A common task in the solution of physics problems is the spectral decomposition of functions. A very common application is in the solution of partial differential equations, as we will see later. This is also important in signal analysis and data analysis generally. There are many possible decompositions of functions into different basis sets; as I have noted before, function space is a vector space and there are many choices of a complete basis set of functions (orthogonal and not). An extremely common one is the Fourier basis, and here we will learn about the practical implementation of discrete Fourier transforms, using the Fast Fourier Transform.

These notes draw heavily on *Numerical Recipes*, a valuable resource for entry-level understanding of numerical methods relevant to physics.

1. Fourier Transforms

We use the common definition of the Fourier tranform as follows:

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} \mathrm{d}t h(t) \exp\left(2\pi i f t\right) \\ h(t) &= \int_{-\infty}^{\infty} \mathrm{d}f H(f) \exp\left(-2\pi i f t\right) \end{aligned} \tag{1}$$

The notation here suggests t is time, and f is frequency. However, of course Fourier transforms can be performed in terms of spatial coordinates (or anything really).

What property of $H(f)$ will lead to a real $h(t)$?

If $H(-f) = [H(f)]^*$, because in that case:

$$\begin{aligned} h(t) &= \int_{-\infty}^{\infty} \mathrm{d}f H(f) \exp\left(-2\pi i f t\right) \\ &= \int_{-\infty}^{\infty} \mathrm{d}f H_r(f) \exp\left(-2\pi i f t\right) + i \int_{-\infty}^{\infty} \mathrm{d}f H_i(f) \exp\left(-2\pi i f t\right) \\ &= \int_{-\infty}^{\infty} \mathrm{d}f H_r(f) \cos\left(-2\pi f t\right) + i \int_{-\infty}^{\infty} \mathrm{d}f H_r(f) \sin\left(-2\pi f t\right) \\ &\quad + i \int_{-\infty}^{\infty} \mathrm{d}f H_i(f) \cos\left(-2\pi i f t\right) - \int_{-\infty}^{\infty} \mathrm{d}f H_i(f) \sin\left(-2\pi i f t\right) \end{aligned} \tag{2}$$

Both imaginary terms have odd integrands, so the result is real.

Clearly if $H(-f) = -[H(f)]^*$, then $h(t)$ is imaginary.

These properties of the Fourier transform become time-saving devices when considering the numerical transforms of functions with known symmetries.

A particularly useful property of the Fourier transform is the convolution rule:

$$\mathrm{FT}(f * g) = F \times G \tag{3}$$

This leads to fast implementations of convolutions.

Finally, it is worth noting that the total power is the same in the Fourier or configuration basis:

$$\int_{-\infty}^{\infty} \mathrm{d}t\, |h(t)|^2 = \int_{-\infty}^{\infty} \mathrm{d}f\, |H(f)|^2 \tag{4}$$

This result is known as Parseval's Theorem.

2. Discrete Fourier Tranform

Obviously on a computation one does not implement the Fourier transform continuously. Instead, what you do is that you have a set of samples $h(t_k)$ in equally spaced locations $t_k = k\Delta$, where k runs from 0 to $N-1$. For reasons we will soon see, very often the N are factors of two. The discrete Fourier transform is a linear transform of these samples similar to the Fourier transform.

How many independent frequencies can you determine for the discrete Fourier transform?

Since you are restricted to linear combinations, you are restricted to an N-dimensional linear space, and there can only be N independently determined freqencies.

What are these frequencies?

The lowest representable frequency is $1/N\Delta$, since this is length of the interval. The discrete Fourier transform then is to the frequencies:

$$f_n = \frac{n}{N\Delta} \tag{5}$$

for n between $-N/2$ and $N/2$ (with $f_{N/2} = f_{-N/2}$).

The highest representable frequency is $1/2\Delta$, which is known as the *Nyquist frequency*.

