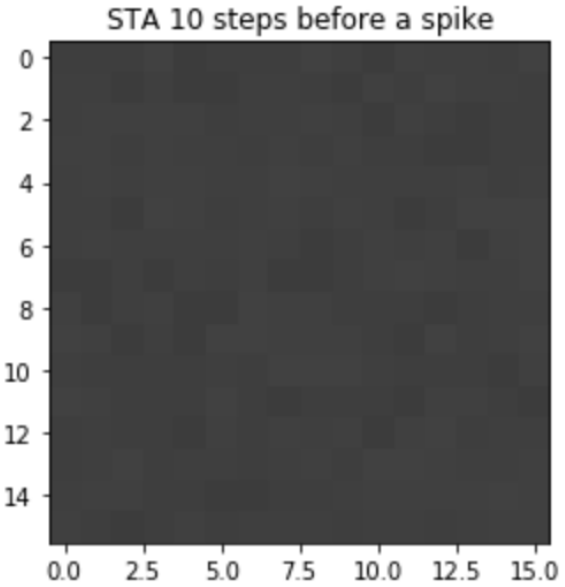

Figure 17: STA image 10 time steps before a spike

It is obvious from the output STA images that the LGN neuron is not sensitive to orientation or direction. It seems like it responds to a sudden increase in the luminous intensity. Such a stimulus may be a flashlight. We can also see that the central pixels of the STA images become darker until 1 time step before a spike, this implies that the LGN neuron's response requires a decrease in the luminous intensity before the sudden increase. In conclusion, the neuron is selective for a sudden central increase in the luminous intensity preceded by a central decrease.

b) In order to see how the STA images changes in time, we choose a spatial direction and sum the STA images over that spatial direction. We perform this procedure separately for the direction of rows and the direction of columns. The `sum` function of `Numpy` allows us to specify the direction that the values of a `Numpy` array will be summed over. This direction is given to the function with the `axis` parameter where an `axis` value of 0 sums a matrix across the rows and a value of 1 sums a matrix across the columns. The code that performs the required operation is the following:

```
1 row_summed_avgs = np.sum(STAs, axis=0) # sum across rows
2 col_summed_avgs = np.sum(STAs, axis=1) # sum across columns
```

The results `row_summed_avgs` and `col_summed_avgs` are 16x10 matrices, they are displayed using the `imshow` function:

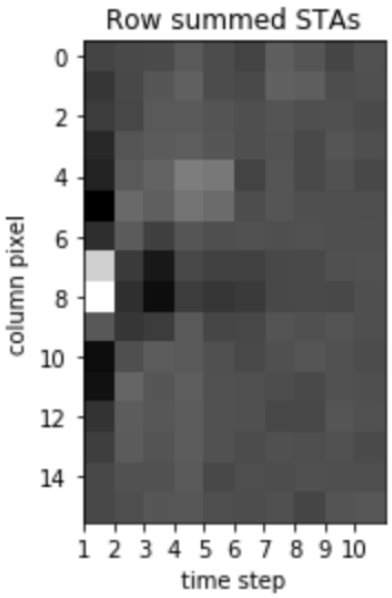

Figure 18: STA images summed across the rows

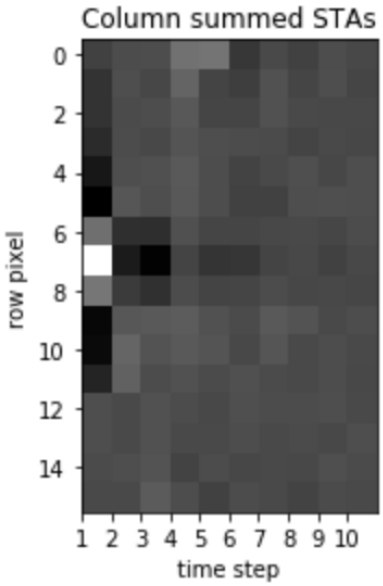

Figure 19: STA images summed across the columns

The images shown in Figure 18 and Figure 19 are almost symmetric. Hence, we once again see that the LGN neuron is not selective for the spatial direction.

It can be observed that the varying pixels are the ones that are near the center, meaning that the change in the pixel values occur mainly between the 4th and the 10th pixels (pixel numbers range between 0 and 15). This observation implies that the LGN neuron is selective for the changes in the values of the central pixels.

We can say that the matrices shown in Figure18 and Figure19 are not space-time separable. If the matrices were space-time separable, we would have observed a time independent figure where all columns corresponding to the time steps are identical. Conversely, we see that central pixels of the columns corresponding to the time steps greater than 7 are all gray; the ones corresponding to the time steps between 3 and 7 are a little darker and the ones corresponding to the time steps less than 3 are brighter.

c) *Frobenius inner product* is an extension of the inner product which takes two matrices as input instead of two vectors. It is defined as:

$$\langle A, B \rangle_F = \sum_{i,j} (A_{ij} \times B_{ij}) \tag{14}$$

where A and B are matrices of the same shape. Note that the definition naturally extends the regular inner product definition.

Just like the inner product is used to project a vector onto another one, we use the *Frobenius inner product* to project the stimulus onto the STA image at a single time step prior to a spike. We perform this projection for each stimulus using the following script:

```
1 stim_projections = np.zeros(np.size(counts))
2 for i in range(np.size(counts)):
3     stim_projections[i] = np.sum(STAs[:,:, 0] * stim[:,:, i])
4 stim_projections /= np.max(stim_projections)
```

The 4th line of the script ensures that we normalize the resulting projections to a maximum of 1. We plot an histogram with 100 bins using the `stim_projections` array:

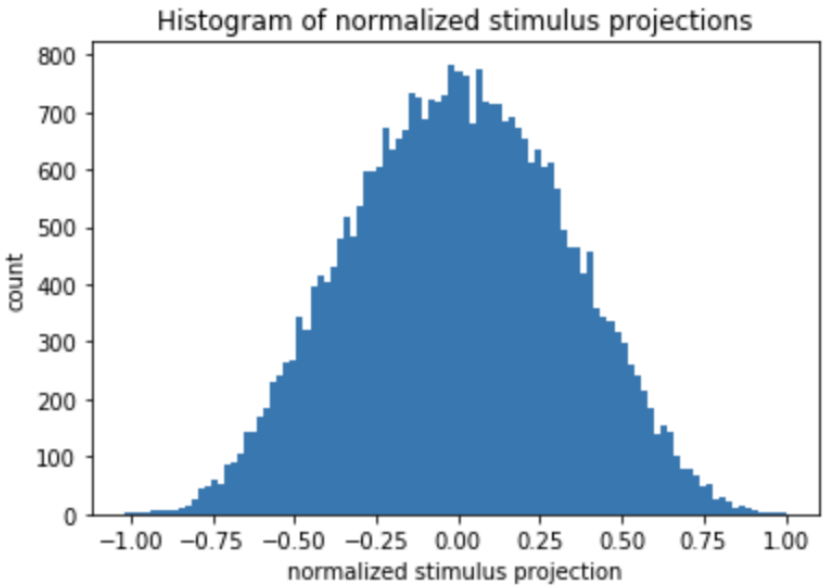

Figure 20: Histogram of normalized stimulus projections

We also create another histogram from stimulus projections at time bins where a non-zero spike count was observed. The data for this histogram is obtained by the following code:

```
1 nonzero_stim_indices = np.where(counts != 0)[0]
2 nonzero_stim_projs = np.zeros(np.size(nonzero_stim_indices))
3 for i in range(np.size(nonzero_stim_projs)):
4     nz = nonzero_stim_indices[i]
5     if nz >= 1:
6         nonzero_stim_projs[i] = np.sum(STAs[:,:, 0] * stim[:,:, nz - 1])
7 nonzero_stim_projs /= np.max(nonzero_stim_projs)
```

Here we only perform the projection when we see a non-zero spike count.

