Fourier Transforms With Wave Equations in Quantum Mechanics

Lauren Farmer[1], Jonah Shoemaker[1], Dakota King[1]

[1]*Arizona State University, Tempe, AZ, USA, email jcshoema@asu.edu*

Background

In quantum mechanics, it is common to need to take a particle's wave function in a certain basis, such as the position basis, momentum basis, energy basis, time basis, etc, and transform it into a complementary basis. It is well known that every function can be expressed as a linear combination of the transcendental functions sin(x) and cos(x). Using Euler's formula, these transcendental functions can be expressed as a single exponential with a complex argument. Thus, it is possible to express any function as a series of complex exponentials. The Fourier transform is a linear transformation that takes a function expressed in a certain basis and "maps" it onto the basis's complementary basis, expressed as such a series of complex exponentials, using an overlap integral:

$$\phi(k) = \int_{-\infty}^{\infty} \psi(x) e^{ikx} dx \qquad \phi(k) = \int_{-\infty}^{\infty} \psi(x) e^{-2\pi ikx} dx$$

In fields such as x-ray crystallography and material physics, researchers often obtain experimental data in a momentum basis and transform the data into position space in order to learn about the physical structure of the object being examined. In other fields such as electrical engineering and spectroscopy, it is often required to transform between energy and time bases in order to encode and decode signals. Only a rare handful of these data sets can be expressed analytically, let alone consequently transformed analytically using the continuous integral definition of the FT, so researchers and engineers primarily evaluate these transformations numerically. The purpose of this study was to find the most efficient method of using Python to compute momentum-space Fourier transforms of one-dimensional position-space wave functions, one of the simpler types of transforms. Three methods were used: calculating the transforms by hand as a control, using the FFT function in Numpy, and building an algorithm to find the transforms. The FFT function in Numpy utilizes linear algebra to compute symmetric blocks of transform at a time in order to decrease computing time and increase accuracy, but it has many issues with ease of implementation for the average coder who might lack extensive experience with Python. This mildly steep learning curve presents a bit of an intimidation factor and no small degree of frustration to beginners. The algorithm designed in this study was built with a for loop, which calculated the FT integral with different discrete wave-number values in each iteration using a Riemann sum technique, and was designed for ease of use. Both of these methods find what is called the Discrete Fourier Transform, since they discretize the two complementary variables in question in order to evaluate the FT integral as a Riemann sum.

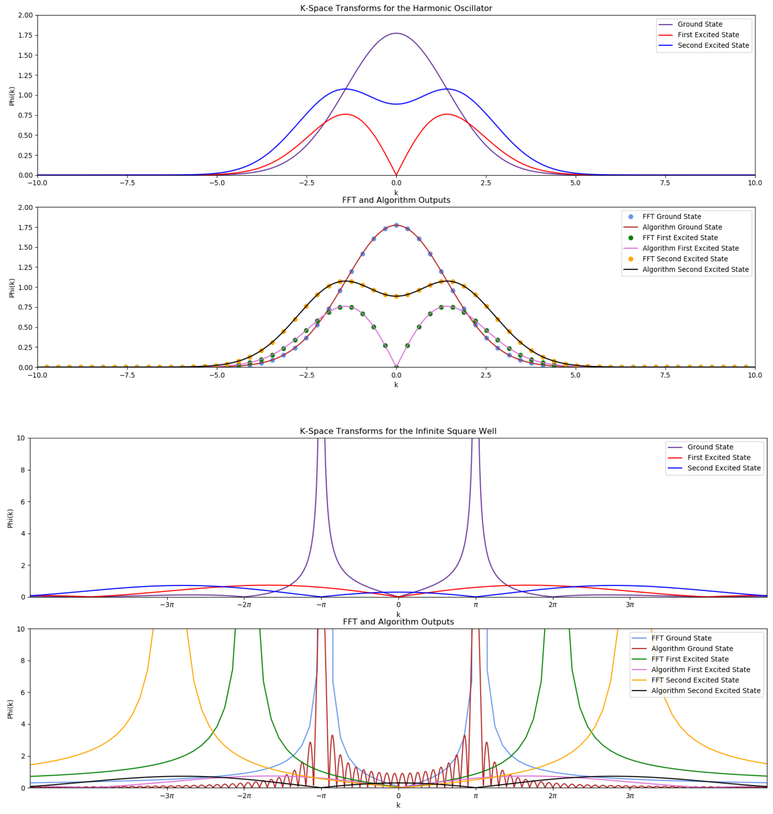
Methods

The analytical Fourier transforms were found using a combination of basic integral techniques, references to integral tables, and raw ingenuity on the part of the researchers. Questions on these analytical solutions can be expressed to the researchers for further delineation.