Then the discrete tranform is:

$$\begin{aligned} H(f_n) &= \int_{-\infty}^{\infty} \mathrm{d}t h(t) \exp(2\pi i f_n t) \\ &\approx \sum_{k=0}^{N-1} h(t_k) \exp(2\pi i f_n t_k)\Delta \\ &\approx \Delta \sum_{k=0}^{N-1} h(t_k) \exp(2\pi i (n/N\Delta)(k\Delta) \\ &\approx \Delta \sum_{k=0}^{N-1} h(t_k) \exp(2\pi i k n/N) \end{aligned} \tag{6}$$

Usually we define $H_n = H(f_n)/\Delta$, because in that case we can work with arrays without reference to an underlying scale.

What is the periodicity of H_n?

The lowest non-zero k is 1, and for this choice the exponential has $2\pi in/N$, which loops over 2π in N steps, so the periodicity of that term is N. All of the other k terms have a harmonic of that periodicity. Thus, as I asserted above $H_{-N/2} = H_{N/2}$.

Also, we can then equally well let n vary from 0 to $N-1$, which makes most sense on a computer.

The inversion of H_n is:

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n \exp(-2\pi ikn/N) \tag{7}$$

3. Calculation of a Fast Fourier Transform

How many operations does it take to calculate H_n if you naively implement the above formulae?

The sums to calculate H_n require a sum over N data points, N times, so the expense up to some prefactor α is N^2, or αN^2. There are many applications in which you might want to have N in the millions. This quickly becomes intractable. Luckily this is not necessary.

A far better way has long been known, called the *Fast Fourier Transform*. It rewrites the sum in a clever tree-based way that reduces the order of operations to $N \log_2 N$.

It works due to the *Danielson-Lanczos lemma*, which is easily shown as follows to allow us to :

$$\begin{aligned}
H_k &= \sum_{j=0}^{N-1} \exp(2\pi ijk/N) h_j \\
&= \sum_{j=0}^{N/2-1} \exp(2\pi i(2j)k/N) h_{2j} \sum_{j=0}^{N/2-1} \exp(2\pi i(2j+1)k/N) h_{2j+1} \\
&= \sum_{j=0}^{N/2-1} \exp(2\pi ijk/(N/2)) h_{2j} + \exp(2\pi ik/N) \sum_{j=0}^{N/2-1} \exp(2\pi ijk/(N/2)) h_{2j+1} \\
&= H_k^e + W^k H_k^o
\end{aligned} \tag{8}$$

where H_k^e is the Fourier transform of the even components, H_k^o is the Fourier transform of the odd components, and $W = \exp(2\pi i/N)$. Note that each H_k^e has only $N/2$ unique values, since it is a Fourier transform of length $N/2$; as you loop over k, it just repeats itself.

Imagine you did this exactly once, where the smaller Fourier transforms you did naively. You would do *two* Fourier transforms, each of expense $\alpha(N/2)^2$. This would produce H_k^e and H_k^o (both periodic with length $(N/2)$). Then you put these together by looping over all k. The total expense

is $N + 2 \times \alpha(N/2)^2 = N + \alpha N^2/2$, or half of the original expense we quoted above for large N. Note though that we keep the leading linear term for reasons that soon will become clear.

But why do the smaller Fourier transforms naïvely? You should do them using this same clever method. In this case:

$$H_k^e = H_k^{ee} + W^{2k} H_k^{eo} \tag{9}$$

where *ee* means "even parts of the even parts" and *eo* means "odd parts of the even parts." Note very importantly that $2k$ appears because each Fourier transform is over a $N/2$-sized array; this must be the case because H_k^e is periodic over $N/2$.

Then the expense for each of those is not $(N/2)^2$ but $N/2 + 2 \times \alpha(N/4)^2$. Then the total is $2N + 4 \times \alpha(N/4)^2 = 2N + \alpha N^2/4$, even smaller. Repeat it again, and you find a cost of $3N + \alpha N^2/8$. If you do this $\log_2 N$ times, you get down to $N = 1$ and the first term is $N \log_2 N$ and the second term goes to $\alpha N^2/N = \alpha N$. So it is the first term that dominates.