We use the same number of bins, 100, and plot an histogram using the `nonzero_stim_projs` array to which we applied the same normalization:

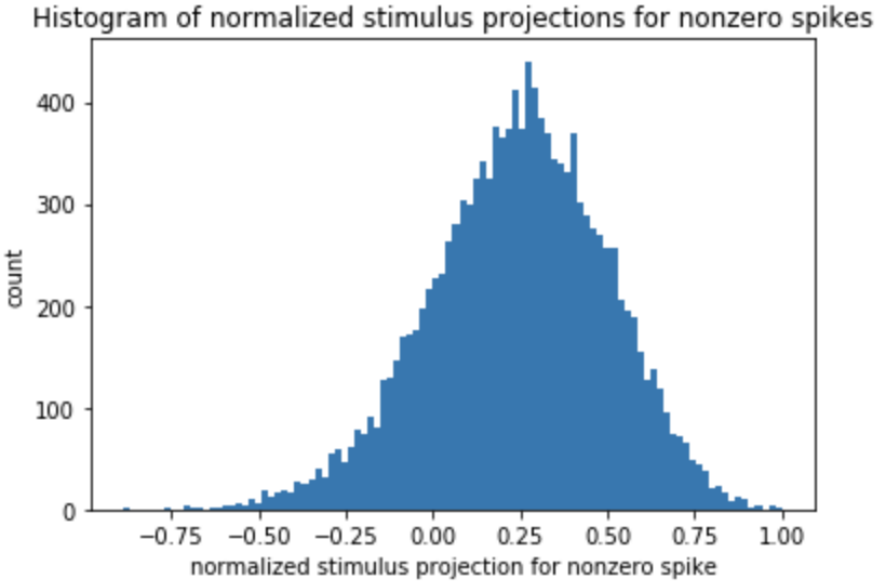

Figure 21: Histogram of normalized stimulus projections for nonzero spikes

We compare the histograms in Figure 20 and Figure 21 by plotting them in the same bar plot:

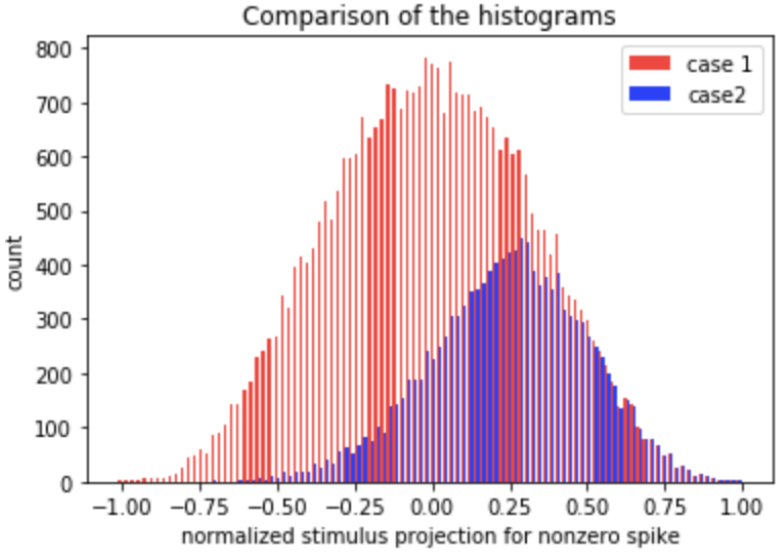

Figure 22: Comparison of the histograms

In Figure 21 the histogram labeled as "case 1" refers to the one, where we include the stimuli eliciting no spikes to our projection and the histogram labeled as "case 2" refers to the one, where only the spike eliciting stimuli are considered. We can observe that the red histogram is a discretized gaussian with 0 mean. This tells us most of our projection values are very close to 0. However, the blue histogram centers the data around 0.25 meaning most of the projection values are around this particular number. We can conclude that the absence of the stimuli that do not elicit any spike shifts the center of the data significantly. Thus, spike-eliciting stimuli can be easily discriminated by projecting them onto the STA image.

Question 3

a) Given two compiled `Matlab` functions that generate the response of two unknown neurons, "neuron 1" and "neuron 2", we want to check whether the responses of the neurons depend on time or not. The compiled functions accept a stimulus input that is a size 50 column vector. The first index of the stimulus array denotes the impulse at the first time instance, second index denotes the impulse at the second time instance and so on.

We use the following `Matlab` script to test the time dependency of the neuron responses:

```
response_magnitudes1 = zeros(50, 1);
response_magnitudes2 = zeros(50, 1);
for i = 1:50
    stimulus = zeros(1, 50);
    stimulus(i) = 1;
    fprintf('The stimulus:\n');
    disp(stimulus)
    response1 = unknownNeuron1(stimulus);
    fprintf('Response of neuron 1:\n');
    disp(response1);
    response_magnitudes1(i) = norm(response1);
    response2 = unknownNeuron2(stimulus);
    fprintf('Response of neuron 2:\n');
    disp(response2);
    response_magnitudes2(i) = norm(response2);
end
fprintf('Even the magnitudes of the responses are different for neuron 1.\n');
disp(response_magnitudes1);
fprintf('Neuron 2 elicits the same response magnitude for all stimulus.\n');
fprintf('However, the response is different as it can be seen from the previous
    print outs.\n');
disp(response_magnitudes2);
```

The code above has a long print out, so we do not provide the output here.

After inspecting the output, we see that the response of neuron 1 is a size 50 column vector were all entries are the same. These response vectors are different for each stimulus input. Hence, we say that the response of neuron 1 is time-variant.

In a similar manner we inspect the output again and see that neuron 2 elicits the same response magnitude for all stimuli. However, when we look at the individual response outputs we see that they are different. We basically shift a value of 1 among all other values of zeros in the stimuli vectors. When the 1 is at the ith index of a stimulus vector, it means that

an impulse is generated at the ith time step. Neuron 2's response is such that it produces a value of 0.6065 at the ith index of the response vector and it produces a value of 0.4111 for all other indices. Thus, we say that the response of neuron 2 is as well time-variant.

Another thing we want to check for the responses is whether the response to a sum of these impulses is equal to the sum of their individual responses. We look at a sample case where the first and the second indices of the stimulus vector are set to 1 and others remain as zeros. We run this case using the following `Matlab` code:

```
stimulus1 = zeros(1, 50);
stimulus1(1) = 1;
stimulus2 = zeros(1, 50);
stimulus2(2) = 1;
sample_stimulus = stimulus1 + stimulus2;
fprintf('Sample stimulus:\n');
disp(sample_stimulus);
expected_response1 = unknownNeuron1(stimulus1) + unknownNeuron1(stimulus2);
actual_response1 = unknownNeuron1(sample_stimulus);
fprintf('Check for neuron 1:\n');
is_expected1 = isequal(expected_response1, actual_response1);
disp(is_expected1);
expected_response2 = unknownNeuron2(stimulus1) + unknownNeuron2(stimulus2);
actual_response2 = unknownNeuron2(sample_stimulus);
fprintf('Check for neuron 2:\n');
is_expected2 = isequal(expected_response2, actual_response2);
disp(is_expected2);
```

The check fails for neuron 2 in the sample case so we say that its response to a sum of impulses is not equal to the sum of its corresponding responses. However, the check evaluates true for neuron 1. Thus we repeat the experiment for other sample stimulus vectors and ensure that this is always the case for neuron 1. At the end, we conclude that the sum of neuron 1's individual responses to a set of impulses is equal to its response to the sum of the impulses.

