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 $h(t)$?**

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:

$$\mathrm{FT}(f * g) = F \times G \tag{2}$$

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:

$$\int_{-\infty}^{\infty} \mathrm{d}t \left|h(t)\right|^2 = \int_{-\infty}^{\infty} \mathrm{d}f \left|H(f)\right|^2 \tag{3}$$

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?**

**What are these frequencies?**

Then the discrete tranform is:

$$\begin{aligned} H(f_n) &= \int_{-\infty}^{\infty} \mathrm{d}t h(t) \exp(2\pi i f_n t) \\ &\approx \sum_{k=0}^{N-1} h(t_k) \exp(2\pi i f_n t_k) \Delta \\ &\approx \Delta \sum_{k=0}^{N-1} h(t_k) \exp(2\pi i (n/N\Delta)(k\Delta) \\ &\approx \Delta \sum_{k=0}^{N-1} h(t_k) \exp(2\pi i k n/N) \end{aligned} \tag{4}$$

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 inversion of $H_n$ is:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \exp(-2\pi i k n/N) \tag{5}$$

## 3. Calculation of a Fast Fourier Transform

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

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 :

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

$$
\begin{aligned}
&= \sum_{j=0}^{N/2-1} \exp(2\pi ijk/(N/2)) h_{2j} \exp(2\pi ik/N) \sum_{j=0}^{N/2-1} \exp(2\pi ijk/(N/2)) h_{2j+1} \\
&= H_k^e + W^k J_k^o
\end{aligned} \tag{6}
$$

where $H_k^e$ is the Fourier transform of the even components, $H_k^o$ is the Fourier transform of the odd components, and $W = \exp(2\pi i/N)$.

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.

If you label the Fourier transform at each leaf as something like $H_k^{eeoeoe\cdots}$, then it corresponds to some $h_n$. Which $h_n$? 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 $n$.

To take the FFT, you can then just reorder $h_n$ appropriately, and then apply the Danielson-Lanczos lemma in a tree-like fashion. This leads to $\sim N$ operations executed $\log_2 N$ times.

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:

$$h_j = f_{2j} + i f_{2j+1} \tag{7}$$

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

$$H_n = F_n^e + i F_n^o \tag{8}$$

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

$$F_n = F_n^e + \exp(2\pi in/N) F_n^o \tag{9}$$

We can then note:

$$
\begin{aligned}
F^e &= \frac{1}{2}(H + H^*) \\
F^o &= \frac{1}{2}(H - H^*)
\end{aligned} \tag{10}
$$

and so

$$F_n = \frac{1}{2}\left(H_n + H^*_{N/2-n}\right) + \frac{1}{2}\exp(2\pi i n/N)\left(H_n - H^*_{N/2-n}\right) \tag{11}$$

## 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:

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \tag{12}$$

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:

$$y_j = \sin(j\pi/N)\,(f_j + f_{N-j}) + \frac{1}{2}\,(f_j - f_{N-j}) \tag{13}$$

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:

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

$$= \sum_{j=0}^{N-1} 2 f_j \sin(\pi j/N) \cos(2\pi jk/N) \tag{14}$$

And using:

$$\sin A \cos B = \frac{1}{2} \left( \sin(A+B) + \sin(A-B) \right) \tag{15}$$

we find:

$$\begin{aligned} Y_k^{(r)} &= \sum_{j=0}^{N-1} f_j \left( \sin(\pi j/N + 2\pi jk/N) + \sin(\pi j/N - 2\pi jk/N) \right) \\ &= \sum_{j=0}^{N-1} f_j \left( \sin(\pi(2k+1)j/N) - \sin(\pi(2k-1)j/N) \right) \\ &= F_{2k+1} - F_{2k-1} \end{aligned} \tag{16}$$

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

A similar analysis shows:

$$Y_k^{(i)} = F_{2k} \tag{17}$$

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

$$F_1 = -F_{-1} \quad \rightarrow \quad F_1 = \frac{1}{2} Y_k^{(r)} \tag{18}$$

and then by using:

$$F_{2k+1} = F_{2k-1} + Y_k^{(r)} \tag{19}$$

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:

$$H_{n_1 n_2} = \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp\left(2\pi i k_2 n_2/N_2\right) \exp\left(2\pi i k_1 n_1/N_1\right) h_{k_1 k_2} \tag{20}$$

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:

$$(f * g)(t) = \int_{-\infty}^{\infty} \mathrm{d}t' f(t - t') g(t') \tag{21}$$

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:

$$\mathrm{FT}(f * g) = F \times G \tag{22}$$

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.