1 Trigonometric interpolation and the FFT

1.1 Some preliminaries

So far, we have talked about using polynomials or piece-wise polynomials to interpolate data for which the underlying function is smooth. If we can further assume that this function is periodic, it is natural to use a basis of periodic functions for interpolation and approximation. Periodic functions appear often in models of wave-like phenomena (accoustics, electromagnetics, light) as well as to represent variables defined in circles, closed curves, spheres, cylinders, etc. Applying these ideas in 2 or 3 dimensions is one of the key mathematical tools behind all kinds of imaging techniques.

When it comes to periodic function approximation, we are familiar with the concept of a Fourier series. Given a smooth, 2π periodic function $f(x)$, we write

$$f(x) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kx) + b_k \sin(kx)$$

Where the coefficients can be obtained directly from the fact that the basis of sines and cosines is orthogonal with respect to the inner product $< f, g >= \int_{-\pi}^{\pi} f(x)g(x)dx$. This gives us:

$$\begin{aligned}
a_0 &= \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)dx = \frac{< f, 1 >}{< 1, 1 >} \\
a_k &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(kx)dx = \frac{< f, \cos(kx) >}{< \cos(kx), \cos(kx) >} \\
b_k &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(kx)dx = \frac{< f, \sin(kx) >}{< \sin(kx), \sin(kx) >}
\end{aligned}$$

In what sense and how rapidly this Fourier series converges to f depends on what we can assume about the smoothness of f. This is important because, for computational purposes, we usually truncate the infinite sum to the first $2n + 1$ terms (up to $k = n$):

$$f(x) \simeq p(x) = a_0 + \sum_{k=1}^{n} a_k \cos(kx) + b_k \sin(kx)$$

We say $p(x)$ here is a *trigonometric polynomial* of degree $\leq n$, and define the space of such polynomials as $\mathcal{T}_n$. This is a $2n+1$ dimensional vector space and its canonical basis is $\{1, \cos(x), \sin(x), \ldots, \cos(nx), \sin(nx)\}$.

1.2 Trigonometric polynomial interpolation problem

We define an interpolation problem: to find $p \in \mathcal{T}_n$ given $2n + 1$ distinct interpolation nodes $x_j \in [0, 2\pi)$ and data $y_j = f(x_j)$. The reason we do not include 2π is because of periodicity: $x = 0$ and $x = 2\pi$ are the same location.

For both approximation and interpolation, it is very useful to re-write real-valued Fourier series as a sum of complex exponentials. This doesn't only make mathematical manipulation easier, but it links it with complex-valued Fourier series and Taylor series of analytic functions on the unit circle.

Using Euler's formula, we have that:

$$
\begin{aligned}
(e^{ix})^k = e^{ikx} &= \cos(kx) + i\sin(kx) \\
\cos(kx) &= \frac{1}{2}(e^{ikx} + e^{-ikx}) \\
\sin(kx) &= \frac{1}{2i}(e^{ikx} - e^{-ikx})
\end{aligned}
$$

Plugging this into the general formula for $p(x)$, we get:

$$
\begin{aligned}
p(x) &= a_0 + \sum_{k=1}^{n} \frac{a_k}{2}(e^{ikx} + e^{-ikx}) + \frac{b_k}{2i}(e^{ikx} - e^{-ikx}) \\
&= a_0 + \sum_{k=1}^{n} \frac{a_k - ib_k}{2} e^{ikx} + \frac{a_k + ib_k}{2} e^{-ikx} \\
&= \sum_{k=-n}^{n} c_k e^{ikx}
\end{aligned}
$$

Now, if we define $z = e^{ix}$, this is a standard way to write a complex number $z \in \mathbb{C}$ on the unit circle. (In general, we can always write $z = |z|e^{i\theta}$ for $\theta \in [0, 2\pi)$). This means our trigonometric polynomial is the restriction of the complex rational function $p(z) = \sum_{k=-n}^{n} z^k$ to the unit circle.

Because we are in the unit circle, we can multiply this polynomial by z^n and obtain an actual complex polynomial. On the unit circle, this is just a rotation by an angle nx. We define:

$$
P(z) = z^n p(x) = \sum_{k=0}^{2n} \tilde{c_k} z^k
$$

Where $\tilde{c_k} = c_{k-n}$ (we just re-index). Going back to the interpolation problem, if we define $z_j = e^{ikx_j}$, we have the equations:

$$
\begin{aligned}
z_j^n f(x_j) = P(z_j) &= \sum_{k=0}^{2n} \tilde{c_k} z_j^k && (1.1) \\
&= \sum_{k=0}^{2n} \tilde{c_k} \left(e^{ix_j}\right)^k && (1.2)
\end{aligned}
$$