b) A `Matlab` function is written to construct the response profile of neuron 1 as a function of stimulus temporal frequency. The stimulus vectors are assumed to be cosines with unity amplitude and zero phase. The function plots the response magnitude as a function of stimulus intensity in the range [0, 20]. The function itself and an helper function it uses are given at the beginning of the next page.

```
function analyze_stimulus_frequency()
    frequencies = linspace(0, pi / 5);
    stimulus = zeros(100, 50);
    response_magnitudes = zeros(1, 100);
    for i = 1:100
        stimulus(i,:) = stimulus(i,:) + cos((1:50) * frequencies(i));
        response_magnitudes(i) = process_response1(stimulus(i,:));
    end
    [response_magnitude, optimal_index] = max(response_magnitudes);
    fprintf('The optimal frequency value is:\n');
    disp(frequencies(optimal_index))
    fprintf('The maximum response magnitude is:\n');
    disp(response_magnitude)
    figure
    plot(frequencies, response_magnitudes);
    title('Response Magnitude vs Temporal Frequency');
    ylabel('Response Magnitude');
    xlabel('Temporal Frequency');
end

function response_magnitude = process_response1(stimulus)
    response = unknownNeuron1(stimulus);
    response_magnitude = norm(response);
end
```

The plot generated by this function is the following:

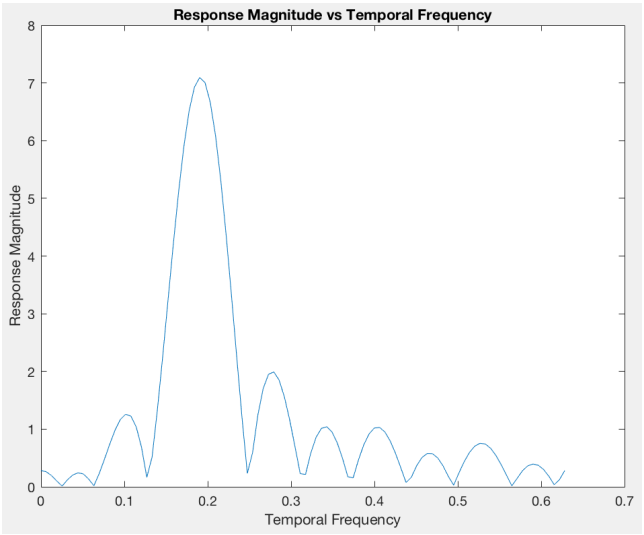

Figure 23: Response Magnitude vs Temporal Frequency

The optimal temporal frequency value for the stimulus is found by the function to be 0.1904. This value of temporal frequency results in a response magnitude of 7.0948.

c) Another `Matlab` function is implemented to construct the response profile of neuron 2 to stimulus intensity. Each stimulus vector is assumed to be constant across time. The function plots the response magnitude as a function of stimulus intensity in the range [0, 20].

The function is provided below together with a helper it uses:

```
function analyze_stimulus_intensity()
    intensities = linspace(0, 20);
    stimulus = zeros(100, 50);
    response_magnitudes = zeros(1, 100);
    for i = 1:100
        stimulus(i,:) = stimulus(i,:) + intensities(i);
        response_magnitudes(i) = process_response2(stimulus(i,:));
    end
    [response_magnitude, optimal_index] = max(response_magnitudes);
    fprintf('The optimal intensity value is:\n');
    disp(intensities(optimal_index))
    fprintf('The maximum response magnitude is:\n');
    disp(response_magnitude)
    figure
    plot(intensities, response_magnitudes);
    title('Response Magnitude vs Stimulus Intensity');
    ylabel('Response Magnitude');
    xlabel('Stimulus Intensity');
end

function response_magnitude = process_response2(stimulus)
    response = unknownNeuron2(stimulus);
    response_magnitude = norm(response);
end
```

The next page begins with the plot generated by this function. Looking at the plot we can say that there is no linear relation between the response magnitude and stimulus intensity. Thus, neuron 2 does not respond linearly to stimulus intensity. The function tells us that the optimal intensity value for neuron 2 is 4.0404 and the corresponding maximal response magnitude is 7.0704.

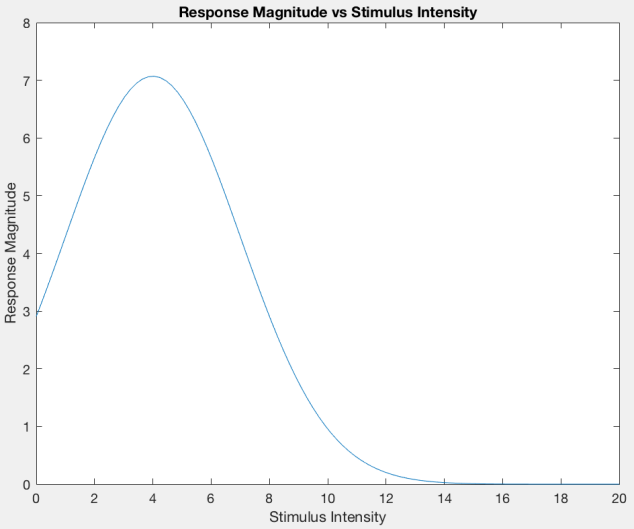

Figure 24: Response Magnitude vs Stimulus Intensity

d) In practice a non-noisy stimulus is not realistic. Hence, we repeat the experiments carried out in parts **b** and **c** with the presence of an additive noise that corrupts the input stimulus. The noise is denoted by $n(t)$ which is a stationary Gaussion process centered at 0 with a standard deviation of σ.

A function to carry out the experiment in part **b** with additive noise is implemented. The function performs 100 independent trials and then constructs a plot using the mean response magnitudes where error bars are also specified. Error bars correspond to the 68% confidence interval, which is the same as the interval that the data points are a single standard deviation away from the center. The function follows:

```
function analyze_noisy_stim_frequency(sigma)
    frequencies = linspace(0, pi / 5, 500);
    response_magnitudes = zeros(100, 500);
    stimulus = zeros(500, 50);
    for i = 1:100
        for j = 1:500
            stimulus(j,:) = normrnd(0, sigma, [1, 50]) + cos((1:50) * frequencies(j
                ));
            response_magnitudes(i, j) = process_response1(stimulus(j,:));
        end
    end
    means = mean(response_magnitudes);
    figure
    hold on
    errorbar(means, std(response_magnitudes), '.r');
```

```
15      plot(1:500, means, '.b');
16      header = ['Response Magnitude vs Temporal Frequency (std=', num2str(sigma), ')
            '];
17      title(header);
18      ylabel('Average Response Magnitude');
19      xlabel('Temporal Frequency');
20      hold off
21  end
```

This function is used to plot the average response magnitudes for the σ values 1, 2.5 and 5. The resulting plots follow:

