Introduction to the Fast Fourier Transform

Contents

1 Introduction: Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a function that maps a discretely sampled signal (can be sampled in time, space or any other dimension) to a frequency domain representation. Discrete Fourier transform of a discrete periodic signal $x(t)$

$$X(f) = \mathcal{F}\left(X(f)\right) = \frac{1}{N}\sum_{t=0}^{N-1} x(t)\exp\left(-j\frac{2\pi}{N}ft\right)$$

where $x(t)$ is the signal, t is time or sample index, $X(f)$ is the frequency representation and f is the frequency.

2 Devising a Simple Algorithm to Compute the DFT (but too long running time).

The DFT consists of a complex exponential multiplied by a discretely sampled signal $x(t)$. The real and imaginary components of the complex exponential and of the sampled signal can be calculated independently. Firstly we need to separate the real and imaginary parts of the complex exponential using Euler's identity.

The simple algorithm `simpleDFT` has $\times 2$ for loops, one *nested* inside the other, both with length N. This means that it has a running time with an upper bound that is proportional to N^2. This can be expressed more formerly using Big Oh notation: The algorithm `simpleDFT` has run time complexity of $O(N^2)$. Big Oh is actually a set definition for a class of algorithms with a particular asymptotic upper bound:

$$O(g(N)) = \{f(N)|0 < f(N) \leq cg(N) \text{ and } N > N_0 \text{ where } N_0 \text{ and } c \text{ are positive constants}\}.$$

So the meaning of $O(N^2)$ for `simpleDFT` is that it will take at most $c \times N^2$ time for $N \geq N_0$ where both c and N_0 are positive constants. This run time complexity tells us that `simpleDFT` for an input of length N will have running time that has an upper bound on the square of the length of the input sequence, which perhaps is a bit slow, especially for large N which will be common for many types of digital signal.

3 How to Reduce Running Time? Divide and Conquer

Divide and conquer is an algorithm design technique where input data to a function is split into 2 or more components and then some operation, usually a recursive application of the same procedure, is applied to the split components. Recursive application of this process eventually results in the data being split into some elementary number of elements, *e.g.* $N' = 1$. This approach to algorithm design can often produce an algorithm with a faster running time.

The DFT can in fact be split into two components both of which are also DFTs. So that $N \times X(f) = X^{x_e}_{N/2}(f) + T(N, f) \times X^{x_o}_{N/2}(f)$ where $X^{x_e}_{N/2}(f)$ and $X^{x_o}_{N/2}(f)$ correspond to DFTs of the even and odd components. The remaining term, $T(N, f)$ is known as a twiddle factor.

This splitting of the DFT can be performed recursively splitting each even and odd component into their own even and odd components. This is very useful because it is exactly what is needed by a divide and conquer

algorithm technique to recursively divide the data and then apply the same technique to the split data. This is the main principal of the **Radix-2 Decimation in Time (DIT) Fast Fourier Transform (FFT)** proposed by **Cooley and Tukey**[1].

The DFT is also symmetric about the point $N/2$ which means $X(f) = X(f+N/2)$. So the Fourier transforms can be replicated for the upper part of the spectrum due to this aliasing property. Also note $\exp(-j2\pi(f + N/2)/N) = -\exp(-j2\pi f/N)$. This information can be used to adjust the twiddle factor $T(N, f)$ where $T(N, f + N/2) = -T(N, f)$.

4 Resulting Recursive FFT Algorithm

The resulting recursive FFT algorithm `recursiveFFT` is a radix 2 DIT out-of-place recursive FFT. **So what is the running time?** The algorithm consists of two recursive function calls where the data (size N) has been split into two equal proportions of $N/2$. There is also a for loop with $N/2$ iterations. The running time can therefore be described recursively:

$$T(N) = T(N/2) + T(N/2) + N/2 = 2T(N/2) + N/2.$$

We would like to compare the running time of `recursiveFFT` with the run time complexity for `simpleDFT` so this recursive description of the run time is not very useful, we need to find a closed form solution. The closed form solution can be found by either the substitution method or a technique known as the Master Method. Both techniques are described in detail in *e.g.* Cormen et al.[2].

