Lab 3 Documentation: Circuit Analysis and Hardware Programming

Author: Jacqueline Arce
EE 104: Professor Pham

Overview:

In this lab, a variety of methods are introduced for circuit analysis. These methods include Python's Matrix Method, LT Spice Analysis, & Ahkab Method. The lab also introduces Jupyter Notebook using the PYNQ-Z2 board to control its hardware.

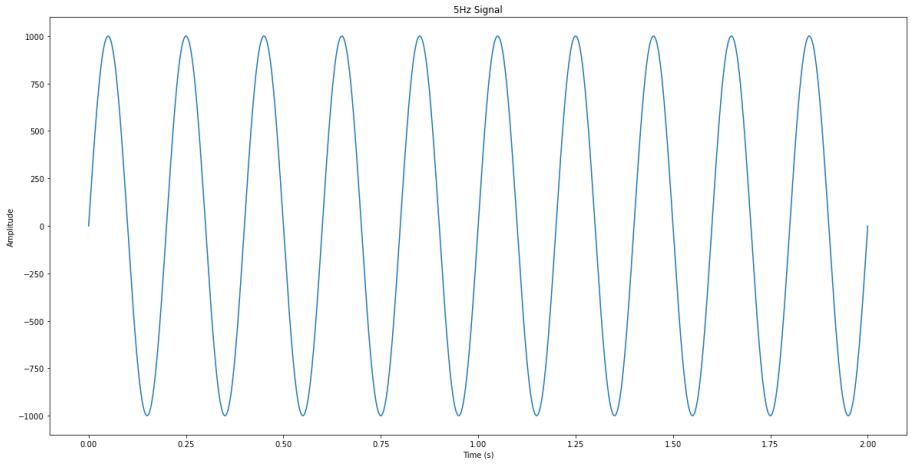
Acknowledgment:

I would like to acknowledge San Jose State University, specifically Professor Pham as the basis of the code was provided in the class

Audio Signal Processing:

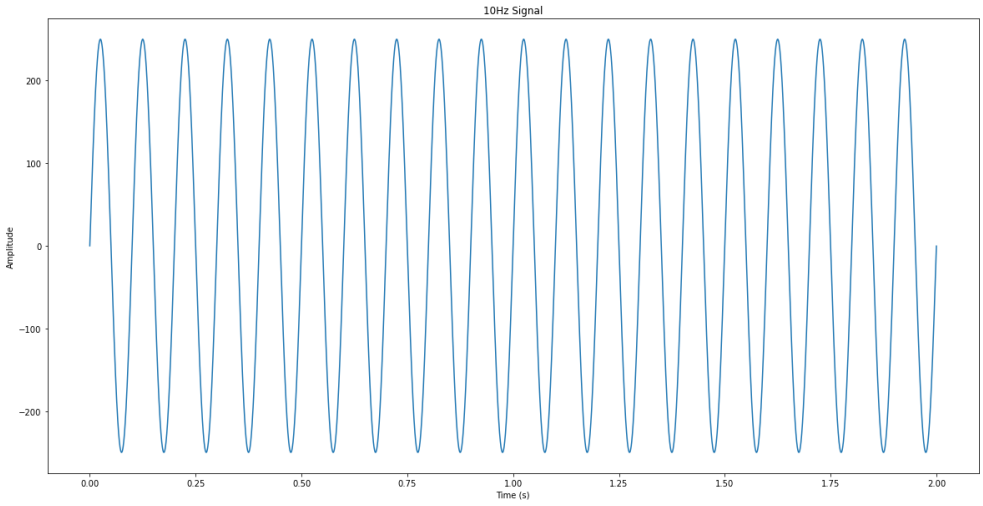
To execute the audio signal processing, the code starts off by creating 3 different frequencies and adding them together. This can be seen in the codes below as it creates a multitone with different frequencies waveform. To begin the proper imports are shown needed to run the code. All frequencies will have a time step of 0.00002. The first frequency is of 5Hz.

```
 8  import numpy as np
 9  from scipy import fftpack
10  from matplotlib import pyplot as plt
11  from scipy.io import wavfile
12
13
14  time_step=0.02
15
16
17
18  freq1=5 #5Hz
19  period1 = 1/freq1
20  time_vec = np.arange(0, 2, time_step)
21
22  sig1 = 1000*(np.sin(2 * np.pi / period1 * time_vec))
23  plt.figure(figsize=(20,10))
24  plt.title('5Hz Signal')
25  plt.ylabel('Amplitude')
26  plt.xlabel('Time (s)')
27  plt.plot(time_vec, sig1)
```