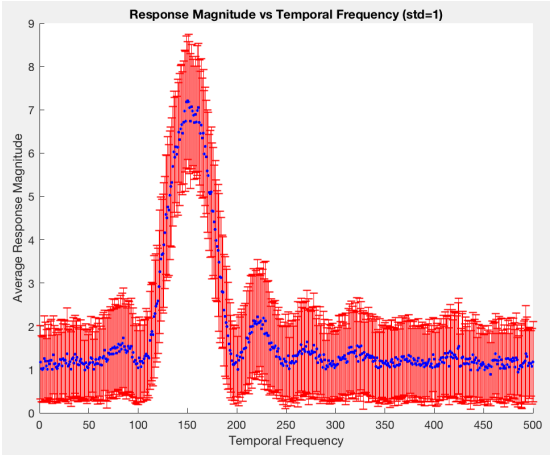

Figure 25: Response Magnitude vs Temporal Frequency $\sigma = 1$

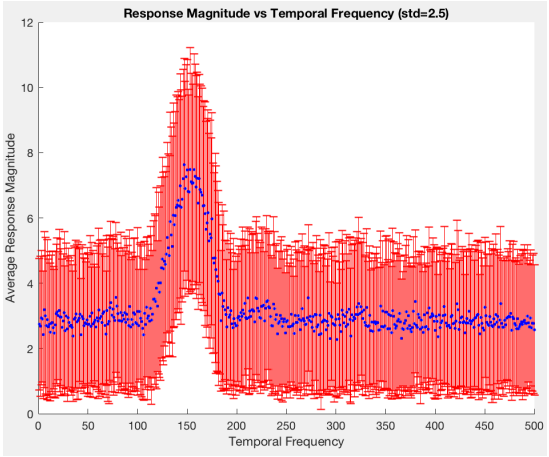

Figure 26: Response Magnitude vs Temporal Frequency $\sigma = 2.5$

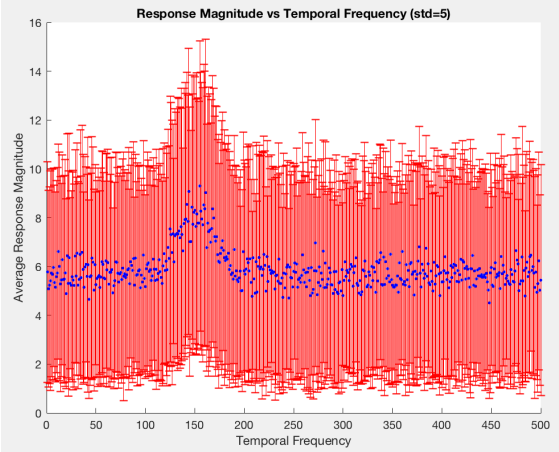

Figure 27: Response Magnitude vs Temporal Frequency $\sigma = 5$

An analogous function is written to perform the noisy version of the experiment in part **c**. The `Matlab` function is given below:

```
function analyze_noisy_stim_intensity(sigma)
    intensities = linspace(0, 20, 500);
    response_magnitudes = zeros(100, 500);
    stimulus = zeros(500, 50);
    for i = 1:100
        for j = 1:500
            stimulus(j,:) = normrnd(0, sigma, [1,50]) + intensities(j);
            response_magnitudes(i, j) = process_response2(stimulus(j,:));
        end
    end
    means = mean(response_magnitudes);
    figure
    hold on
    errorbar(means, std(response_magnitudes), '.r');
    plot(1:500, means, '.b');
    header = ['Response Magnitude vs Stimulus Intensity (std=', num2str(sigma), ')
        '];
    title(header);
    ylabel('Average Response Magnitude');
    xlabel('Stimulus Intensity');
    hold off
end
```

Plots generated by this function for the same set of σ values are given in the next pages.

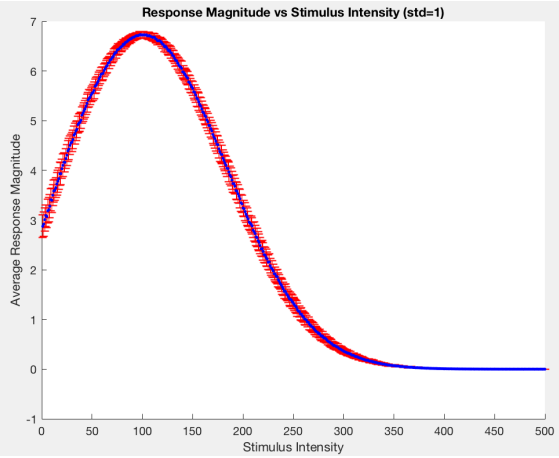

Figure 28: Response Magnitude vs Stimulus Intensity $\sigma = 1$

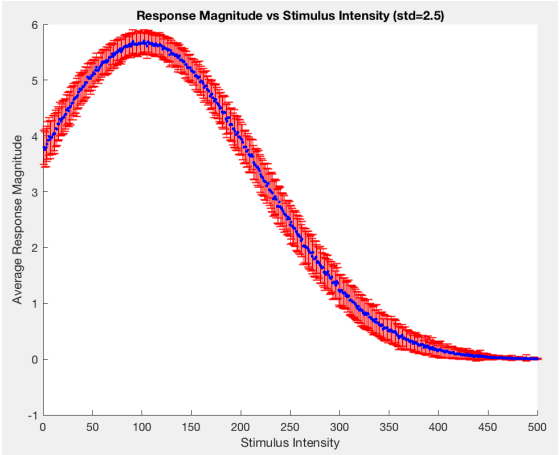

Figure 29: Response Magnitude vs Stimulus Intensity $\sigma = 2.5$

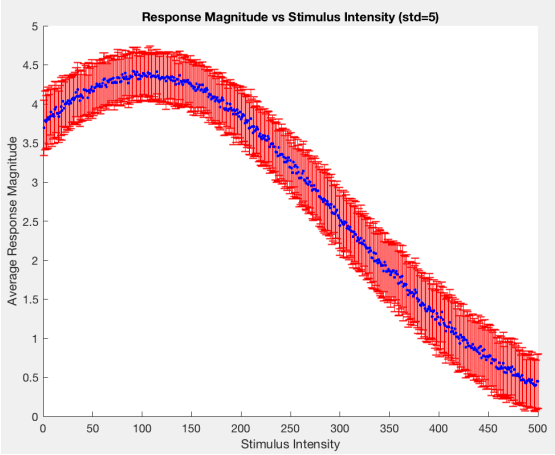

Figure 30: Response Magnitude vs Stimulus Intensity $\sigma = 1$

Looking at the figures we can conclude that the noise does not affect the overall response of the neurons. It may shift the optimal values a little but the characteristics of the response remains the same. However, note that for the case of temporal intensity the increased level of noise slightly damps the oscillations in the response.

Question 4

a) The on-center difference-of-gaussians (DOG) center-surround receptive field centered at 0 is given as:

$$D(x,y) = \frac{1}{2\pi\sigma_c^2} e^{-(x^2+y^2)/2\sigma_c^2} - \frac{1}{2\pi\sigma_s^2} e^{-(x^2+y^2)/2\sigma_s^2} \tag{15}$$

where σ_s denotes the standard deviation of the surrounding gaussian and σ_c denotes that of the central gaussian.

The following `Python` method implements the function given in (15):

```python
def DOG(x, y, sigma_c, sigma_s):
    """
    Implements an on-center difference-of-gaussians (DOG) center-surround
    filter centered at 0.
    Args:
        x: x coordinate
        y: y coordinate
        sigma_c: Central gaussian width
        sigma_s: Surround gaussian width
    Returns:
        result: The output of the DOG filter, D(x, y)
    """
    const_c = 1 / (2 * sigma_c ** 2)
    const_s = 1 / (2 * sigma_s ** 2)
    exp_c = np.exp(-(x ** 2 + y ** 2) * const_c)
    exp_s = np.exp(-(x ** 2 + y ** 2) * const_s)
    result = (const_c / np.pi) * exp_c - (const_s / np.pi) * exp_s
    return result
```

To be able to understand why and how the DOG center-surround receptive field is useful, we sample a 21x21 matrix using the `Python` method above.

