Arun Suresh

Homework 4

---

1. The Haar function is defined as $H(x) = \begin{cases} 0 & x < 0 \\ 1 & 0 \leq x < \frac{1}{2} \\ -1 & \frac{1}{2} \leq x < 1 \\ 0 & 1 < x \end{cases}$

Plotted below is the Haar function and a time-shifted version (with a timeshift of -2)

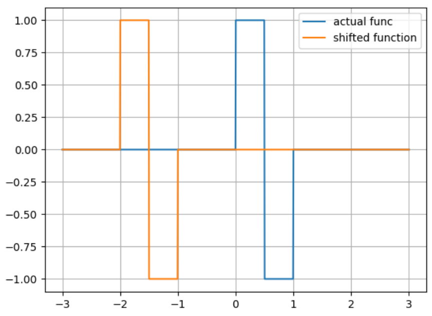


Figure 1: Plots of original and timeshifted Haar function

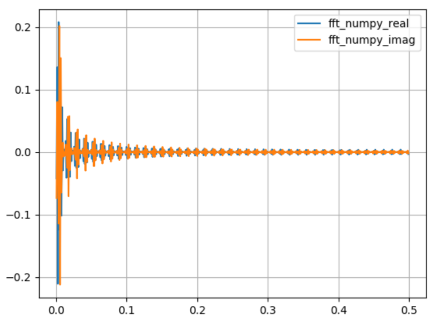
The fourier transform of the Haar function was calculated in Python (ver 3.70) using two different numerical packages. The first one was the FFT functionalities that come included with the numpy library for python. The second one was the pyFFTW, which is the python wrapping of regular FFTW method written for C++. A mask was applied to filter out all the negative frequencies that are symmetric to the positive frequency side (This comes as a part of both FFTW and Numpy.fft, as a normalization method so that the inverse fourier transform could be computed with relative ease)
The following are the plots of the fourier transform using the numpy fft functionalities.

The following are the plots of the fourier transform using the pyFFTW library.

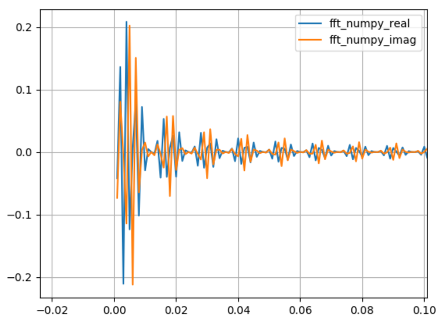
The y-axis for the above plots, and for all plots that follow measures the amplitude of each constituent wave, while the x-axis measures the scaled[1] frequency of the constituent wave.

It is expected that we notice numerous frequencies dying out as N gets larger, because the Haar function is a discontinuous function and thus does not admit a finite fourier series.

[1]actual frequency can be obtained by computing $(x - value) * \frac{N}{6}$ where $N = 1000$ is the number of frequency samples and 6 is the length of the axis $(-3, 3)$