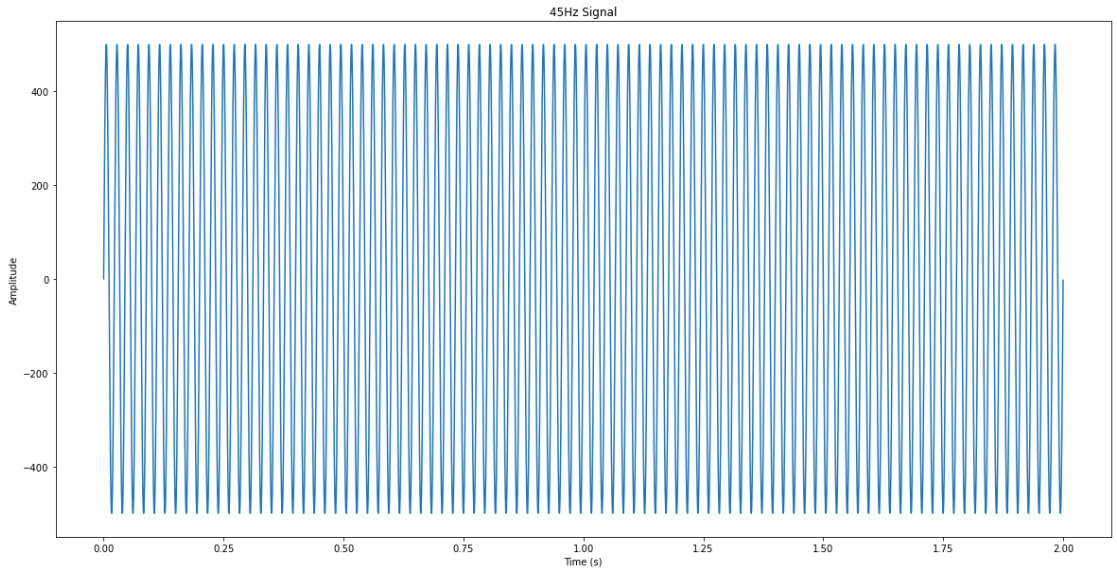
The second frequency is of 10Hz. The period is defined to then be able to used in the signal equation of amplitude × sin($\frac{2\pi}{period}$ × time). The sine function is then plotted.

```
freq2=10 #10Hz
period2 = 1/freq2
time_vec = np.arange(0, 2, time_step)

sig2 = 250*(np.sin(2 * np.pi / period2 * time_vec))
plt.figure(figsize=(20,10))
plt.title('10Hz Signal')
plt.ylabel('Amplitude')
plt.xlabel('Time (s)')
plt.plot(time_vec, sig2)
```