We can write this as a set of linear equations:

$$
\begin{aligned}
\begin{bmatrix}
1 & z_0 & z_0^2 & \dots & z_0^{2n} \\
1 & z_1 & z_1^2 & \dots & z_1^{2n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & z_{2n} & z_{2n}^2 & \dots & z_{2n}^{2n}
\end{bmatrix}
\begin{bmatrix}
\tilde{c_0} \\ \tilde{c_1} \\ \vdots \\ \tilde{c_{2n}}
\end{bmatrix}
&=
\begin{bmatrix}
z_0^n f(x_0) \\ z_1^n f(x_1) \\ \vdots \\ z_{2n}^n f(x_{2n})
\end{bmatrix} && (1.3) \\
\mathbf{Vc} &= \mathbf{F} && (1.4)
\end{aligned}
$$

This is a Vandermonde system, but the fact that its entries are all on the unit circle makes a gigantic difference in terms of conditioning and how quickly and accurately we can solve this system.

From now on, we will assume that our interpolation nodes are equispaced, that is, $x_j = j\frac{2\pi}{2n+1}$, for $j = 0, \dots, 2n$. If that is the case, we can show that $\mathbf{V}$ has *orthogonal columns*, and $\mathbf{V}^*\mathbf{V} = (2n+1)\mathbf{I}$.

Let's denote $\mathbf{v}_\ell = \mathbf{V}(:,\ell)$. Then, we must show that $(\mathbf{v}_k)^*\mathbf{v}_\ell = (2n+1)\delta_{k,\ell}$. Using the formula for the entries of the Vandermonde matrix:

$$(\mathbf{v}_k)^*\mathbf{v}_\ell = \sum_{j=0}^{2n} z_j^{\ell-k} \tag{1.5}$$

$$= \sum_{j=0}^{2n} \left(e^{i(\ell-k)\frac{2\pi}{2n+1}}\right)^j \tag{1.6}$$

$$\tag{1.7}$$

This is a sum of powers of $r = e^{i(\ell-k)\frac{2\pi}{2n+1}}$, and so, it is a geometric sum. Assume $\ell - k \neq 0$, then $r \neq 1$, and from Calculus II, we have the formula:

$$(\mathbf{v}_k)^*\mathbf{v}_\ell = \frac{1-r^{2n+1}}{1-r} = \frac{1-e^{2\pi i(\ell-k)}}{1-e^{i(\ell-k)\frac{2\pi}{2n+1}}} = 0 \tag{1.8}$$

since $e^{2\pi i m} = 1$ for any integer m. On the other hand, if $\ell - k = 0$, then

$$(\mathbf{v}_k)^*\mathbf{v}_k = \sum_{j=0}^{2n} \left(e^0\right)^j = 2n+1$$

So, not only is $\mathbf{V}$ really well-conditioned ($\kappa(\mathbf{V}) = 1$), we can solve the system in $O(n^2)$ operations by applying the transpose and dividing by $(2n+1)$:

$$(2n+1)\mathbf{c} = \mathbf{V}^*\mathbf{V}\mathbf{c} = \mathbf{V}^*\mathbf{F} \tag{1.9}$$

$$\mathbf{c} = \frac{1}{(2n+1)}\mathbf{V}^*\mathbf{F} \tag{1.10}$$

Let's see what formula we get for our coefficients:

$$\tilde{c}_k = \frac{1}{2n+1}\sum_{j=0}^{2n} z_j^{-k} z_j^{n} f(x_j) \tag{1.11}$$

$$= \frac{1}{2n+1}\sum_{j=0}^{2n} \left(e^{ij\frac{2\pi}{2n+1}}\right)^{n-k} f(x_j) \tag{1.12}$$

This looks very similar to the *Discrete Fourier Transform* of $f(x)$; we have really just "shifted" the coefficients by n and divided them by $N = 2n+1$. That is, for $k = 0, 1, \dots, 2n$, if we define:

Definition 1 (Discrete Fourier Transform) *Given a vector* $\mathbf{f} \in \mathbb{C}^N$*, we define its Discrete Fourier Transform* $\widehat{\mathbf{f}} = \mathbf{D}[\mathbf{f}]$ *as*

$$\widehat{\mathbf{f}}_k = \mathbf{D}[\mathbf{f}]_k = \sum_{j=0}^{N-1} \left(e^{ij\frac{2\pi}{N}}\right)^{-k} \mathbf{f}_j \tag{1.13}$$

It can be written as a matrix-vector product, with $\mathbf{D}_{k,j} = \left(e^{ij\frac{2\pi}{N}}\right)^{-k}$.

