Understanding the FFT Algorithm

Wed 28 August 2013

The Fast Fourier Transform (FFT) is one of the most important algorithms in signal processing and data analysis. I've used it for years, but having no formal computer science background, It occurred to me this week that I've never thought to ask *how* the FFT computes the discrete Fourier transform so quickly. I dusted off an old algorithms book and looked into it, and enjoyed reading about the deceptively simple computational trick that JW Cooley and John Tukey outlined in their classic 1965 paper (http://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/) introducing the subject.

The goal of this post is to dive into the Cooley-Tukey FFT algorithm, explaining the symmetries that lead to it, and to show some straightforward Python implementations putting the theory into practice. My hope is that this exploration will give data scientists like myself a more complete picture of what's going on in the background of the algorithms we use.

# The Discrete Fourier Transform

The FFT is a fast, $\mathcal{O}[N\log N]$ algorithm to compute the Discrete Fourier Transform (DFT), which naively is an $\mathcal{O}[N^2]$ computation. The DFT, like the more familiar continuous version of the Fourier transform, has a forward and inverse form which are defined as follows:

Forward Discrete Fourier Transform (DFT):

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i~2\pi~k~n~/~N}$$

Inverse Discrete Fourier Transform (IDFT):

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{i~2\pi~k~n~/~N}$$

The transformation from $x_n \to X_k$ is a translation from configuration space to frequency space, and can be very useful in both exploring the power spectrum of a signal, and also for transforming certain problems for more efficient computation. For some examples of this in action, you can check out Chapter 10 of our upcoming Astronomy/Statistics book, with figures and Python source code available here (http://www.astroml.org/book_figures/chapter10/). For an example of the FFT being used to simplify an otherwise difficult differential equation integration, see my post on Solving the Schrodinger Equation in Python (http://jakevdp.github.io/blog/2012/09/05/quantum-python/).

Because of the importance of the FFT in so many fields, Python contains many standard tools and wrappers to compute this. Both NumPy and SciPy have wrappers of the extremely well-tested FFTPACK library, found in the submodules `numpy.fft` and `scipy.fftpack` respectively. The fastest FFT I am aware of is in the FFTW (http://www.fftw.org/) package, which is also available in Python via the PyFFTW (https://pypi.python.org/pypi/pyFFTW) package.

For the moment, though, let's leave these implementations aside and ask how we might compute the FFT in Python from scratch.

# Computing the Discrete Fourier Transform

For simplicity, we'll concern ourself only with the forward transform, as the inverse transform can be implemented in a very similar manner. Taking a look at the DFT expression above, we see that it is nothing more than a straightforward linear operation: a matrix-vector multiplication of $\vec{x}$,

$$\vec{X} = M \cdot \vec{x}$$

with the matrix M given by

$$M_{kn} = e^{-i\,2\pi\,k\,n\,/\,N}.$$

With this in mind, we can compute the DFT using simple matrix multiplication as follows:

In [1]:
```
import numpy as np
def DFT_slow(x):
    """Compute the discrete Fourier Transform of the 1D array x"""
    x = np.asarray(x, dtype=float)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)
```

We can double-check the result by comparing to numpy's built-in FFT function:

```
In [2]:  x = np.random.random(1024)
         np.allclose(DFT_slow(x), np.fft.fft(x))

Out[2]:  True
```

Just to confirm the sluggishness of our algorithm, we can compare the execution times of these two approaches:

```
In [3]:  %timeit DFT_slow(x)
         %timeit np.fft.fft(x)

         10 loops, best of 3: 75.4 ms per loop
         10000 loops, best of 3: 25.5 µs per loop
```

We are over 1000 times slower, which is to be expected for such a simplistic implementation. But that's not the worst of it. For an input vector of length N, the FFT algorithm scales as $\mathcal{O}[N\log N]$, while our slow algorithm scales as $\mathcal{O}[N^2]$. That means that for $N = 10^6$ elements, we'd expect the FFT to complete in somewhere around 50 ms, while our slow algorithm would take nearly 20 hours!

So how does the FFT accomplish this speedup? The answer lies in exploiting symmetry.