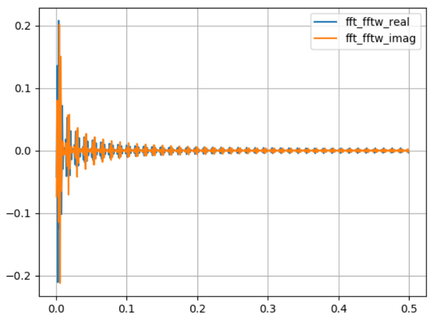


(a) full FT

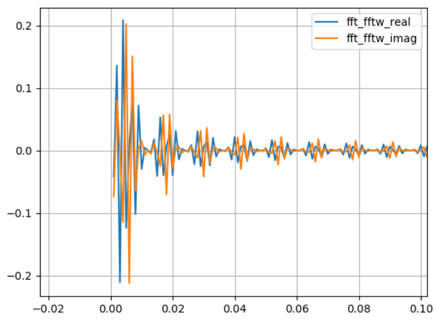


(b) FT zoomed

Figure 2: Numpy FFT



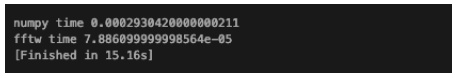
(a) full FT



(b) FT zoomed

Figure 3: FFTW

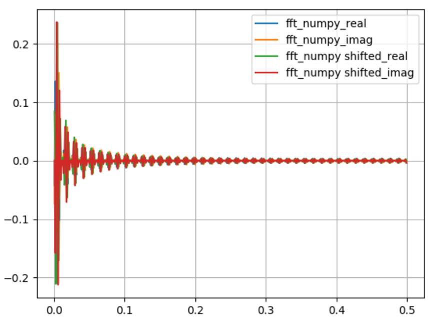
It is more or less clear that both numpy FFT and FFTW both return very similar fourier transforms. However, the main difference between using Numpy FFT and FFTW is that of computational time. While the numpy FFT took $0.003s$ to complete, FFTW took $7.886\times10^{-5}s$ to finish, which is considerably faster than numpyFFT, considering that this was performed only over a thousand data points. Presented below is a screenshot of the compilation time printed to the console.



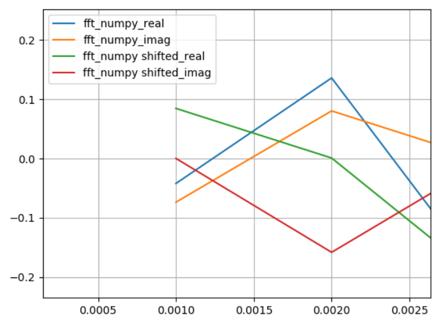
```
numpy time 0.0002930420000000211
fftw time 7.886099999998564e-05
[Finished in 15.16s]
```

Figure 4: Computation time

The fourier transforms of the timeshifted Haar function were also computed and plotted alongside the fourier transforms of the non-timeshifted Haar function. The plots are presented below.



(a) full FT



(b) FT zoomed

Figure 5: Numpy FFT zero vs. non-zero timeshift

Initially looking at the non-zoomed fourier transform, it looks like a cacophonous mess of plots. So, we zoom into just one of the spikes to analyze what is really happening. In figure 5(b), it is abundantly clear what is happening. Notice that a timeshift to the space domain, produces a complex phase factor in the frequency domain - this means that the original solution is just rotated by some angle in the complex plane. This in turn means, that the real and imaginary values of the output oscillate in such a way that the magnitude is preserved (as the magnitude of any phase factor is 1).
Looking at plot 5(b), we can observe this happening. Notice that the real and imaginary part of the non-shifted FT are different from the ones of the shifted ones. However, both of their magnitudes are the same $\sqrt{0.08^2 + 0.136^2} \approx \sqrt{0^2 + 0.157^2} \approx 0.157$. This fact is even more clear when we plot out the absolute value (which coincides for timeshift and non-timeshift version of the function) of our complex output instead of real and imaginary parts seperately. Presented below is that plot.

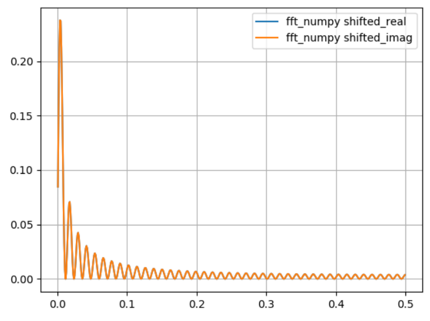


Figure 6: Magnitudes

Results from FFTW and Numpy.fft once again coincided just like before. The above plots are only those obtained from numpy.fft and the FFTW plots are not provided to reduce redundancy. The time difference in computation was once again significant, with FFTW performing much better than numpy.fft

2. The Sine cardinal function is given by $F(k) = \begin{cases} 1 & k = 0 \\ \frac{\sin(k)}{k} & otherwise \end{cases}$ Plotted below is the Sine cardinal function and a time-scaled version of it (by a factor of 2). Notice that in this case, the function is already given in the frequency domain, and we are required to calculate the inverse fourier transform.