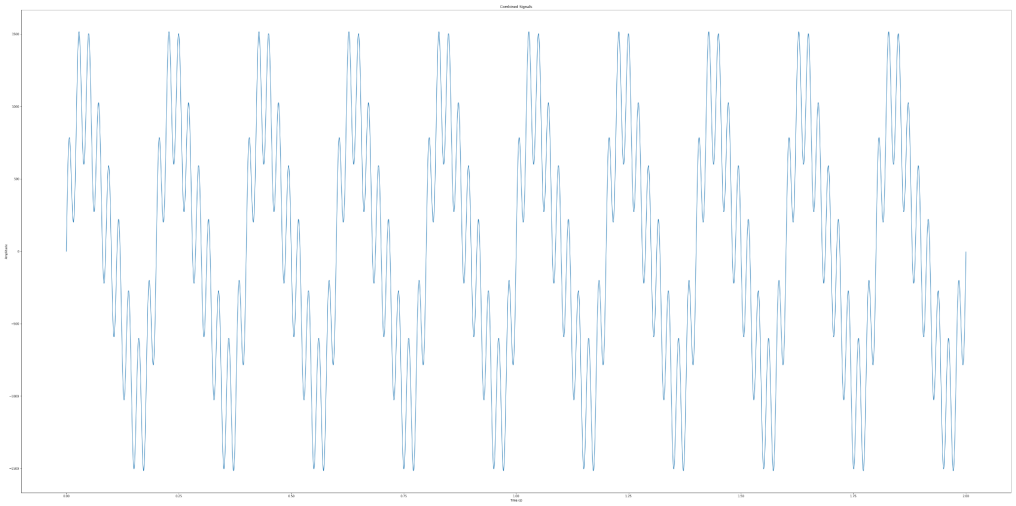
The last frequency being added is of 45Hz. Once more converted to a sine wave and plotted.

```
freq3=45 #100Hz
period3 = 1/freq3
time_vec = np.arange(0, 2, time_step)

sig3 = 500*(np.sin(2 * np.pi / period3 * time_vec))
plt.figure(figsize=(20,10))
plt.title('45Hz Signal')
plt.ylabel('Amplitude')
plt.xlabel('Time (s)')
plt.plot(time_vec, sig3)
```



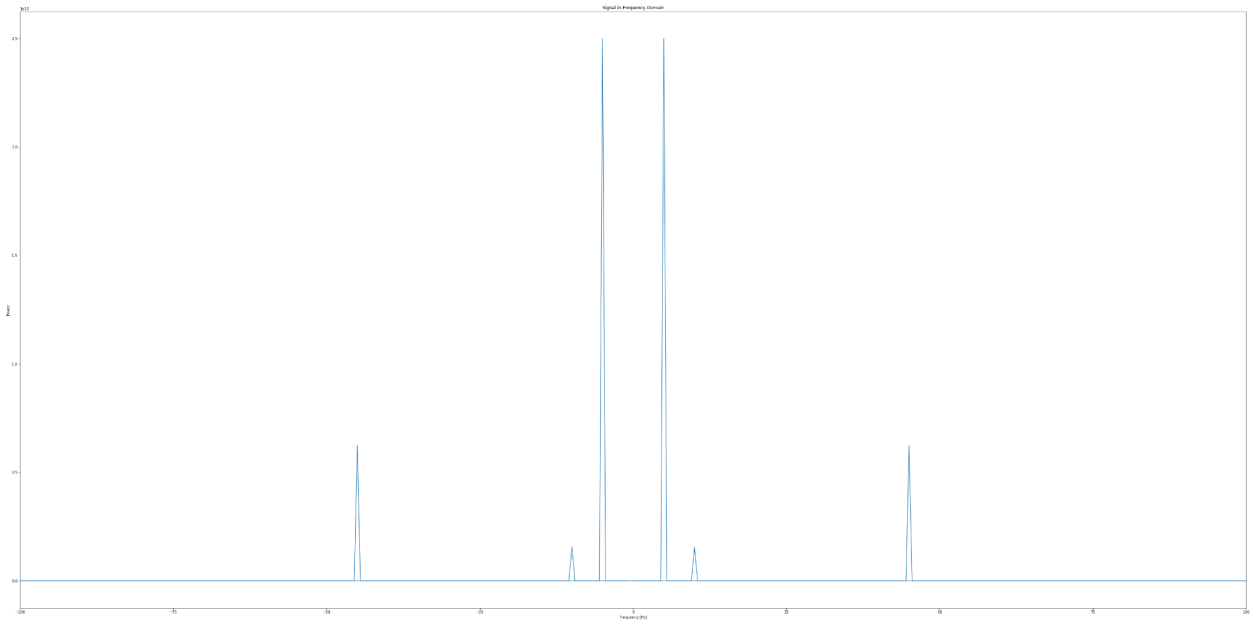
The signals are added below and plotted to in the time domain to see how they are combined. The signals are converted to a WAV file to listen to.

```
# Generate the signal
sig = sig1 + sig2 + sig3
plt.figure(figsize=(60,30))
plt.title('Combined Signals')
plt.ylabel('Amplitude')
plt.xlabel('Time (s)')
plt.plot(time_vec, sig)

# Create .WAV file of combined siganl
wavfile.write('CombinedSignal.wav', 44100, sig.astype(np.int16))
```