# Symmetries in the Discrete Fourier Transform

One of the most important tools in the belt of an algorithm-builder is to exploit symmetries of a problem. If you can show analytically that one piece of a problem is simply related to another, you can compute the subresult only once and save that computational cost. Cooley and Tukey used exactly this approach in deriving the FFT.

We'll start by asking what the value of X_{N+k} is. From our above expression:

$$\begin{aligned} X_{N+k} &= \sum_{n=0}^{N-1} x_n \cdot e^{-i~2\pi~(N+k)~n~/~N}\\ &= \sum_{n=0}^{N-1} x_n \cdot e^{-i~2\pi~n} \cdot e^{-i~2\pi~k~n~/~N}\\ &= \sum_{n=0}^{N-1} x_n \cdot e^{-i~2\pi~k~n~/~N} \end{aligned}$$

where we've used the identity $\exp[2\pi~i~n] = 1$ which holds for any integer n.

The last line shows a nice symmetry property of the DFT:

$$X_{N+k} = X_k.$$

By a simple extension,

$$X_{k+i\cdot N} = X_k$$

for any integer i. As we'll see below, this symmetry can be exploited to compute the DFT much more quickly.

# DFT to FFT: Exploiting Symmetry

Cooley and Tukey showed that it's possible to divide the DFT computation into two smaller parts. From the definition of the DFT we have:

$$\begin{align}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i~2\pi~k~n~/~N} \tag{1}\\
&= \sum_{m=0}^{N/2 - 1} x_{2m} \cdot e^{-i~2\pi~k~(2m)~/~N} + \sum_{m=0}^{N/2 - 1} x_{2m + 1} \cdot e^{-i~2\pi~k~(2m + 1)~/~N} \tag{2}\\
&= \sum_{m=0}^{N/2 - 1} x_{2m} \cdot e^{-i~2\pi~k~m~/~(N/2)} + e^{-i~2\pi~k~/~N} \sum_{m=0}^{N/2 - 1} x_{2m + 1} \cdot e^{-i~2\pi~k~m~/~(N/2)} \tag{3}
\end{align}$$

We've split the single Discrete Fourier transform into two terms which themselves look very similar to smaller Discrete Fourier Transforms, one on the odd-numbered values, and one on the even-numbered values. So far, however, we haven't saved any computational cycles. Each term consists of $(N/2)*N$ computations, for a total of N^2.

The trick comes in making use of symmetries in each of these terms. Because the range of k is $0 \le k < N$, while the range of n is $0 \le n < M \equiv N/2$, we see from the symmetry properties above that we need only perform half the computations for each sub-problem. Our $\mathcal{O}[N^2]$ computation has become $\mathcal{O}[M^2]$, with M half the size of N.

But there's no reason to stop there: as long as our smaller Fourier transforms have an even-valued M, we can reapply this divide-and-conquer approach, halving the computational cost each time, until our arrays are small enough that the strategy is no longer beneficial. In the asymptotic limit, this recursive approach scales as $\mathcal{O}[N\log N]$.

This recursive algorithm can be implemented very quickly in Python, falling-back on our slow DFT code when the size of the sub-problem becomes suitably small:

In [4]:
```
def FFT(x):
    """A recursive implementation of the 1D Cooley-Tukey FFT"""
    x = np.asarray(x, dtype=float)
    N = x.shape[0]

    if N % 2 > 0:
        raise ValueError("size of x must be a power of 2")
    elif N <= 32:  # this cutoff should be optimized
        return DFT_slow(x)
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = np.exp(-2j * np.pi * np.arange(N) / N)
        return np.concatenate([X_even + factor[:N / 2] * X_odd,
                               X_even + factor[N / 2:] * X_odd])
```

Here we'll do a quick check that our algorithm produces the correct result:

In [5]:
```
x = np.random.random(1024)
np.allclose(FFT(x), np.fft.fft(x))
```

Out[5]:
```
True
```

And we'll time this algorithm against our slow version:

In [6]:
```
%timeit DFT_slow(x)
%timeit FFT(x)
%timeit np.fft.fft(x)
```

```
10 loops, best of 3: 77.6 ms per loop
100 loops, best of 3: 4.07 ms per loop
10000 loops, best of 3: 24.7 µs per loop
```

Our calculation is faster than the naive version by over an order of magnitude! What's more, our recursive algorithm is asymptotically $\mathcal{O}[N \log N]$: we've implemented the Fast Fourier Transform.