We note that this DFT matrix $\mathbf{D}$ is $\mathbf{V}^*$ for $N = 2n+1$ equispaced nodes. Then, for the vectors of function samples $\mathbf{f}_j = f(x_j)$ and $\mathbf{F}_j = z_j^n f(x_j)$, we can show that:

$$\tilde{c}_k = \frac{1}{2n+1}\widehat{\mathbf{f}}_{s(k)} = \frac{1}{2n+1}\widehat{\mathbf{F}}_k \tag{1.14}$$

where $s(k) = (k-n)\text{mod}(2n+1)$ shifts indices accordingly. That is: after dividing by the number of interpolation nodes, the DFT is a transform that goes from equispaced function evaluations on the complex unit circle to coefficients of a polynomial interpolant $P(z)$. This corresponds to trigonometric interpolation for the original function samples $f(x_j)$.

The *Inverse Discrete Fourier Transform* (IDFT) is just the opposite map (from coefficients to polynomial interpolant evaluation at equispaced points in the unit circle). Given our notation, $\mathbf{D}^{-1} = \frac{1}{2n+1}\mathbf{D}^* = \frac{1}{2n+1}\mathbf{V}$.

1.3 The Fast Fourier Transform

The ***Fast Fourier Transform*** (FFT) is recognized as one of the ten most important algorithms of the 20th century, and for good reason. Turns out these sums of complex exponentials given by the Discrete Fourier Transform (DFT) show up everywhere in applications, and the FFT gives us an algorithm to compute the DFT of $N = (2n+1)$ function samples (equispaced in the unit circle) in $O(N \log N)$ time (with a small constant, about 3/2)! For our purposes, this means we can calculate our coefficients with two (very fast) lines of code:

1. $\widehat{f} = \text{fft}(\mathbf{f})$ where $\mathbf{f}(j) = f(x_j)$.
2. $\mathbf{c} = \frac{1}{2n+1}\text{fftshift}(\widehat{f})$.

(both fft and fftshift are functions of numpy.fft in Python). Equivalently, we can compute an FFT of $\mathbf{F}$.

So, how does the FFT work? The main idea behind this algorithm is a divide-and-conquer strategy, and it was first used by none other than Gauss in 1805, to interpolate asteroid paths from samples of their trajectories. However, this algorithm was not formalized until 1965, by Cooley and Tukey.

The key result that needs to be shown is this: *a DFT for N points can be calculated using two DFTs for $N/2$ points, and then adding a linear combination of them (which is $2N$ operations).* If $W_0(N)$ is the work for one DFT, this divide-and-conquer approach costs $W_1(N) = 2W_0(N/2) + 2N$.

This idea is then applied recursively / repeatedly: two $N/2$ DFTs involve four $N/4$ DFTs, and so on. If $N = 2^L$ ($L = \log_2 N$), we can bisect the size of the DFT exactly L times, and end up with N DFTs of size 1. The amount of work for L bisections of the data, $W_L(N)$, is:

$$\begin{aligned} W_L(N) &= 2W_{L-1}(N/2) + 2N \\ &= 4W_{L-2}(N/4) + 4(N/2) + 2N = \cdots \\ &= NW_0(1) + 2LN = N + 2N\log_2 N \end{aligned}$$

This algorithm can be sped up a bit so that the constant in front of the $N\log_2 N$ is 3/2, and further optimizations are usually made in software packages such as fftw. For 2D and 3D imaging and scattering applications, this fast algorithm is very literally what makes them possible.

1.3.1 The FFT algorithm

We recall that the DFT can be interpreted as a polynomial evaluation for a set of equispaced complex numbers in the unit circle. That is, given the definition of an N point DFT:

$$\mathbf{D}[\mathbf{f}]_k = \sum_{j=0}^{N-1} \left(e^{ij\frac{2\pi}{N}}\right)^{-k} \mathbf{f}_j \tag{1.15}$$

If we define the polynomial $P(z) = \sum_{j=0}^{N-1} \mathbf{f}_j z^k$ in $\mathcal{P}_{N-1}$, computing the DFT is evaluating $P(z_j^{-1}) = P(z_j^*)$, with $z_j = e^{ij\frac{2\pi}{N}}$ the set of uniform samples in the unit circle.

We can re-write this polynomial as the sum of two pieces:

$$\begin{aligned} P(z) &= (f_0 + f_2 z^2 + f_4 z^4 + \cdots) + z(f_1 + f_3 z^2 + f_5 z^4 + \cdots) && (1.16)\\ &= P_{\text{even}}(z^2) + z P_{\text{odd}}(z^2) && (1.17) \end{aligned}$$

for polynomials $P_{\text{even}}, P_{\text{odd}}$ of degree $\leq N/2 - 1$ (for N even, for N odd they are $\leq (N-1)/2$).