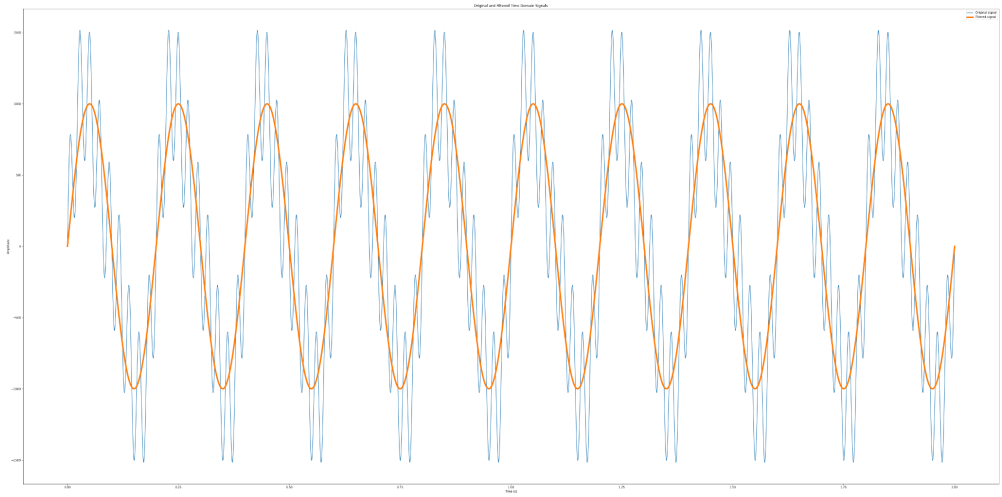
The power of the FFT is generated and plotted in the frequency domain.

```
65  # Compute the power
66  sig_fft = fftpack.fft(sig)
67  power = np.abs(sig_fft)**2
68  sample_freq = fftpack.fftfreq(sig.size, d=time_step)
69
70  # Plot the power
71  plt.figure(figsize=(60, 30))
72  plt.title('Signal in Frequency Domain')
73  plt.ylabel('Power')
74  plt.xlabel('Frequency [Hz]')
75  plt.plot(sample_freq, power)
```