Note that we still haven't come close to the speed of the built-in FFT algorithm in numpy, and this is to be expected. The FFTPACK algorithm behind numpy's `fft` is a Fortran implementation which has received years of tweaks and optimizations. Furthermore, our NumPy solution involves both Python-stack recursions and the allocation of many temporary arrays, which adds significant computation time.

A good strategy to speed up code when working with Python/NumPy is to vectorize repeated computations where possible. We can do this, and in the process remove our recursive function calls, and make our Python FFT even more efficient.

# Vectorized Numpy Version

Notice that in the above recursive FFT implementation, at the lowest recursion level we perform $N / 32$ identical matrix-vector products. The efficiency of our algorithm would benefit by computing these matrix-vector products all at once as a single matrix-matrix product. At each subsequent level of recursion, we also perform duplicate operations which can be vectorized. NumPy excels at this sort of operation, and we can make use of that fact to create this vectorized version of the Fast Fourier Transform:

In [7]:
```
def FFT_vectorized(x):
    """A vectorized, non-recursive version of the Cooley-Tukey FFT"""
    x = np.asarray(x, dtype=float)
    N = x.shape[0]

    if np.log2(N) % 1 > 0:
        raise ValueError("size of x must be a power of 2")

    # N_min here is equivalent to the stopping condition above,
    # and should be a power of 2
    N_min = min(N, 32)

    # Perform an O[N^2] DFT on all length-N_min sub-problems at once
    n = np.arange(N_min)
    k = n[:, None]
    M = np.exp(-2j * np.pi * n * k / N_min)
    X = np.dot(M, x.reshape((N_min, -1)))

    # build-up each level of the recursive calculation all at once
    while X.shape[0] < N:
        X_even = X[:, :X.shape[1] / 2]
        X_odd = X[:, X.shape[1] / 2:]
        factor = np.exp(-1j * np.pi * np.arange(X.shape[0])
                        / X.shape[0])[:, None]
        X = np.vstack([X_even + factor * X_odd,
                       X_even - factor * X_odd])

    return X.ravel()
```

Though the algorithm is a bit more opaque, it is simply a rearrangement of the operations used in the recursive version with one exception: we exploit a symmetry in the `factor` computation and construct only half of the array. Again, we'll confirm that our function yields the correct result:

In [8]:
```
x = np.random.random(1024)
np.allclose(FFT_vectorized(x), np.fft.fft(x))
```

Out[8]:
```
True
```

Because our algorithms are becoming much more efficient, we can use a larger array to compare the timings, leaving out `DFT_slow`:

In [9]:
```
x = np.random.random(1024 * 16)
%timeit FFT(x)
%timeit FFT_vectorized(x)
%timeit np.fft.fft(x)
```

```
10 loops, best of 3: 72.8 ms per loop
100 loops, best of 3: 4.11 ms per loop
1000 loops, best of 3: 505 µs per loop
```

We've improved our implementation by another order of magnitude! We're now within about a factor of 10 of the FFTPACK benchmark, using only a couple dozen lines of pure Python + NumPy. Though it's still no match computationally speaking, readibility-wise the Python version is far superior to the FFTPACK source, which you can browse here (http://www.netlib.org/fftpack/fft.c).

So how does FFTPACK attain this last bit of speedup? Well, mainly it's just a matter of detailed bookkeeping. FFTPACK spends a lot of time making sure to reuse any sub-computation that can be reused. Our numpy version still involves an excess of memory allocation and copying; in a low-level language like Fortran it's easier to control and minimize memory use. In addition, the Cooley-Tukey algorithm can be extended to use splits of size other than 2 (what we've implemented here is known as the *radix-2* Cooley-Tukey FFT). Also, other more sophisticated FFT algorithms may be used, including fundamentally distinct approaches based on convolutions (see, e.g. Bluestein's algorithm and Rader's algorithm). The combination of the above extensions and techniques can lead to very fast FFTs even on arrays whose size is not a power of two.