An outline of the substitution method is now shown. The substitution method to solve recurrences consists of two main steps: 1) determine *recurring* expressions for the recurrence relation; and 2) use the recurring expressions to substitute into each other. A series of recurring expressions for the run time of `recursiveFFT` are:

N		Expression	
N	$T(N)$	$= 2T(N/2) + N/2$	(1)
$N/2$	$T(N/2)$	$= 2T(N/(2 \times 2)) + N/(2 \times 2)$	(2)
$N/(2 \times 2)$	$T(N/(2 \times 2))$	$= 2T(N/(2 \times 2 \times 2)) + N/(2 \times 2 \times 2)$	(3)
$N/(2 \times 2 \times 2)$	$T(N/(2 \times 2 \times 2))$	$= 2T(N/(2 \times 2 \times 2 \times 2)) + N/(2 \times 2 \times 2 \times 2)$	(4)
$N/(2^3)$	$T(N/2^3)$	$= 2T(N/2^4) + N/2^4$	(5)
$N/(2^4)$	$T(N/2^4)$	$= 2T(N/2^5) + N/2^5$	(6)
$\vdots$	$\vdots$	$\vdots$	$\vdots$

We can use these to substitute back into the initial expression for $T(N)$. Starting by substituting (2) into (1):

	Expression	
$T(N)$	$= 2T(N/2) + N/2$	(1)
	$= 2\left(2T(N/(2 \times 2)) + N/(2 \times 2)\right) + N/2$	*(2) into (1)*
	$= 2^2 T(N/2^2) + 2N/2$	(7)
$T(N)$	$= 2^2\left(2T(N/2^3) + N/2^3\right) + 2N/2$	*(3) into (7)*
	$= 2^3 T(N/2^3) + 3N/2$	(8)
$T(N)$	$= 2^3\left(2T(N/2^4) + N/2^4\right) + 3N/2$	*(4) into (8)*
	$= 2^4 T(N/2^4) + 4N/2$	(9)
$\vdots$	$\vdots$	$\vdots$

After a number of steps you can often start to see a sequence emerging. Here we have:

$$T(N) = 2^i T(N/2^i) + iN/2 \quad \text{for } 1 \leq i \leq \infty.$$

This is still in recursive form and with an additional variable i. We can eliminate i if we look at the base case for `recursiveFFT`:

```
if N = 1
    XRe[0] = xRe[0], XIm[0] = xIm[0]
else
⋮
```

Here the base case says that when $N = 1$ we have a constant time expression (no for loops). So when $N = 1$ we let $T(N = 1) = 1$. This gives us enough information to eliminate i because we know that a value of i must

[1] J. Cooley and J. Tukey, *"An algorithm for the machine calculation of complex Fourier series"*, Math. Comput. 19: 297-301.
[2] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *"Introduction to Algorithms*, MIT Press.

be found that satisfies both the base case and $T(N/2^i)$. If we let $N/2^i = 1$ then $N = 2^i$ therefore $i = \log_2(N)$. Thus when $i = \log_2(N)$ we have $T(N/N) = T(1) = 1$, resulting in:

$$T(N) = 2^{\log_2(N)} + \frac{N}{2}\log_2(N) = N + \frac{N}{2}\log_2 N.$$

Big Oh notation enables us to ignore the lower power terms and any constant factors, which in this case is the N in the first term and the division by 2 in the second, so we can say $T(N) = O(N \log N)$. This is much faster than the running time for `simpleDFT` algorithm with $O(N^2)$.

5 Any Further Improvements? Yes, remove the recursion!

Recursion is not always very useful, particularly algorithms that result in very deep (many levels) of recursion which may then result in over using the call stack. The call stack is used to store return addresses, function arguments and other information every time a function is called. For embedded systems with limited memory the stack size will also be of limited size. Recursive functions can sometimes overflow the call stack (exceed the limit), particularly on systems with small stack sizes. Therefore it is often better to convert a recursive function to a non-recursive function so that it does not have a risk of overflowing the call stack.

The `recursiveFFT` algorithm can result in many levels of recursion, *e.g.* for a signal consisting of only $N = 8$ samples the resulting recursion tree will have a depth of 4. Further levels are required for longer signals. The recursion tree is actually a binary tree and the number of levels in a binary tree is equal to $\log_2(N) + 1$.