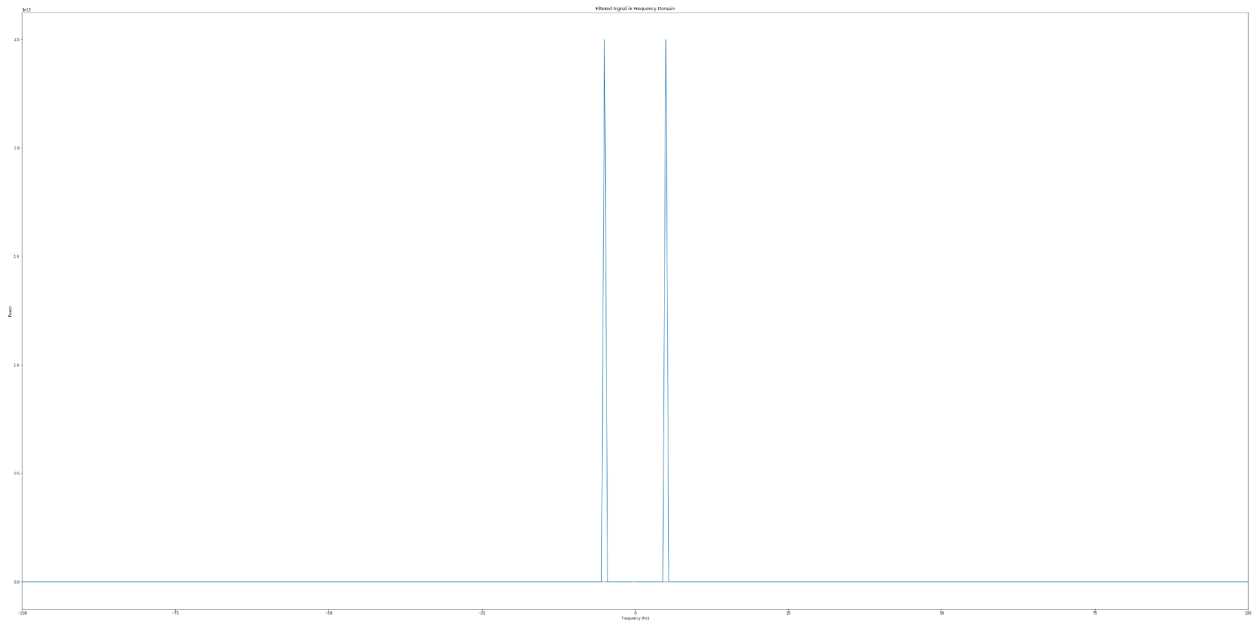
Next the peak frequency is found so that it can be the focus to then remove all the high frequencies. This signal is saved as a WAV file. The frequency is then transformed back to a signal and plotted with the original signal.

```
77  # Find the peak frequency
78  pos_mask = np.where(sample_freq > 0)
79  freqs = sample_freq[pos_mask]
80  peak_freq = freqs[power[pos_mask].argmax()]
81
82  # Remove all high frequencies
83  high_freq_fft = sig_fft.copy()
84  high_freq_fft[np.abs(sample_freq) > peak_freq] = 0
85  filtered_sig = fftpack.ifft(high_freq_fft)
86
87  # Create .WAV file of combined siganl without high frequencies
88  wavfile.write('CombinedSignalNoHighFreq.wav', 44100, filtered_sig.astype(np.int16))
89
90  # PLot without high frequencies
91  plt.figure(figsize=(60,30))
92  plt.title('Original and Filtered Time Domain Signals')
93  plt.plot(time_vec, sig, label='Original signal')
94  plt.plot(time_vec, filtered_sig, linewidth=5, label='Filtered signal')
95  plt.xlabel('Time (s)')
96  plt.ylabel('Amplitude')
97  plt.legend()
```



Lasty the power is computed once more and plotted to visualize the difference.

```
 99  # Compute the new power
100  sig_fft1 = fftpack.fft(filtered_sig)
101  power = np.abs(sig_fft1)**2
102  sample_freq = fftpack.fftfreq(filtered_sig.size, d=time_step)
103
104  # Plot new power
105  plt.figure(figsize=(60, 30))
106  plt.title('Filtered Signal in Frequency Domain')
107  plt.ylabel('Power')
108  plt.xlabel('Frequency (Hz)')
109  plt.plot(sample_freq, power)
```

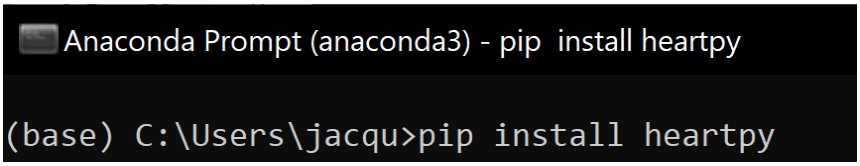


In the plot above the two peaks that were once at +/- 10 and +/- 45 have now been removed. The WAV files that are now created to in the directory of this code can be listened to and compared to get a second understanding of the change a part from the visuals. The first WAV file is barely audible as it is already a low frequency to begin with. Removing the high frequencies in this set creates a WAV file is is no longer audible.