Another `Python` method is written in order to sample a matrix using the DOG center-surround receptive field. The method accepts the shape of a matrix as an argument (which is (21, 21) in our case), considers the center of the matrix ((10, 10) in our case) as the center of the receptive field which is (0, 0) and applies the function specified in (15) to each row, column pair of the matrix. The method is given at the beginning of the next page.

```python
def DOG_receptive_field(x, y, shape, sigma_c, sigma_s):
    """
    Samples a matrix of the specified shape using the DOG filter.
    Args:
        x: x coordinate of the receptive filter's center
        y: y coordinate of the receptive filter's center
        shape: Shape of the sampled matrix
        sigma_c: Central gaussian width for the receptive filter
        sigma_s: Surround gaussian width for the receptive filter
    Returns:
        sample: Matrix sample generated from the DOG filter
    """
    sample = np.zeros(shape)
    for i in range(- int(shape[0] / 2), 1 + int(shape[0] / 2)):
        for j in range(- int(shape[1] / 2), 1 + int(shape[1] / 2)):
            sample[x + i + int(shape[0] / 2),
                   y + j + int(shape[1] / 2)] = DOG(x + i, y + j, sigma_c, sigma_s)
    return sample
```

We obtain our 21x21 sample matrix referred as "DOG kernel" by running the following line of code where `SHAPE` is (21, 21), `SIGMA_C` (standard deviation of the central gaussian) is 2 and `SIGMA_S` (standard deviation of the surrounding gaussian) is 4:

```python
DOG_kernel = DOG_receptive_field(0, 0, SHAPE, SIGMA_C, SIGMA_S)
```

A 2D view of the DOG kernel, which is plotted using `imshow`, is provided below:

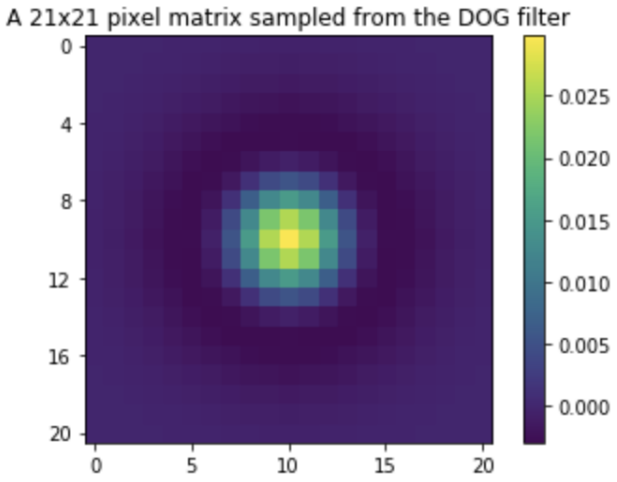

Figure 31: The DOG kernel (2D)

Oftentimes a 3D view of a function of 2 variables gives a better visualization. Hence, a 3D view of the DOG kernel is plotted using the `plot_surface` function of the `mplot3d` package inside the `mpl_toolkits` library:

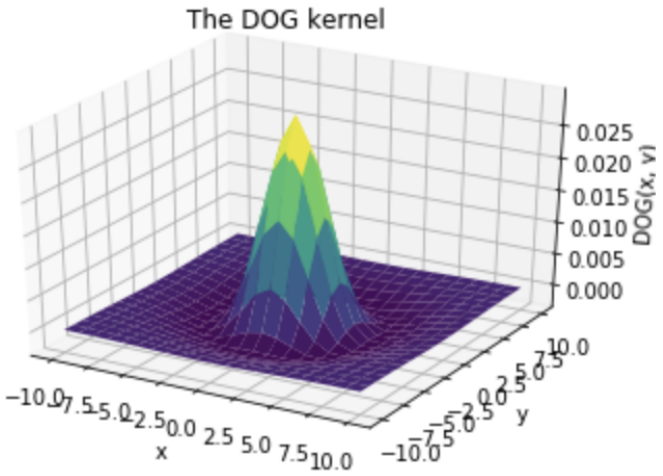

Figure 32: The DOG kernel (3D)

As it can be clearly seen from Figure 32, the DOG kernel favors the central pixels and suppresses the pixels that are away from the center. Notice that the standard deviation of the surrounding gaussian must be grater than that of the central gaussian to observe the pit encircling the peak.

The DOG kernel is especially useful to model some LGN cells which exhibit the exact same behavior that favor the central parts of an image and suppress the parts that are distant from the center. In fact, we worked on such an LGN neuron in **Question 2**. At bottom, we can use the DOG center-surround receptive field whenever we want to establish a model that it sensitive to the central region of an image.

b) The neural activity of an LGN cell can be expressed as the amount of overlap between the cells' DOG receptive fields and the visual inputs as the receptive fields shift over the visuals. This procedure is convolution in words. Thus, we can describe the neural activity using the convolution of the DOG kernel over an image.

The required 2D convolution procedure is implemented as a `Python` function. The function places the image topographically according to the center of the DOG kernel, computes the frobenius inner product of the kernel and the overlapping part of the image, records the result as the neural activity at that particular pixel and repeatedly does the same operations while shifting the image. Since the image does not overlap properly with the kernel at its boundaries, the function adds a padding to the image which consists of 0 values. The function is given at the beginning of the next page.

```python
def convolve(image, kernel):
    """
    Given a kernel matrix, computes the 2 dimensional convolution
    of the kernel and an iamge.
    Args:
        image: The given image
        kernel: The given kernel matrix to be used in the convolution
    Returns:
        result: The resulting matrix after the convolution
    """
    pad_x = int(np.shape(kernel)[0] / 2)
    pad_y = int(np.shape(kernel)[1] / 2)
    img_padding = np.zeros((2 * pad_x + np.shape(image)[0], 2 * pad_y + np.shape(
        image)[1]))
    img_padding[pad_x: np.shape(image)[0] + pad_x, pad_y: np.shape(image)[1] +
        pad_y] = image
    result = np.zeros(np.shape(image))
    for i in range(pad_x, np.shape(image)[0] + pad_x):
        for j in range(pad_y, np.shape(image)[1] + pad_y):
            result[i - pad_x, j - pad_y] = np.sum(
                img_padding[i - pad_x: i + pad_x + 1, j - pad_y: j + pad_y + 1] *
                    kernel)
    return result
```

We apply the `convolve` function to the following image:

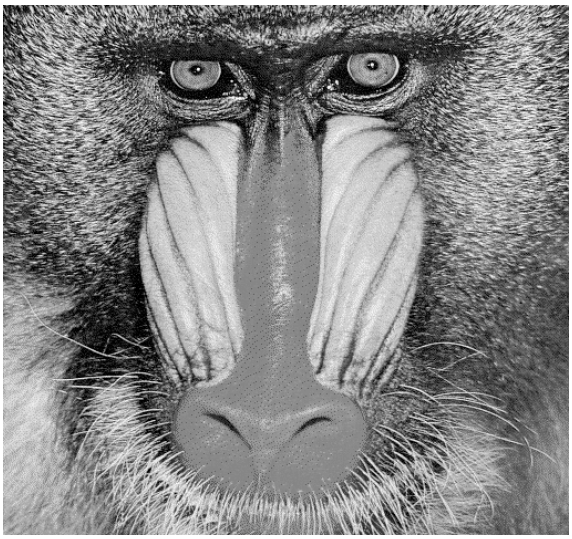

Figure 33: The image that will be used in convolution

After loading the image in Figure 33 as `image`, we obtain the convolution result using the line:

```
1 DOG_image = convolve(image, DOG_kernel)
```

The `DOG_image` is the following:

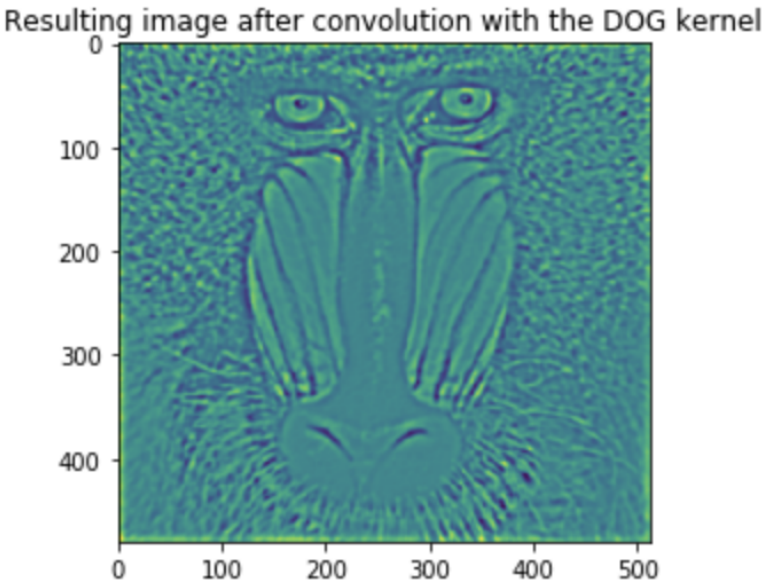

Figure 34: The image after convolution with the DOG kernel

A repeated DOG center-surround receptive field acts like a spatial filter, and can be characterised by its frequency-domain transfer function. The transfer function for DOG center-surround receptive field is *bandpass* which makes the mid-range spatial frequencies more prominent and emphasize edges in an image.