Though the pure-Python functions are probably not useful in practice, I hope they've provided a bit of an intuition into what's going on in the background of FFT-based data analysis. As data scientists, we can make-do with black-box implementations of fundamental tools constructed by our more algorithmically-minded colleagues, but I am a firm believer that the more understanding we have about the low-level algorithms we're applying to our data, the better practitioners we'll be.

This blog post was written entirely in the IPython Notebook. The full notebook can be downloaded here (http://jakevdp.github.io/downloads/notebooks/UnderstandingTheFFT.ipynb), or viewed statically here (http://nbviewer.ipython.org/url/jakevdp.github.io/downloads/notebooks/UnderstandingTheFFT.ipynb) fft (http://jakevdp.github.io/tag/fft.html) tutorial (http://jakevdp.github.io/tag/tutorial.html)

Comments

ALSO ON **PYTHONIC PERAMBULATIONS**

Simulating …

5 years ago · 16 comments

This weekend I found myself in a particularly drawn-out game of …

Optimizing …

8 years ago · 54 comments

Background: The Non-Uniform Fast Fourier Transform¶ The Fast …

The Waiting …

5 years ago · 13 comments

Image Source: Wikipedia License CC-BY-SA 3.0 If you, like me, frequently …

A I

6 ye

Thi
pre
tha

53 Comments 1 Login ▾

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

22 Share Best Newest Oldest

Adam Ginsburg
10 years ago

I liked the article, but I think you really hid the key point in the Cooley-Tukey algorithm: "...we see from the symmetry properties above that we need only perform half the computations for each sub-problem." I think it is important to spell out that you are reducing the number of e^(2 pi i k m / (N/2)) calculations that need to be done.

10 0 Reply • Share ›

asmeurer
10 years ago

It's a bad idea to use i as an integer when you are also using it as sqrt(-1).

10 1 Reply • Share ›

Bill Xia ➦ asmeurer
10 years ago

Yeah, I also found it unreasonable to use i as an integer at the last line of the 3rd part.
Apart from this, the whole article is perfect and easy to understand.

1 0 Reply • Share ›

이성주 Lee Seongjoo
10 years ago edited

What an excellent post. Lots of learning. Thank you. A minor issue with a power of 2 check: np.log2(1024*16) can result in 13.999999999999998 instead of 14.0 for some. Then, the test fails even though logic is faultless. Not Python nor numpy issue. A floating point issue in general (http://floating-point-guid...). N%2 > 0 would be a safer choice avoiding the potential floating point issue.

5 0 Reply • Share ›

lskjdflk → 이성주 Lee Seongjoo
10 years ago

Good point; a standard trick for checking if an integer is a power of two:

N & (N - 1) == 0

True iff N is zero or a power of two.

13 0 Reply • Share ›

jakevdp Mod → lskjdflk
10 years ago

That's an awesome trick. I need to remember that!

8 0 Reply • Share ›

Jebin Matthew → 이성주 Lee Seongjoo
5 years ago

Isn't that used to check for even numbers? That you'd mean 6 is a power of 2, right?

0 0 Reply • Share ›

Ashish → Jebin Matthew
5 years ago

Try it and see - it is only for powers of 2

0 0 Reply • Share ›

Cowlicks → 이성주 Lee Seongjoo
8 years ago

You should use `float.is_integer` see here: https://docs.python.org/2/l...

0 0 Reply • Share ›

jakevdp Mod → Cowlicks
8 years ago

That's fine for Python floats, but as far as I'm aware, no similar routine exists for NumPy floats, which is the context in which this comes up.