So you can take any Fourier transform it, and break it in two. Then you can take each piece, and break that in two, and so on. Until you have a one-element Fourier transform to perform. A one element discrete Fourier Transform is just equal to that same element. So actual α is 1, or just the cost of figuring out which element you have narrowed down to.

If you label the Fourier transform at each leaf as something like $H_k^{eeoeoe\cdots}$, then it corresponds to some h_j (which is the same for all values of k). Which h_j? Each time you sort into even and odd, you are taking the lowest bit of significance, and reordering the whole array according to that. Then you take the next bit, and reorder according to that. This means that you are taking the index k, reversing its bits, and identifying k with its bit-reversal corresponding to j.

To take the FFT, you can then just reorder h_j appropriately, and then apply the Danielson-Lanczos lemma in a tree-like fashion, using the fact that each stage each subtransform is periodic with period 2, 4, ..., $N/2$ as you go up in levels. This leads to $\sim N$ operations executed $\log_2 N$ times. It is a book-keeping nightmare (just look at the code in *Numerical Recipes* for example) but it is worth it.

Further tricks can be applied to reduce the number of cos() and sin() calls.

The `numpy.fft` module is an implementation of these techniques.

4. Real FFTs

A common application is the FFT of a function $f(t)$ that is known to be real in configuration space. It seems intuitive that we should be able to do this more quickly, because we know all the imaginary parts are zero, and yes we can.

You can do so by defining:

$$h_j = f_{2j} + if_{2j+1} \tag{10}$$

If you Fourier tranform this, you get back (since the Fourier transform is linear):

$$H_n = F_n^e + iF_n^o \tag{11}$$

We also know from above that the Fourier tranform we want can be expressed as:

$$F_n = F_n^e + \exp(2\pi in/N)F_n^o \tag{12}$$

We can then note:

$$\begin{aligned} F^e &= \frac{1}{2}\left(H + H^*\right) \\ F^o &= \frac{1}{2}\left(H - H^*\right) \end{aligned} \tag{13}$$

and so

$$F_n = \frac{1}{2}\left(H_n + H^*_{N/2-n}\right) + \frac{1}{2}\exp(2\pi in/N)\left(H_n - H^*_{N/2-n}\right) \tag{14}$$

5. Sine and Cosine Transforms

Finally, you can perform sine and cosine transforms of real function efficiently as well with some cleverness.

The sine transform is defined as:

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \tag{15}$$

It is not just the imaginary part of the Fourier transform! It contains waves which fill the array with half-integer numbers of wavelengths. It is relevant to solving partial differential equations with zero value boundary conditions; similarly, the cosine transform is relevant to problems with zero derivative boundary conditions.

A sine transform can be reduced to a regular Fourier transform as follows. Define an new array:

$$y_j = \sin(j\pi/N)\left(f_j + f_{N-j}\right) + \frac{1}{2}\left(f_j - f_{N-j}\right) \tag{16}$$

The first part is symmetric (around $j = N/2$) and the second is antisymmetric. In the Fourier transform, this means that the real part (projected onto cos()) survives in the real part while the imaginary part (projected onto sin()) survives in the imaginary part.