We can see that the edges in Figure 33 are accentuated in Figure 34, while the other parts of the image are masked. This agrees with out theoretical expectations and shows us that convolution with a DOG kernel indeed emphasizes edges in an image.

c) To get a better view of the edges emphasized in Figure 34, we can apply thresholding. Thresholding is a common method in image processing where we create a binary image by selecting pixels in a grayscale image that has a value greater than a certain threshold. It is obvious that the edges are darker than other parts of the image in Figure 34. Hence, we must tune a threshold value such that the edges stay below the threshold while other parts of the image stay above it. A simple `Python` function that performs thresholding is given in the next page.

```python
def detect_edges(filtered_image, threshold):
    """
    Given an appropriately filtered image and an optimal threshold
    value, sets the pixels to 1 if they are above the threshold, and
    to 0 if they are below the threshold.
    Args:
        filtered_image: The image that is filtered such that it is
            ready for thresholding
        threshold: An optimal threshold value for edge detection
    Returns:
        result: Resulting image after thresholding
    """
    result = filtered_image
    result[np.where(filtered_image >= threshold)] = 1
    result[np.where(filtered_image < threshold)] = 0
    return result
```

After some trial and error we find the threshold value that best separates the edges. The following lines apply thresholding to the image generated after convolution:

```
THRESHOLD = -4
img_edges_detected = detect_edges(DOG_image, THRESHOLD)
```

The resulting image, `img_edges_detected`, is the following:

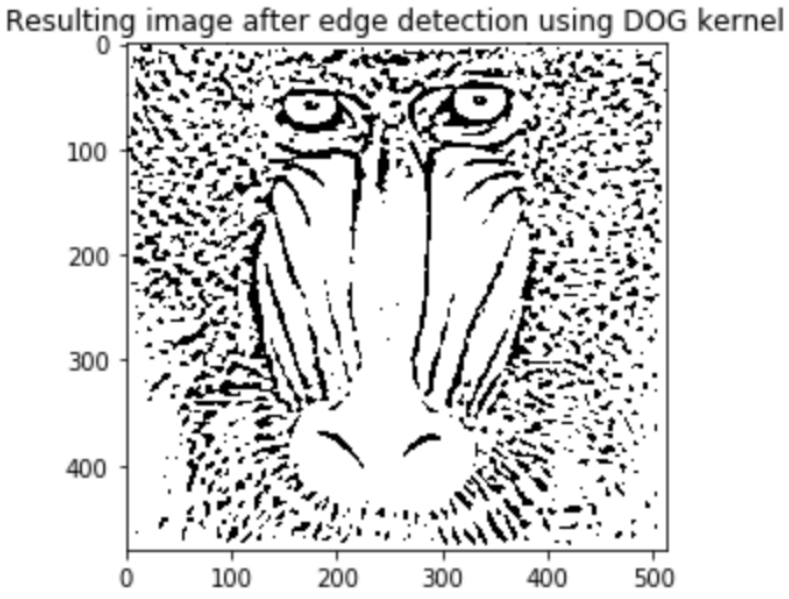

Figure 35: The image that will be used in convolution

d) Similar to DOG center-surround receptive field, Gabor receptive field is given by:

$$D(\vec{x}) = exp(-(k\vec{(\theta)} \cdot \vec{x})^2/2\sigma_l^2 - (k_\perp\vec{(\theta)} \cdot \vec{x})^2)/2\sigma_w^2)\ cos(2\pi \frac{k_\perp\vec{(\theta)} \cdot \vec{x}}{\lambda} + \phi) \tag{16}$$

where the standard deviations σ_l, σ_w, the orientation angle θ, the phase angle ϕ and the constant λ are the parameters. $k\vec{(\theta)}$ is a unit vector with orientation θ and $k_\perp\vec{(\theta)}$ is another unit vector perpendicular to it.

(16) is implemented in `Python` as follows:

```
def Gabor(x, theta, sigma_l, sigma_w, phi, lambda_):
    """
    Implements a Gabor filter with a specified orientation, and other
    specified parameters.
    filter centered at 0.
    Args:
        x: The coordinate vector
        theta: The orientation
        sigma_l: A constant
        sigma_w: A constant
        phi: The phase angle
        lambda_: A constant
    Returns:
        result: The output of the Gabor filter, D(x)
    """
    k = np.array([np.sin(theta), np.cos(theta)])
    k_orth = np.array([np.cos(theta), -np.sin(theta)])
    inner_k = k.dot(x)
    inner_k_orth = k_orth.dot(x)
    exp = np.exp(- (inner_k ** 2) / (2 * (sigma_l ** 2)) - (inner_k_orth ** 2) /
        (2 * (sigma_w ** 2)))
    result = exp * np.cos(phi + 2 * np.pi * inner_k_orth / lambda_)
    return result
```

The Gabor receptive field is used analogously to part **a**. The code used to sample a matrix with a specified shape using the `Gabor` function is given below:

```
def Gabor_receptive_field(x, y, shape, theta, sigma_l, sigma_w, phi, lambda_):
    """
    Samples a matrix of the specified shape using the Gabor filter.
    Args:
        Args:
            x: x coordinate for the Gabor filter
```

```
            y: y coordinate for the Gabor filter
            shape: Shape of the sampled matrix
            theta: The orientation for the Gabor filter
            sigma_l: A constant for the Gabor filter
            sigma_w: A constant for the Gabor filter
            phi: The phase angle for the Gabor filter
            lambda_: A constant for the Gabor filter
    Returns:
        sample: Matrix sample generated from the Gabor filter
    """
    sample = np.zeros(shape)
    for i in range(- int(shape[0] / 2), 1 + int(shape[0] / 2)):
        for j in range(- int(shape[1] / 2), 1 + int(shape[1] / 2)):
            sample[x + i + int(shape[0] / 2),
                   y + j + int(shape[1] / 2)] = Gabor(np.array([x + i, y + j]),
                                                      theta, sigma_l, sigma_w,
                                                      phi, lambda_)
    return sample
```

Again, we sample a 21x21 "Gabor kernel" with the following line of code:

```
Gabor_kernel_90 = Gabor_receptive_field(0, 0, SHAPE, THETA_90, SIGMA_L, SIGMA_W,
    PHI, LAMBDA)
```

where (SHAPE) equals (21, 21), `THETA_90` equals $\pi/2$, `SIGMA_L` and `SIGMA_W` are 3, `PHI` is 0 and `LAMBDA` is 6. A 2D view of the `Gabor_kernel_90` follows:

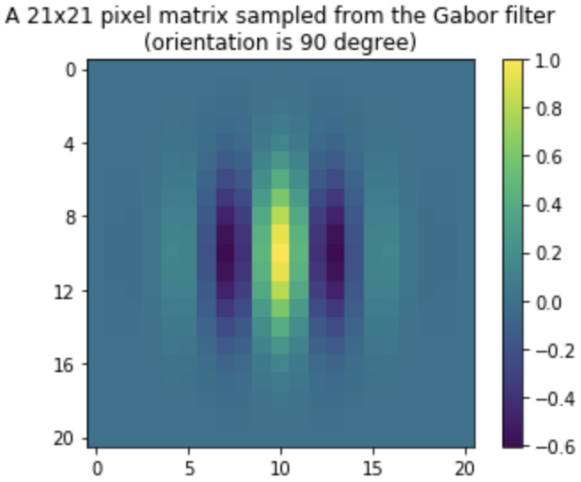

Figure 36: The Gabor kernel (2D) ($\theta = \pi/2$)

