hw6

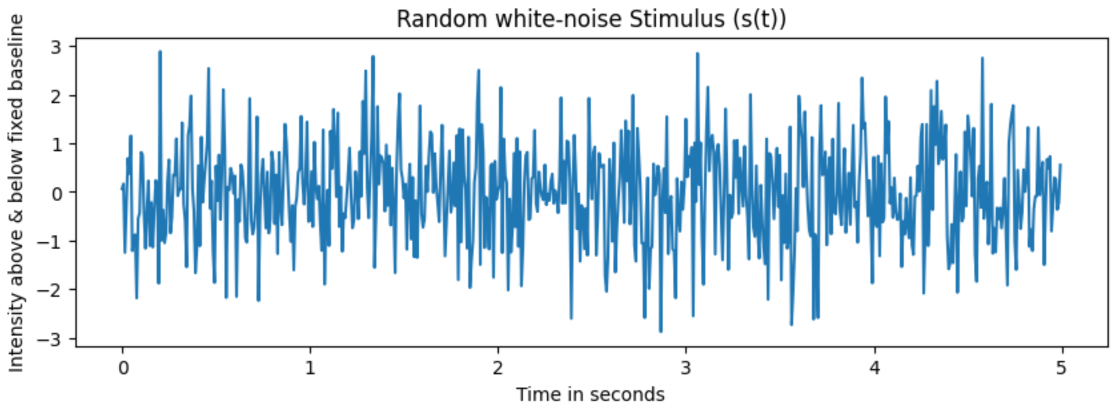
March 4, 2024

1 1. Simulating spike trains with Poisson statistics

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.fft import fft, fftfreq
from scipy.signal import convolve
from IPython.display import display, Math
```

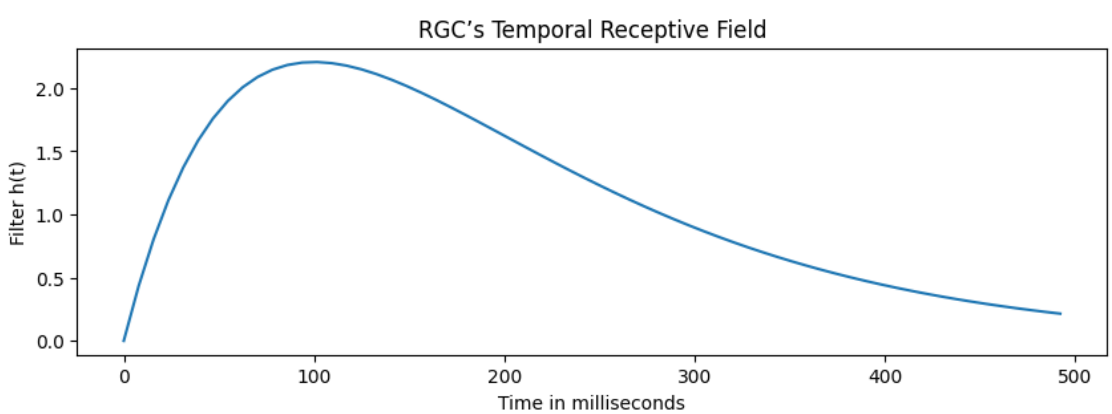
2 a

[2]:
```
refresh_rate_Fs = 128 #Fs
duration = 5*60
delta_t = 1/refresh_rate_Fs #aka time step or sampling interval & 1/Fs
sampling_times = np.arange(0, duration, delta_t) #aka time vector

s_t = np.random.normal(0, 1, len(sampling_times))