Then:

$$Y_k^{(r)} = \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N)$$

$$
\begin{aligned}
&= \sum_{j=0}^{N-1} (f_j + f_{N-j}) \sin(\pi j/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} f_j \sin(\pi j/N) \cos(2\pi jk/N) + \sum_{j=0}^{N-1} f_{N-j} \sin(\pi j/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} f_j \sin(\pi j/N) \cos(2\pi jk/N) + \sum_{j=0}^{N-1} f_j \sin(\pi(N-j)/N) \cos(2\pi(N-j)k/N) \\
&= \sum_{j=0}^{N-1} f_j \sin(\pi j/N) \cos(2\pi jk/N) + \sum_{j=0}^{N-1} f_j \sin(\pi - \pi j/N) \cos(2\pi - 2\pi jk/N) \\
&= \sum_{j=0}^{N-1} f_j \sin(\pi j/N) \cos(2\pi jk/N) + \sum_{j=0}^{N-1} f_j \sin(\pi j/N) \cos(2\pi jk/N) \\
&= \sum_{j=0}^{N-1} 2 f_j \sin(\pi j/N) \cos(2\pi jk/N) \qquad (17)
\end{aligned}
$$

And using:

$$\sin A \cos B = \frac{1}{2} \left(\sin(A+B) + \sin(A-B) \right) \qquad (18)$$

we find:

$$
\begin{aligned}
Y_k^{(r)} &= \sum_{j=0}^{N-1} f_j \left(\sin(\pi j/N + 2\pi jk/N) + \sin(\pi j/N - 2\pi jk/N) \right) \\
&= \sum_{j=0}^{N-1} f_j \left(\sin(\pi(2k+1)j/N) - \sin(\pi(2k-1)j/N) \right) \\
&= F_{2k+1} - F_{2k-1} \qquad (19)
\end{aligned}
$$

where in the last line we just recognize that the terms are exactly terms of the sine tranform.

A similar analysis shows:

$$Y_k^{(i)} = F_{2k} \qquad (20)$$

So the even terms are easy. The odd terms can be determined with:

$$F_1 = -F_{-1} \quad \rightarrow \quad F_1 = \frac{1}{2} Y_k^{(r)} \qquad (21)$$

and then by using:

$$F_{2k+1} = F_{2k-1} + Y_k^{(r)} \qquad (22)$$

A very similar technique will yield an efficient cosine tranform.

6. Multidimensional FFTs

Multidimensial discrete Fourier transforms of course exist. They have the expected form:

$$H_{n_1 n_2} = \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp\left(2\pi i k_2 n_2/N_2\right) \exp\left(2\pi i k_1 n_1/N_1\right) h_{k_1 k_2} \tag{23}$$

Clearly these can be calculated with a series of one-dimensional FFTs. Since the FFTs are linear, they are therefore commutative, so it doesn't matter how you do it.

However, it is reasonably complicated to implement. Especially in the context of the sort of tricks I have discussed, it is the right thing to use the existing packages for these techniques.

7. Application: convolution

The convolution of two functions is as follows:

$$(f * g)(t) = \int_{-\infty}^{\infty} \mathrm{d}t' f(t - t') g(t') \tag{24}$$

In this formulation, one often thinks of f as a "kernel" with which you are "smoothing" g. E.g., g is a delta function, then you just get back the kernel.

As noted above, the Fourier transform of the convolution is the Fourier transform of the two functions, multiplied together:

$$\mathrm{FT}(f * g) = F \times G \tag{25}$$

In the notebook I show several examples, in 1D and 2D.

8. Application: filtering

"Filtering" is really just another form of convolution. Usually however, filtering is performed by setting ranges of the Fourier components to zero. This will correspond to a kernel whose Fourier transform is unity everywhere except at those Fourier components. The kernel in configuration space will be positive and negative.

In the notebook I show a 2D example of this.

9. Application: differential equations

FFTs are used in differential equations as a special case of "spectral methods." These methods are related to what we did in the eigensystem case, modeling waves on a string.

The basic idea is well-illustrated by the case of solving Poisson's equation.

$$\nabla^2\phi(\vec{x}) = 4\pi\rho(\vec{x}) \tag{26}$$

In this case for simplicity will consider periodic boundary conditions.