A 3D view of the kernel is also provided to better understand the what the function defined by (16) does:

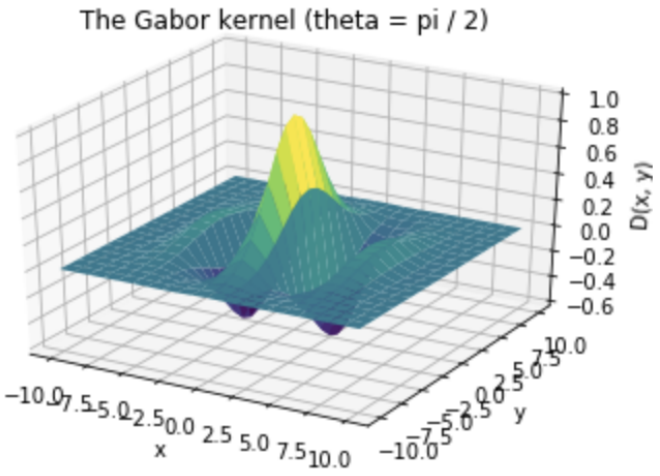

Figure 37: The Gabor kernel (3D) ($\theta = \pi/2$)

It is relatively harder to determine what the effect of the Gabor kernel is. Looking at Figure 36 and Figure 37, we can say that the kernel favors central pixels and it disregards the pixels that are distant from the center. In the DOG kernel, we saw a circular peak, however here in the Gabor kernel we can see that the peak is enclosed in a linear region. The peak is surrounded by two pits that are again enclosed in linear regions. From the center to the left and the right endpoints of the matrix, we see the peaks and pits alternate in a damped manner which is caused by the cosine factor. Another important observation is that the orientation of the linear regions enclosing the peaks and the pits is exactly θ degrees which is $\pi/2$ for this particular case.

Gabor Kernel is again useful to emphasize edges in an image but it involves additional sensitivity for direction and orientation. It is also known that this kernel is commonly used in texture analysis, which aims to find a particular frequency content in the image in specific directions in a localized region around the point or region of analysis [4] .

e) It is known that simple cells in the visual cortex (V1) have Gabor receptive fields. In order to see the neural activity of these cells we perform the same convolution process using the Gabor kernel:

```
1 Gabor_image_90 = convolve(image, Gabor_kernel_90)
```

The `convolve` function that was previously explained is used again since the procedure does not differ for the Gabor kernel. The resulting image is given at the beginning of the next page.

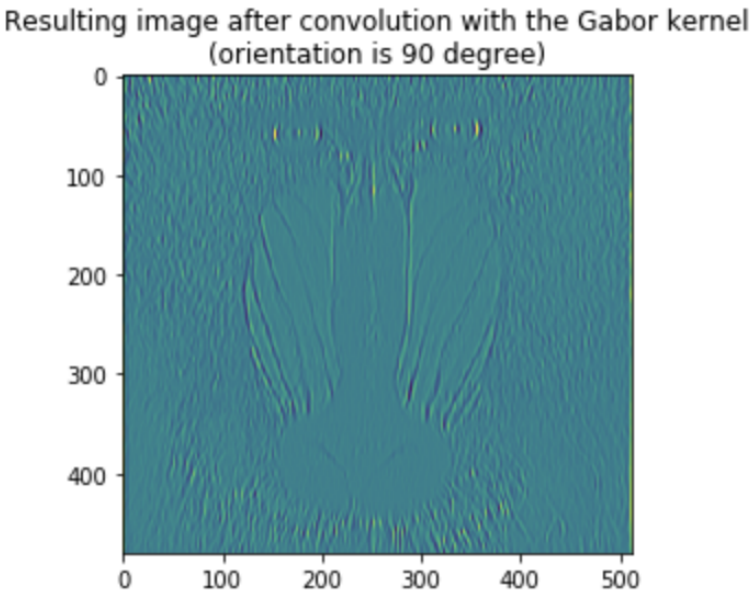

Figure 38: The image after convolution with the Gabor kernel where θ is $\pi/2$

It can be observed that the image above decouples the regions in the direction of 90 degrees, where the texture (or frequency) content is the same.

f) Obviously, a single orientation is not enough to spot all regions with the same texture content. Thus, we generate other Gabor kernels with a θ value of 0, $\pi/3$ and $\pi/6$.

The kernels are generated as follows:

```
1 Gabor_kernel_0 = Gabor_receptive_field(0, 0, SHAPE, THETA_0, SIGMA_L, SIGMA_W, PHI
      , LAMBDA)
2 Gabor_kernel_30 = Gabor_receptive_field(0, 0, SHAPE, THETA_30, SIGMA_L, SIGMA_W,
      PHI, LAMBDA)
3 Gabor_kernel_60 = Gabor_receptive_field(0, 0, SHAPE, THETA_60, SIGMA_L, SIGMA_W,
      PHI, LAMBDA)
```

Note that the values of `SIGMA_L`, `SIGMA_W`, `PHI` and `LAMBDA` are still the same.

2D and 3D views of the Gabor kernels generated above follow in the next pages.

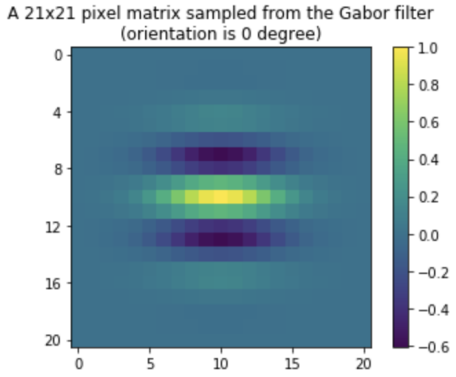

Figure 39: The Gabor kernel (2D) ($\theta = 0$)

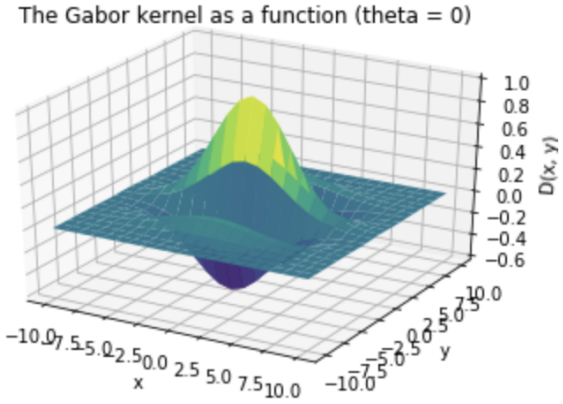

Figure 40: The Gabor kernel (3D) ($\theta = 0$)

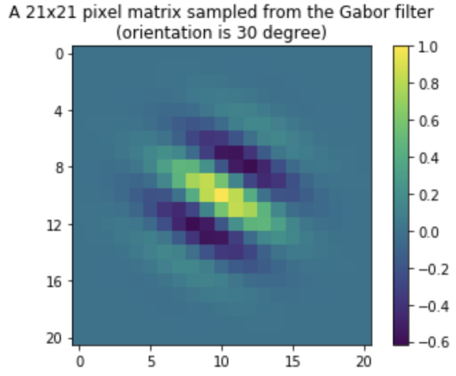

Figure 41: The Gabor kernel (2D) ($\theta = \pi/6$)

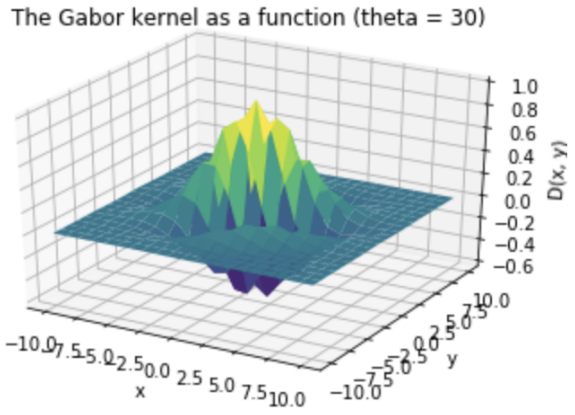

Figure 42: The Gabor kernel (3D) ($\theta = \pi/6$)