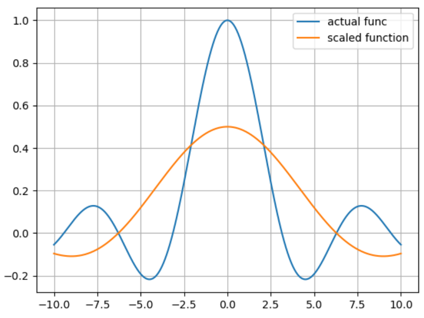


Figure 7: Sine cardinal function in the Frequency domain

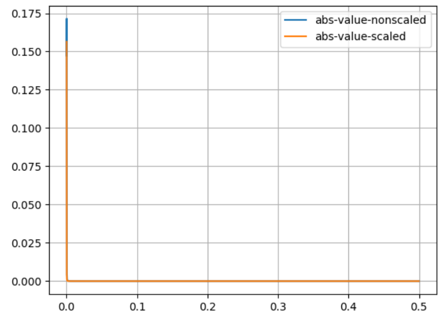
The x-axis represents frequency and the y-axis represents the amplitudes. The inverse fourier transform was calculated using the same two packages used in the previous problem. A mask was once again applied to filter out all the negative space values (which would be symmetric) The corresponding IFT plots for the non-scaled and time-scaled functions are presented in Figures 8 and 9. The absolute value of the output was calculated for the plots (this was done to avoid a few numerical errors that might have been occurred)[2]

It is apparent that both numpy.fft and FFTW returned very similar looking plots (although, it seems as it FFTW had a little bit of a better smoothing than numpy.fft and we had to zoom in considerably to notice the linear numerical approximation of the curve). The time difference in computation was once again significant with FFTW performing far better than numpy.fft - by a factor of 100. Presented in Figure 10 is the time data that was output to the console.[3] The fftw performed to its fullest efficiency during its second computation compared to the first, because it is possible that a significant amount of time was spent in loading up the module during its first computation.

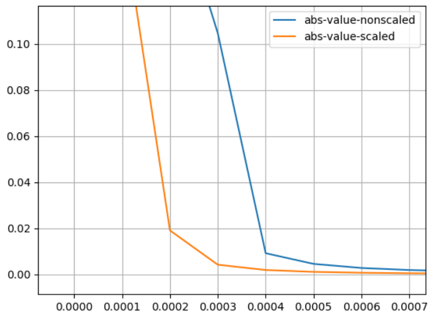
The time scaling is also very apprent in the zoomed version of the plots in Figure 8 and 9. We notice that the scaled function is shrunk by a factor of 2, which is what we would expect from a timescaling by a factor of 2.

[2]Since the sine cardinal function is an even function, we expect the imaginary part of the output to be 0 always. However, upon plotting the real and imaginary parts, some noise was observed in both outputs. In order to rectify that error completely, the absolute value was computed

[3]due to some Latex issue, I was not able to typeset the images to their proper spots

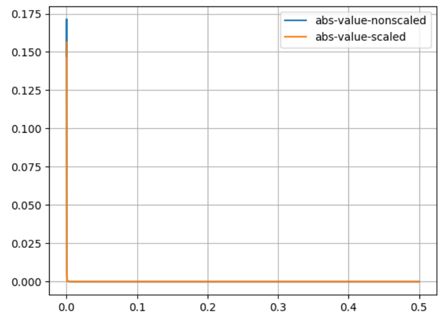


(a) full FT

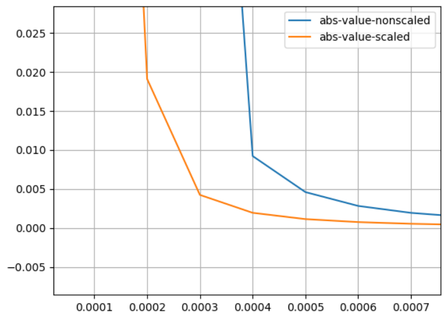


(b) FT zoomed

Figure 8: Numpy FFT

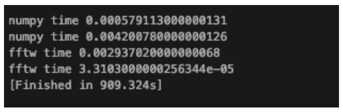


(a) full FT



(b) FT zoomed

Figure 9: FFTW



```
numpy time 0.000579113000000131
numpy time 0.004200780000000126
fftw time 0.002937020000000068
fftw time 3.3103000000256344e-05
[Finished in 909.324s]
```

Figure 10: Time differences in computation