first_5s = 5*refresh_rate_Fs
plt.figure(figsize = (10, 3))
plt.plot(sampling_times[:first_5s], s_t[:first_5s])
plt.xlabel('Time in seconds')
plt.ylabel('Intensity above & below fixed baseline')
plt.title('Random white-noise Stimulus (s(t))')
plt.show()
```



3 b

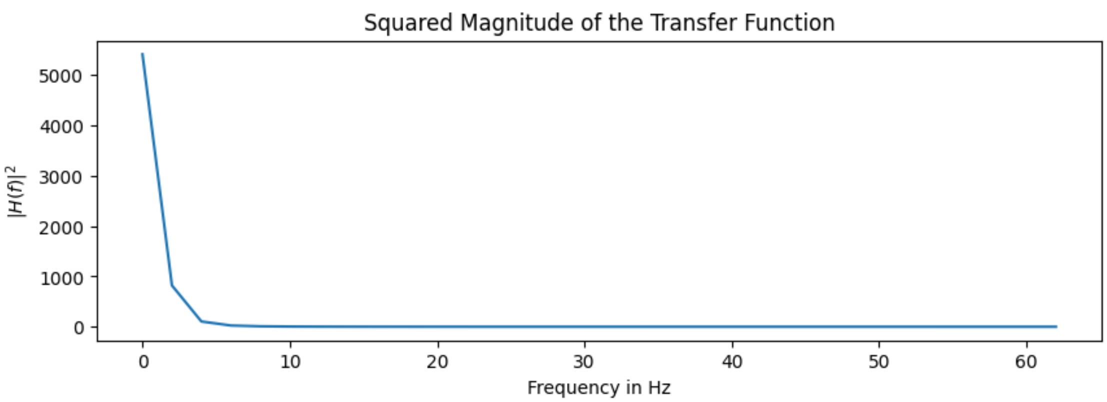
[3]:
```
tau_in_seconds = 0.1
R = 6
ht_sampling_times = np.arange(0, 0.5, delta_t)

scaling_amplitude = R * tau_in_seconds
h_t = (scaling_amplitude) * (ht_sampling_times / tau_in_seconds**2) * np.
 ↪exp(-ht_sampling_times / tau_in_seconds)

ht_st_time_conversion = ht_sampling_times*1000
plt.figure(figsize = (10, 3))
plt.plot(ht_st_time_conversion, h_t)
plt.xlabel('Time in milliseconds')
plt.ylabel('Filter h(t)')
plt.title('RGC’s Temporal Receptive Field')
plt.show()
```



4 c

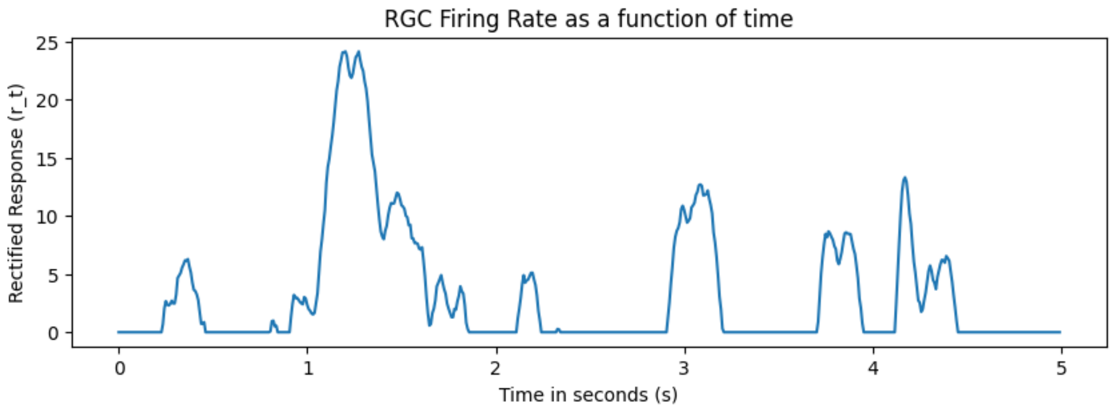
Low-pass filter - rise and decay & small τ value - the cutoff frequency and τ filters out high frequency due to its significant decline from its peak

[4]:
```
ft_ht = fft(h_t)
frequencies = fftfreq(len(ht_sampling_times), delta_t)