Heart Rate Analysis:

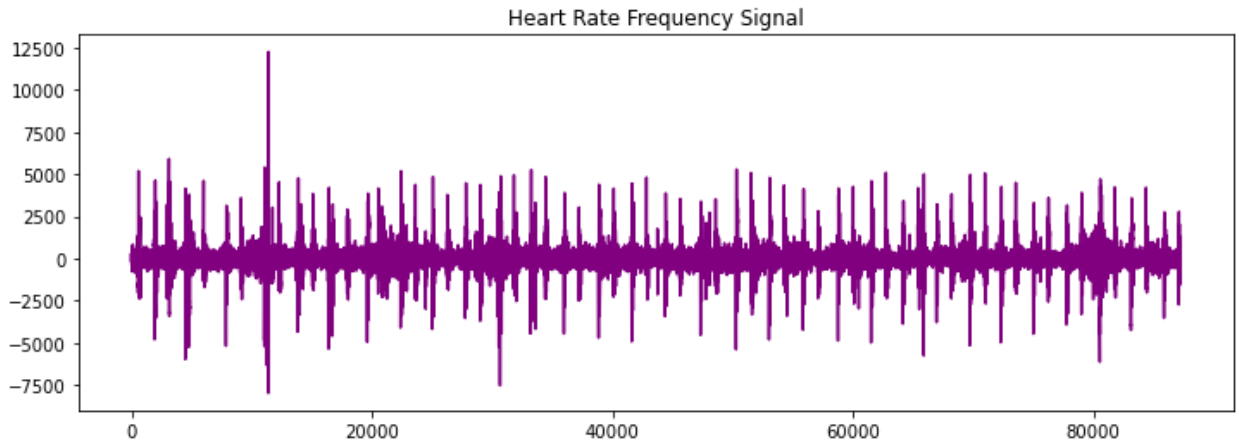
https://www.kaggle.com/datasets/kinguistics/heartbeat-sounds

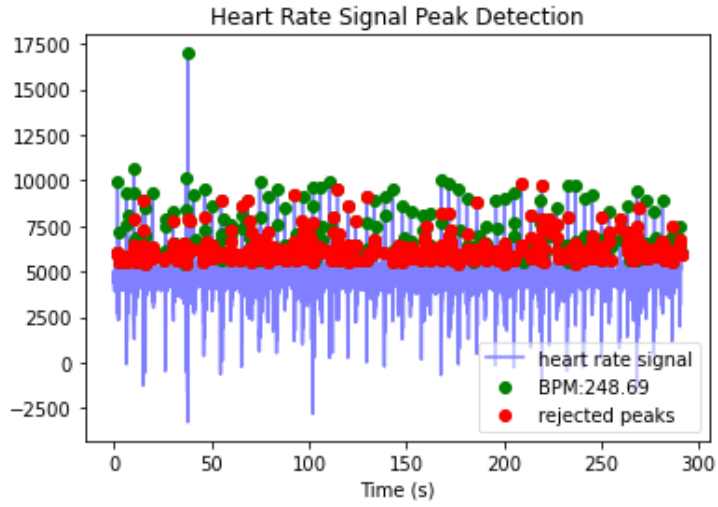
The link above is used to obtain a heartbeat WAV file with a minimum of 30 beats in the sample for this project. The file is converted using a WAV to CSV converter. Within the CSV file, the data needs to be a numerical one-dimensional array in order for the code to work properly. Prior to running the code, make sure to pip install the following.



Both the CSV and code need to be in the same directory in order to run properly. The code below was discussed in class and modified. The sample rate is changed to 300 to be over twice the max frequency. The CSV file name was replaced to match the one that was created. The trend line for the heart rate frequency signal is changed to purple and a title is added.

```
# Import Packages
import heartpy as hp
import matplotlib.pyplot as plt

sample_rate = 300

data=hp.get_data('108_Output_mono.csv')
plt.figure(figsize=(12,4))
plt.plot(data, color='purple')
plt.title("Heart Rate Frequency Signal")
plt.show

# Run Analysis
wd, m = hp.process(data, sample_rate)

# Visualise in plot of custom size
plt.figure(figsize=(3,4))
hp.plotter(wd, m)

#Display Computed Measures
for measure in m.keys():
    print('%s: %f' %(measure, m[measure]))
```






```
bpm: 248.689852
ibi: 241.264368
sdnn: 121.086758
sdsd: 105.711051
rmssd: 197.716126
pnn20: 0.937500
pnn50: 0.812500
hr_mad: 100.000000
sd1: 130.862192
sd2: 109.917504
s: 45188.811491
sd1/sd2: 1.190549
breathingrate: 0.214669
```

Red Alert:

The base code given in class for the Red Alert game is leveraged in three different ways, A need for speed, Try again, and Shuffling.