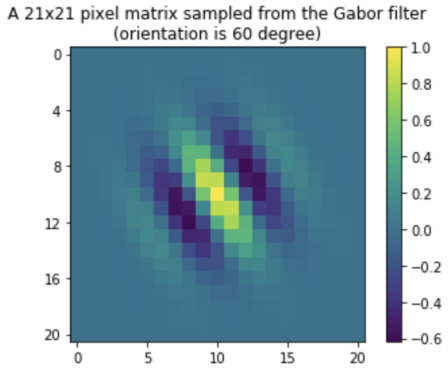

Figure 43: The Gabor kernel (2D) ($\theta = \pi/3$)

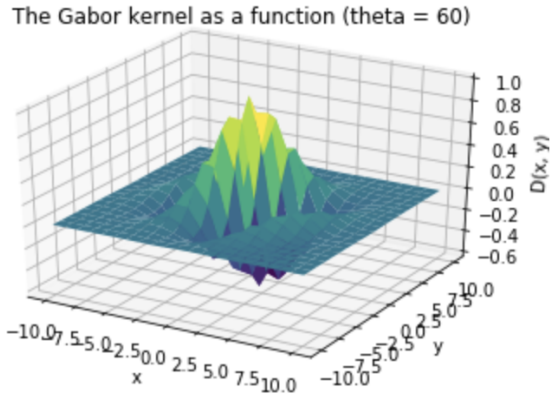

Figure 44: The Gabor kernel (3D) ($\theta = \pi/3$)

We perform convolution between these Gabor kernels and the original image. We obtain:

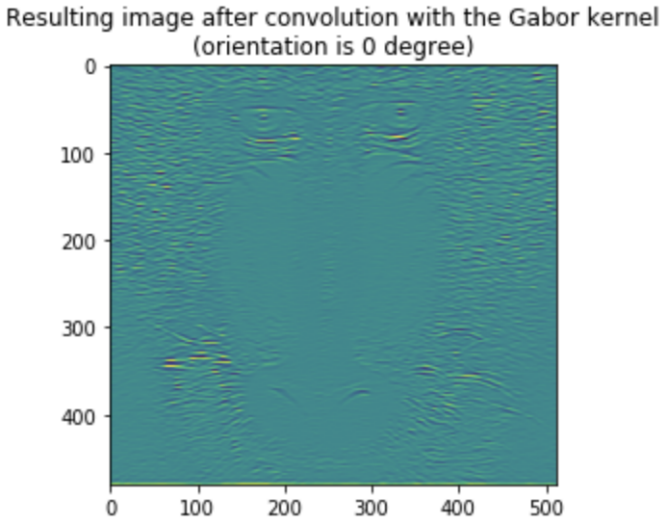

Figure 45: The image after convolution with the Gabor kernel where θ is 0

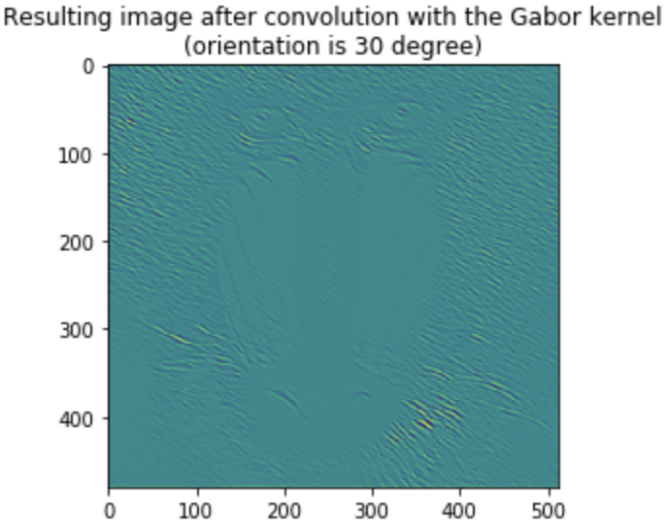

Figure 46: The image after convolution with the Gabor kernel where θ is $\pi/6$

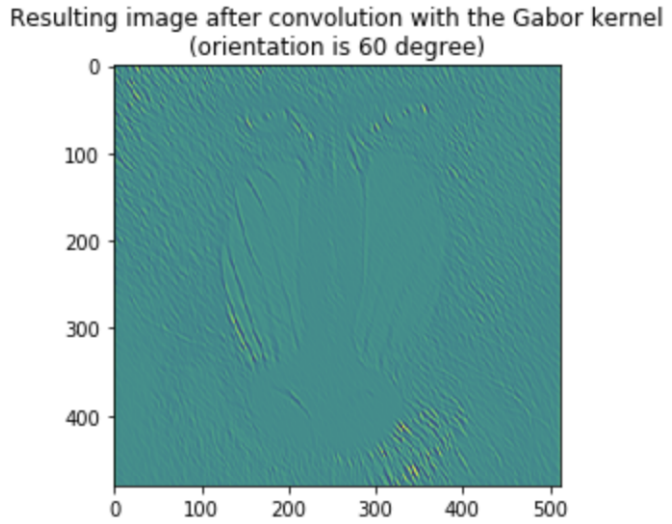

Figure 47: The image after convolution with the Gabor kernel where θ is $\pi/3$

As we expect, the images decoupled the regions of the same texture in their respective orientations. A good practice is to linearly combine the images resulting from the convolution using unit weight for each image. This allows us to be sensitive to multiple orientations. We perform the combination using a simple sum of all Gabor Kernels:

```
1 Gabor_image_combined = Gabor_image_0 + Gabor_image_30 + Gabor_image_60 +
    Gabor_image_90
```

The resulting image is displayed below:

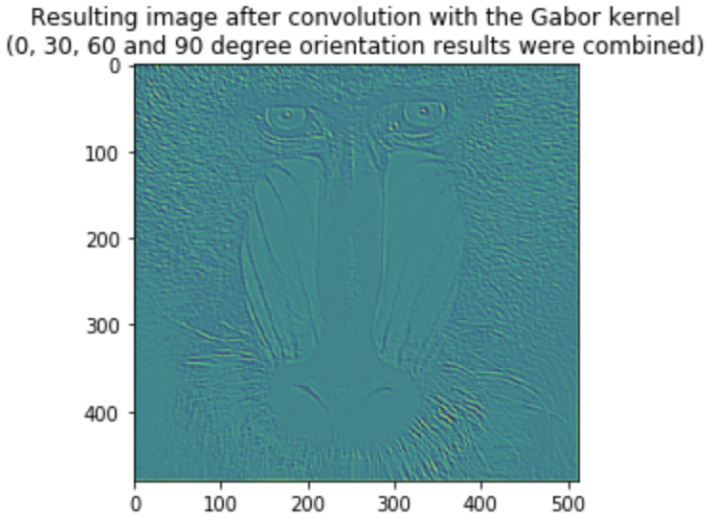

Figure 48: The combination of all Gabor kernels

The combined image clearly performs better than the individual ones in terms of texture analysis and edge detection. This is because of the fact that the edges that are emphasized well in a certain orientation may be emphasized poorly in another orientation.

To make further improvements, one can

- add more orientations.
- tune the phase angle.
- tune other parameters of the `Gabor function`.
- tune the weights when combining the individual kernels.

References

[1] [Accessed: 10- Mar- 2019]. [Online]. Available: http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx

[2] P. Dayan and L. Abbott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2014.

[3] "Lateral geniculate nucleus," [Accessed: 11- Mar- 2019]. [Online]. Available: https://www.sciencedirect.com/topics/neuroscience/lateral-geniculate-nucleus

[4] [Accessed: 11- Mar- 2019]. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TRAPP1/filter.html