nyquist_index = len(frequencies)//2 #half of frequencies array
ft_slice = ft_ht[:len(frequencies)//2]
positive_frequencies = frequencies[:nyquist_index]

plt.figure(figsize = (10, 3))
plt.plot(positive_frequencies, np.abs(ft_slice)**2)
plt.xlabel('Frequency in Hz')
plt.ylabel('$|H(f)|^2$')
plt.title('Squared Magnitude of the Transfer Function')
plt.show()
```



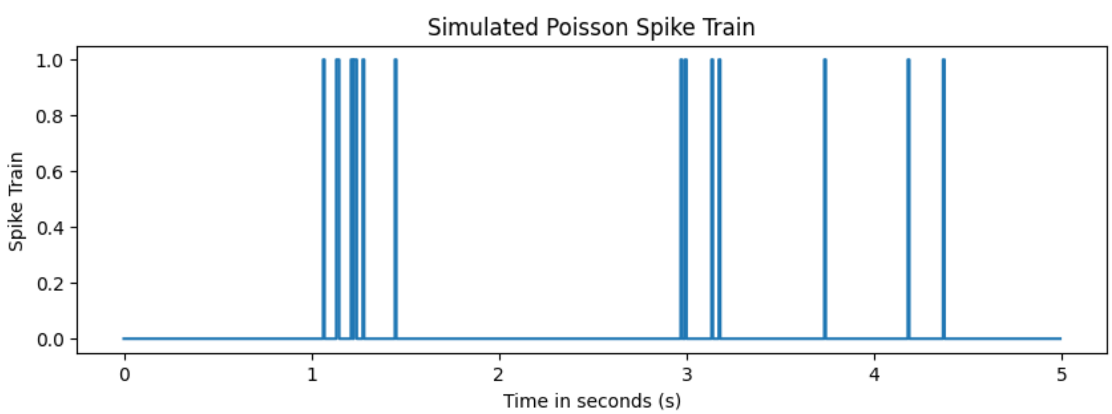
5 d

[5]:
```
r_t = convolve(s_t, h_t, mode = 'same')
rt_rectifying = np.maximum(r_t, 0)

plt.figure(figsize = (10, 3))
plt.plot(sampling_times[:first_5s], rt_rectifying[:first_5s])
plt.xlabel('Time in seconds (s)')
plt.ylabel('Rectified Response (r_t)')
plt.title('RGC Firing Rate as a function of time')
plt.show()
```



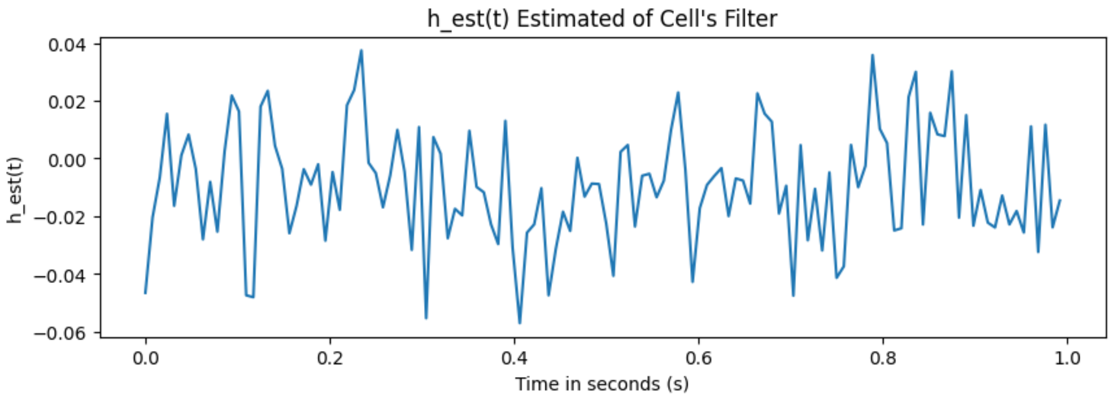
6 e

[6]:
```
rate_per_sample = rt_rectifying / refresh_rate_Fs #rate per second conversion
spst = np.random.poisson(rate_per_sample) #Simulated Poisson spike train