Starting with a need for speed, this alteration changes the speed at which the stars fall creating a nonuniform row of stars. The code below assigns the randint() function to pick a speed between 0, 1, or 2. When adding this to the duration line, it assigns a random speed to each star that is falling.

```
111 def animate_stars(stars_to_animate):
112     #pass
113     for star in stars_to_animate:
114         random_speed_adjustment = random.randint(0,2)
115         duration = START_SPEED - current_level + random_speed_adjustment
116         star.anchor = ("center", "bottom")
117         animation = animate(star, duration=duration, on_finished=handle_game_over, y=HEIGHT)
118         animations.append(animation)
```

The second alteration is to try again. This change allows the user to play again without running the code again. An if statement is added to the update portion of the code. This checks for either game over or game complete conditions along with if the space key is pressed. Upon conditions being met, the game will reset the game variables, clear the stars, and return the level to level 1.

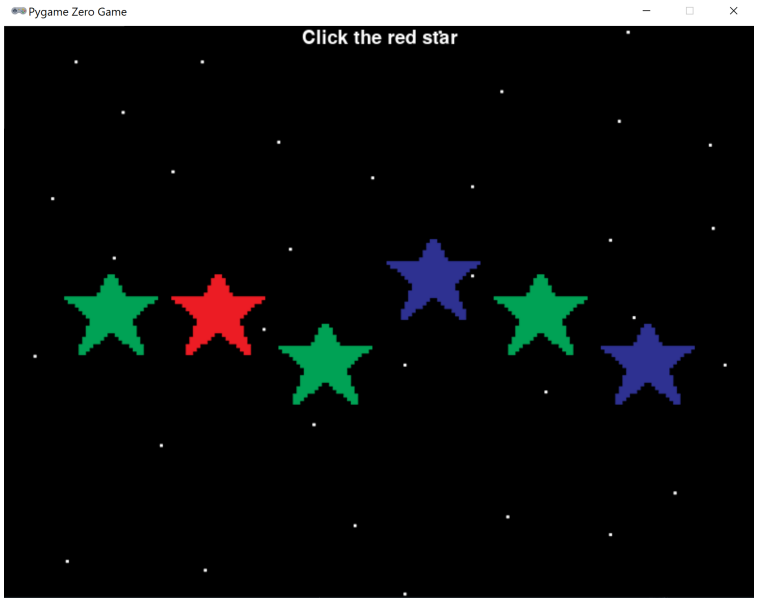
```
63  def update():
64      global stars
65      global game_over
66      global game_complete
67      global current_level
68      global keyboard
69
70      if len(stars) == 0:
71          stars = make_stars(current_level)
72
73      if (game_complete or game_over) and keyboard.space:
74          stars = []
75          game_over = False
76          game_complete = False
77          current_level = 1
```

The last alteration is Shuffling. In this change, the stars shuffle positions in an attempt to confuse the users. This is done by adding a shuffle function. An if statement allows the shuffle any time the stars are on screen. The shuffle happens for a duration of every 0.5 seconds. The code below was added to the initial base to execute shuffling.

```
37  def shuffle():
38      global stars
39      if stars:
40          x_values = [star.x for star in stars]
41          random.shuffle(x_values)
42          for index, star in enumerate(stars):
43              new_x = x_values[index]
44              animation = animate(star, duration=0.5, x=new_x)
45              animations.append(animation)
46
47  clock.schedule_interval(shuffle, 1)
```

The final end game will look as followed with the need for speed, Try again in the end game and game completed, and Shuffling respectively.

A Need For Speed: Stars falling at different rates



Try Again: Text Prompting the user to press the “space” key to play again

Shuffling: Stars are displayed overlapping because they are switching spots