In order to implement the FFT function in Numpy, first an input "x" array was created using a given step size. This input array was plugged into the position space function to obtain an array of position space values. Then, the np.fft.fft() function was applied to this output array, after which the np.fft.fftshift() function was applied in order to shift the zero-frequency component from the beginning to the middle of the array – essentially, this can be interpreted as "putting the array in order", although this is not entirely accurate. However, in spite of its inherent mystery and intrigue, this step is quite crucial, as the graph of the unshifted output is quite nonsensical. Since these output values are often complex, the modulus of each value was taken for the purpose of graphing. Then, it was necessary to create a "k" array whose spread and step size were based on the step size originally used. In the last unintuitive step, since the FT definition used by the FFT function contains a 2π in the exponential while the traditional quantum mechanical definition does not, this k-array had to be multiplied by 2π in order to shift the "frequency" of the sinusoidal terms.

The home-made algorithm was designed with a for loop for a heart. First it populated an x-array and k-array of the same shape and spread as those of the FFT method, and created two empty arrays of the same shape as the k-array for the real and imaginary components of the output k-space values. Then, it ran a for loop, starting with the lowest value in the k-array, running the FT integration using a basic Riemann sum algorithm twice – once for the real component of the integral (cos(x)) and once for the imaginary (sin(x)) – and then storing each of these components into the two initially empty arrays. The loop was then repeated with the next value in the k-array, up until the last value of the k-array. After the loop ran its course, these results were added in the form x+iy, with x as the real output and y the imaginary, and the modulus of these resulting complex valued outputs was taken for the sake of graphing once again.

Results



The simple harmonic oscillator potential

$$V(x) = \frac{1}{2} m\omega^2 x^2$$

Stationary state solutions (ground and first two excited states).

$$\psi_0 = \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-mwx^2/(2\hbar)}; \quad \psi_1 = \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \sqrt{\frac{2m\omega}{\hbar}} x e^{-mwx^2/(2\hbar)}; \quad \psi_3 = \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \frac{1}{2\sqrt{2}} \left(4(\frac{m\omega}{\hbar})x^2 - 2\right) e^{-mwx^2/(2\hbar)}$$

For simplicity and the sake of just doing Fourier transforms, we set all of the constants equal to one,

$$u_0(x) = e^{-x^2}; \quad u_1(x) = x e^{-x^2}; \quad u_2(x) = (x^2 - 1) e^{-x^2}$$

Fourier transform into k-space

$$\phi_n(k) = \int_{-\infty}^{\infty} u_n(x) e^{-ikx} dx \Rightarrow \boxed{\phi_0(k) = \sqrt{\pi} e^{-k^2/4}; \quad \phi_1(k) = -\frac{1}{2} i\sqrt{\pi}(k) e^{-k^2/4}; \quad \phi_2(k) = -\frac{1}{4}\sqrt{\pi}(k^2 + 2) e^{-k^2/4}}$$

The infinite square well potential

$$V(x) = 0 \quad 0 < x < L$$
$$V(x) = \infty \quad elsewhere$$

Stationary state solutions (ground and first two excited states).

$$\psi_1(x) = \sqrt{2}\sin(\pi x); \quad \psi_2(x) = \sqrt{2}\sin(2\pi x); \quad \psi_3(x) = \sqrt{2}\sin(3\pi x)$$

Fourier transform into k-space

$$\phi_n(k) = \int_0^1 \psi_n(x) e^{-ikx} dx \Rightarrow \boxed{\phi_1(k) = -\frac{\sqrt{2}\pi(1 - e^{-ik})}{k^2 - \pi^2}; \quad \phi_2(k) = -\frac{\sqrt{2}\pi(2 - 2e^{-ik})}{k^2 - 4\pi^2}; \quad \phi_3(k) = -\frac{3\sqrt{2}\pi(1 + e^{-ik})}{k^2 - 9\pi^2}}$$