plt.figure(figsize = (10, 3))
plt.plot(sampling_times[:first_5s], spst[:first_5s], drawstyle = 'steps-pre')
plt.xlabel('Time in seconds (s)')
plt.ylabel('Spike Train')
plt.title('Simulated Poisson Spike Train')
plt.show()
```



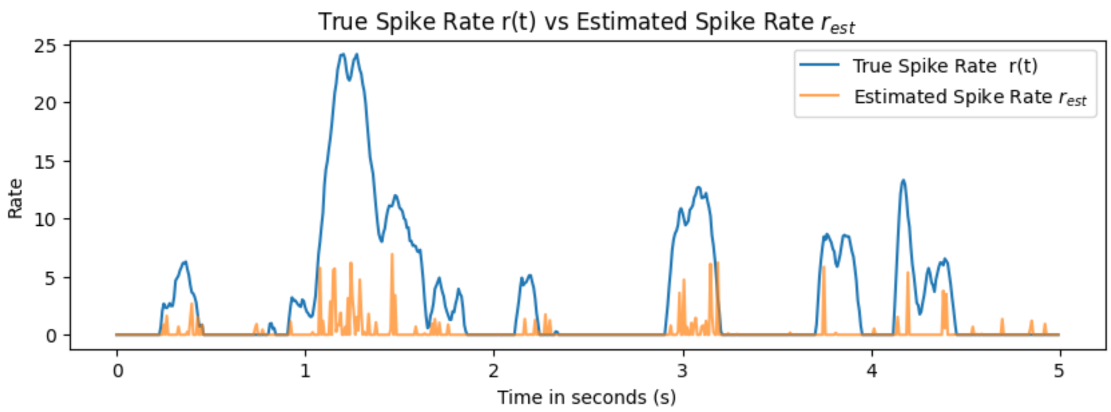
7 f

[7]:
```
time_reversed_stimulus  = s_t[::-1]
h_est = convolve(spst, time_reversed_stimulus, mode = 'same') / np.sum(spst)

plt.figure(figsize = (10, 3))
plt.plot(sampling_times[:refresh_rate_Fs], h_est[:refresh_rate_Fs])  # Plotting␣
↪first 1 second
plt.xlabel('Time in seconds (s)')
plt.ylabel('h_est(t)')
plt.title('h_est(t) Estimated of Cell\'s Filter')
plt.show()
```



8 g

[8]:
```
h_est[h_est < 0] = 0
r_est = convolve(s_t, h_est, mode = 'same')
r_est_rectified = np.maximum(r_est, 0)

plt.figure(figsize = (10, 3))
plt.plot(sampling_times[:first_5s], rt_rectifying[:first_5s], label = 'True␣
↪Spike Rate  r(t)')
plt.plot(sampling_times[:first_5s], r_est_rectified[:first_5s], label =␣
↪'Estimated Spike Rate $r_{est}$', alpha = 0.7)
plt.xlabel('Time in seconds (s)')
plt.ylabel('Rate')
plt.title('True Spike Rate r(t) vs Estimated Spike Rate $r_{est}$')
plt.legend()
plt.show()
```



---

9 2. Permutation-based analysis of spike train statistics

10 a

[9]:
```
shuffling_spst = np.random.permutation(spst) #randomize = shuffling
shuffling_h_est = convolve(shuffling_spst, time_reversed_stimulus, mode =␣
↪'same') / np.sum(shuffling_spst)
```

11 b

[10]:
```
shuffling_j = 1000
h_shuff = np.zeros((len(sampling_times), shuffling_j))