The solution to the equation can be found through Fourier transforms as follows:

$$-k^2\tilde{\phi}(\vec{k}) = 4\pi\tilde{\rho}(\vec{k}) \tag{27}$$

so:

$$\tilde{\phi}(\vec{k}) = -4\pi k^{-2}\tilde{\rho}(\vec{k}) \tag{28}$$

and then we just Fourier transform back.

We could imagine doing this on a grid with a discrete Fourier transform. I.e. Fourier transform to $\tilde{\rho}$ on a grid, and then multiple by k^{-2} and then transform back. But it turns out that doing so is not exactly the best thing to do; in particular, the high k values cause trouble and lead to errors.

The better way to think about the problem is to perform a discrete Fourier transform on the discrete version of Poisson's equation. The discrete transform can be derived by considering the estimate of the derivative at each half grid point $n + 1/2$:

$$\left.\frac{\partial\phi}{\partial x}\right|_{n+1/2} = \frac{\phi_{n+1} - \phi_n}{\Delta} \tag{29}$$

and the second derivative at grid point n:

$$\begin{aligned}
\left.\frac{\partial^2\phi}{\partial x^2}\right|_n &= \frac{1}{\Delta}\left[\left.\frac{\partial\phi}{\partial x}\right|_{n+1/2} - \left.\frac{\partial\phi}{\partial x}\right|_{n-1/2}\right] \\
&= \frac{1}{\Delta^2}\left[\phi_{n+1} - \phi_n - \phi_n + \phi_{n-1}\right] \\
&= \frac{1}{\Delta^2}\left[\phi_{n+1} - 2\phi_n + \phi_{n-1}\right]
\end{aligned} \tag{30}$$

This makes the equations as follows:

$$\frac{1}{\Delta^2}\left[\phi_{n+1} - 2\phi_n + \phi_{n-1}\right] = 4\pi\rho_n \tag{31}$$

Now think about the Fourier transform of this equation. We need the Fourier transforms:

$$\begin{aligned}
\phi_n &= \Delta\sum_m \tilde{\phi}_m \exp\left[2\pi i m n/N\right] \\
\phi_{n+1} &= \Delta\sum_m \tilde{\phi}_m \exp\left[2\pi i m(n+1)/N\right] \\
&= \Delta\sum_m \tilde{\phi}_m \exp\left[2\pi i m/N\right]\exp\left[2\pi i m n/N\right] \\
\phi_{n-1} &= \Delta\sum_m \tilde{\phi}_m \exp\left[2\pi i m(n-1)/N\right]
\end{aligned}$$

$$= \quad \Delta \sum_m \tilde{\phi}_m \exp\left[-2\pi i m/N\right] \exp\left[2\pi i m n/N\right] \tag{32}$$

So the Fourier transforms of these three terms are related by:

$$\begin{aligned} \mathrm{FT}(\phi_n) &= \tilde{\phi}_m \\ \mathrm{FT}(\phi_{n+1}) &= \tilde{\phi}_m \exp\left[2\pi i m/N\right] \\ \mathrm{FT}(\phi_{n-1}) &= \tilde{\phi}_m \exp\left[-2\pi i m/N\right] \end{aligned} \tag{33}$$

And the Fourier transform of the discrete Poisson equation becomes:

$$\frac{1}{\Delta^2}\left[\exp\left(2\pi i m/N\right) + \exp\left(-2\pi i m/N\right) - 2\right]\tilde{\phi}_m = 4\pi\tilde{\rho}_m \tag{34}$$

or:

$$\frac{2}{\Delta^2}\left[\cos\left(2\pi m/N\right) - 1\right]\tilde{\phi}_m = 4\pi\tilde{\rho}_m \tag{35}$$

and this is the right equation to use. For small m notice that this is precisely $-k^2$. But for high m the term is limited in size, which prevents small scale terms from dominating.

In the Gravity on a Mesh project, part of the exercise is to design this factor for the case of isolated boundary conditions, which is somewhat different and has to itself be calculated numerically.