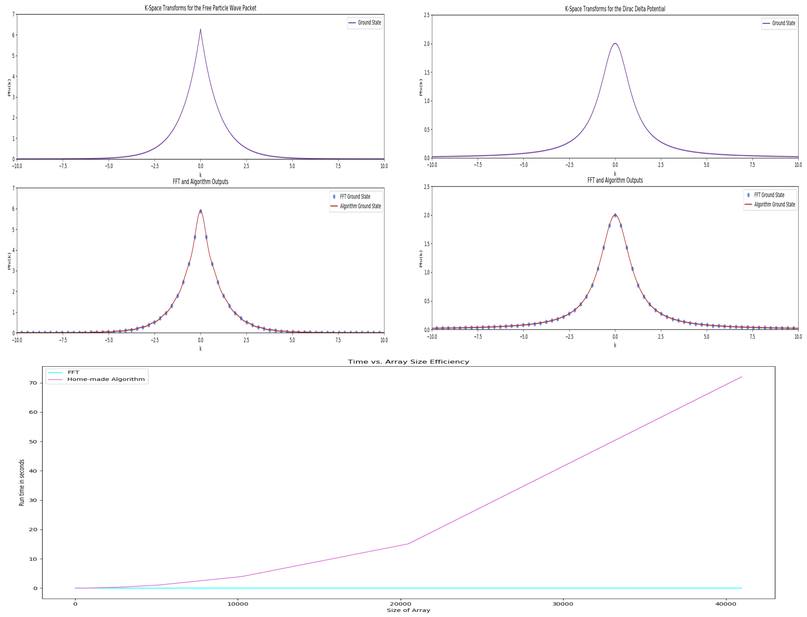
Free particle

Initial wavefunction

$$u(x, t = 0) = \frac{1}{x^2 + 1}$$

Fourier transform into k-space

$$\phi(k) = \int_{-\infty}^{\infty} \psi(x) e^{ikx} dx \Rightarrow \boxed{\phi(k) = 2\pi e^{-|x|}}$$



Results of Median Absolute Error Test for Ground State Harmonic Oscillator:

MAE(FFT) = 0.003415 MAE(Alg) = 1.7098 e-7

Delta potential

$$V(x) = -\alpha\delta(x)$$

The stationary state solution (One bound state)

$$\psi(x) = \frac{\sqrt{m\alpha}}{\hbar^2} e^{-m\alpha|x|/\hbar^2}$$

Again, for simplicity and just caring about the Fourier transform, we set the constants equal to one

$$u(x) = e^{-|x|}$$

Fourier transform into k-space

$$\phi_n(k) = \int_{-\infty}^{\infty} u(x) e^{-ikx} dx \Rightarrow \boxed{\phi(k) = \frac{2}{k^2 + 1}}$$

Summary

While the home-made algorithm displayed far less run-time efficiency than the FFT function as the array size increased, it was designed to be easier to implement by the user for quantum transform applications. In addition, the self-designed algorithm was incapable of evaluating complex-valued wave functions, but more versatile at handling real-valued functions without any additional modifications required. In fact, the FFT function failed admirably at evaluating the first and second excited states of the infinite square well, showing asymptotic behavior at the zeros of the denominator of the analytic solutions, even though those zeros were actually removable singularities and therefore should have caused no asymptotic behavior at those zeros. Neither the control nor the home-made algorithm displayed these anomalous divergences. It is possible that the sinusoidal functions being transformed caused issues due to the difference in the FT definition used by the FFT. Another issue encountered with the FFT was in getting it to produce more tightly spaced outputs in the desired range of k-values – it was only possible to limit the range, but only at the expense of decreasing the number of outputs overall in order to do so. In addition, it was observed that when the array size was increased past a certain size, the FFT was incapable of handling the operation due to the lack of memory in the computer. This is possibly due to the size of the square matrix blocks that the FFT evaluates increasing to a point that the computer was incapable of handling the volume of instant element-wise calculations. The self-made algorithm displayed no such shortcomings due to the single calculation progression it was designed to implement – it simply took exponentially longer amounts of time to find the transform as the array size increased, but always inexorably progressed to the end. It is the glacier of algorithms, slowly but surely eating up the land of calculations until only Fourier ice remains.

Acknowledgments

The researchers would like to thank Kellie Gillespie, Elizabeth Rodriguez, Jineane Ford, and Carroll Farmer for their continued love and support in the researchers' academic careers.

In addition, the researchers would like to acknowledge the community of Stack Overflow for their magnanimous support.

Some code was adapted from the Matlab website for use on this project. Scikit-learn.org was referenced for the MAE test.