for j_index in range(shuffling_j):
    shuffling_spst = np.random.permutation(spst)
    h_shuff[:, j_index] = convolve(shuffling_spst, time_reversed_stimulus, mode␣
↪= 'same') / np.sum(shuffling_spst)
```

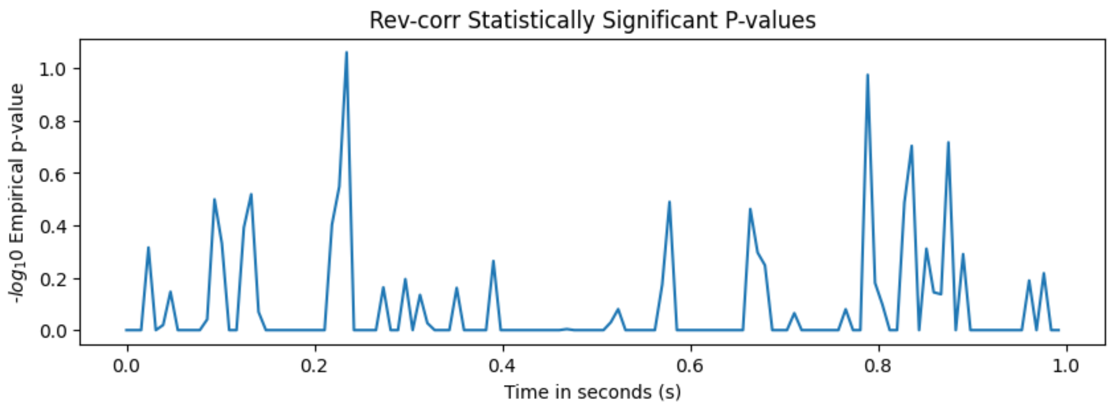
[11]:
```
h_shuff.shape
```

[11]: (38400, 1000)

12 c

[12]:
```
empirical_p_values = np.zeros(len(sampling_times))
for t_index in range(len(sampling_times)):
    empirical_p_values[t_index] = (np.sum(np.abs(h_shuff[t_index, :]) >= np.
↪abs(h_est[t_index])) + 1) / (shuffling_j + 1)
```

```
neg_log10_pt = -np.log10(empirical_p_values[:refresh_rate_Fs])
plt.figure(figsize = (10, 3))
plt.plot(sampling_times[:refresh_rate_Fs], neg_log10_pt)
plt.xlabel('Time in seconds (s)')
plt.ylabel('-$log_10$ Empirical p-value')
plt.title('Rev-corr Statistically Significant P-values')
plt.show()
```



13 d

[13]:
```
p_value = 0.05
expected_false_positives = p_value * refresh_rate_Fs
true_false_positives = np.sum(empirical_p_values[:refresh_rate_Fs] < p_value)

print("\nExpected p-value < 0.05 under the null hypothesis of no correlation:␣
  ↪", expected_false_positives)
print("\nTime bins actually passing empirical p-value threshold: ",␣
  ↪true_false_positives)
```

```
Expected p-value < 0.05 under the null hypothesis of no correlation:  6.4

Time bins actually passing empirical p-value threshold:  0
```

14 e

[14]:
```
N = 128
bonferroni_a_threshold = p_value / N

inversed_a_threshold = 1 / bonferroni_a_threshold
shuffles_rounded = np.ceil(inversed_a_threshold)
```

```
shuffling_total_int = int(shuffles_rounded)
shuffles_required = shuffling_total_int - 1

print("\nValue of N: ", N)
print("\nValue of  _Bonferroni: ", bonferroni_a_threshold)
display(Math(r'\text{{Number of shuffles performed to be able to find the time␣
 ↪bins satisfying }} p < \alpha_{{Bonferroni}}: {}'.format(shuffles_required)))
```

```
Value of N:  128

Value of  _Bonferroni:  0.000390625
```

Number of shuffles performed to be able to find the time bins satisfying $p < \alpha_{Bonferroni} : 2559$