It seems like we have only re-written the formula, but here is the trick:

$$\begin{aligned} z_j^2 &= e^{2\pi i \frac{2j}{N}} \\ &= e^{2\pi i} e^{2\pi i (\frac{2j}{N})} = e^{2\pi i (\frac{2j}{N} + 1)} \\ &= e^{2\pi i (\frac{2(j+N/2)}{N})} = z_{j+N/2}^2 \end{aligned}$$

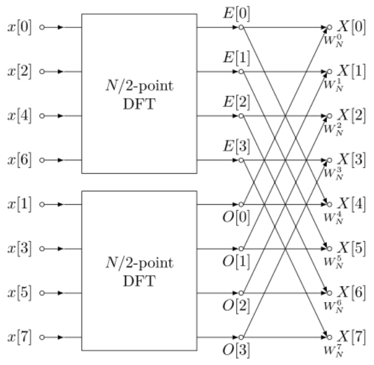
So, evaluating $P_{\text{even}}(z_j^2)$ only requires us to evaluate for $j = 0, 1, \ldots, N/2 - 1$. Intuitively: because we are using N roots of unity *squared*, they wrap twice around the unit circle, so we only need to evaluate these polynomials at half of them (one revolution around the circle).

The algorithm to compute $P(z_j)$ is then:

```
1. Compute all $P_{\text{even}}(z_j^2)$ and $P_{\text{odd}}(z_j^2)$ using two DFTs of size $N/2$.
2. for $j = 0, 1, \ldots, N/2 - 1$:
3.     $P(z_j) = P_{\text{even}}(z_j^2) + z_j P_{\text{odd}}(z_j^2)$
4.     $P(z_{j+N/2}) = P_{\text{even}}(z_j^2) + z_{j+N/2} P_{\text{odd}}(z_j^2)$
```

Each polynomial evaluation is a DFT for $N/2$ points. We then just do one multiplication and one addition for each one of the N evaluations $P(z_j)$: $2N$ operations. Further optimization can be obtained by realizing that $z_{j+N/2} = -z_j$; this means we add $z_j P_{\text{odd}}(z_j^2)$ for the first $N/2$ points, and subtract $z_j P_{\text{odd}}(z_j^2)$ for the last $N/2$ points. This means the work after one round of divide and conquer is $W_1(N) = 2W_0(N/2) + (3/2)N$.

Given how data flows in this algorithm, it is sometimes known as the *butterfly algorithm* or the FFT butterfly algorithm. The one-level divide and conquer algorithm can be illustrated with the diagram:

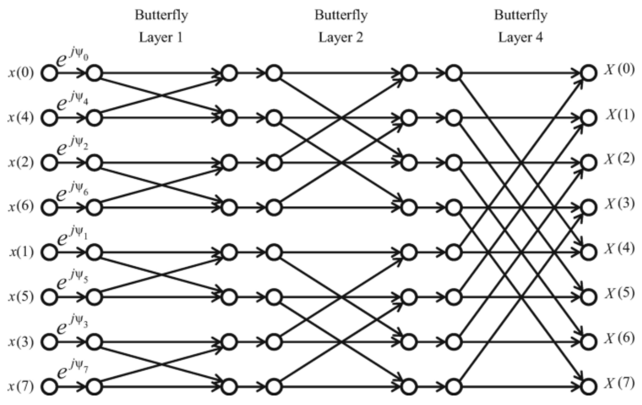


Using this algorithm as our basis, we can write down the FFT algorithm in one of two equivalent ways:

- A *recursive* algorithm can be written; the DFT calls are recursive calls to the same function for each half of the samples (even and odd numbered). The recursion stops when the DFTs are size 1 (or a fixed small size).
- We can think of this algorithm as taking the list of indices and defining a *binary tree* of depth $L = \log_2 N + 1$; each parent of the tree splits into two children: even and odd indices. We can write the algorithm collecting contributions from children to parents using a for loop.

This results in a fast algorithm that computes the DFT in $N + (3/2)N \log_2 N$ operations. It should be noted that if N is such that the division can't be done neatly (N is not a power of 2), zero-padding can be done to apply this algorithm as described.

The multi-level FFT divide and conquer algorithm can be illustrated with the "butterfly diagram" (for $N = 8$ points and 3 levels in this example):



Butterfly Layer 1
Butterfly Layer 2
Butterfly Layer 4
x(0)
x(4)
x(2)
x(6)
x(1)
x(5)
x(3)
x(7)
X(0)
X(1)
X(2)
X(3)
X(4)
X(5)
X(6)
X(7)