1 0 Reply • Share ›

C **charris208**
10 years ago

I prefer a more intuitive approach as opposed to the algebraic. Divide the sample points into lower and upper halves, the even frequency components repeat in both halves, the odd frequency components are the negative in the upper half. That allows the separation of the signal into the even frequency part and odd frequency part. For the even frequency part is only necessary to keep the first half. The odd frequencies can be turned into even frequencies by multiplying by a (complex) frequency of -1 which shifts them all left by one sample in the frequency domain, and only the second half needs to be retained after that. Note the one bit has been reversed after operation as the even frequencies are in the lower half and the odd frequencies are in the upper half. Now repeat on each half.

3 0 Reply • Share ›

SR **SATASHREE ROY**
4 years ago

Very clear explanation about the symmetry of DFT. But there is an issue with the FFT implementation.
The code somehow fails when the input is a sinusoid, for example, x = np.sin(np.pi * np.arange(128)/64). The returned values don't match with those of np.fft.fft
What could possibly be the issue?

1 0 Reply • Share ›

jakevdp Mod → SATASHREE ROY
4 years ago

The DFT of a sinusoid is near zero everywhere but the sinusoid's frequency. I think the disagreement you are seeing is probably within the bounds of floating point precision. You wouldn't see this with non-sinusoidal inputs because the expected result is not near zero.

0 0 Reply • Share ›

F **FATIH**
4 years ago

Hi!

Why you use X.shape[1] at the last part of the article, in place of using X.shape[0] ?

1 0 Reply • Share ›

jakevdp Mod ➜ FATIH
4 years ago

For a 2D matrix X, X.shape[0] refers to the number of rows, while X.shape[1] refers to the number of columns.

1 0 Reply • Share ›

P

pi
5 years ago edited

The notebook won't complete.

return np.concatenate(...

This line is throwing an error: TypeError: slice indices must be integers or None or have an __index__ method

1 0 Reply • Share ›

jakevdp Mod ➜ pi
5 years ago

That's a change in recent versions of numpy & python. Change ``N / 2`` to ``N // 2`` to make it to work.

1 0 Reply • Share ›

E

Erin Wiles
6 years ago

How are you scaling the outcome of these FFT and FFT_vectorized definitions?

The DFT can be scaled by 2/N, where N is the number of samples. (This scale is the same for the Numpy FFT).

When I implement FFT and FFT_vectorized examples in Python and test is with a sinusoid input, x=sin(2*pi*f*t). The scale has a relationship to the frequency, f. if f=1 Hz, they are scaled by 2/pi. Then as f inrease by 1 Hz, the pattern is 2/(pi*f) or 2/(pi*k). This result is puzzling to me.

Has anyone else tried scaling them?

1 0 Reply • Share ›

E

Erin Wiles Erin Wiles

6 years ago

I think I spot the bug, the factor needs to be the same k for each frequency bin. I am not sure how to fix it but I will try.

0 0 Reply • Share ›

W

William Lane

9 years ago edited

How would you go about returning the iFFT? I've tried reversing the exponent sign and dividing by the numer of elements, but it is off (not close enough for np.isclose). Any tips?

1 0 Reply • Share ›

W

William Lane William Lane

9 years ago

I realized that my problem was dividing by the number of elements. If you're not careful and simply modify the above algorithm by including .../N, it will **recursively** divide by N. Instead, you want it to only divide by N on the first level (no recursion). I got around this by giving ifft a second argument: ifft(X, norm=True). Inside the function, if recursion was used then the False variable was passed, e.g., ifft(X[::2], False), and at the end was a simple check: if norm is True: x /= N; return(x).

1 0 Reply • Share ›

Quim Llimona

10 years ago

Is the vectorized execution time correct? It looks like it profiled FFT_slow, FFT and FFT_vectorized, instead of FFT, FFT_vectorized and numpy's native code.

1 0 Reply • Share ›

이성주 Lee Seongjoo

Quim Llimona

10 years ago edited

I was also somewhat confused but it seems right. The number of x elements is increased from 1024 to 1024*16 resulting in increased FFT execution time from 4.07 ms to 72.8 ms. It is indeed an order of magnitude improvement.

1 0 Reply • Share ›

Quim Llimona
→ 이성주 Lee Seongjoo
10 years ago edited

That's right! Quite confusing, indeed... Especially because the running time with the long signal happens to be the same as with the short signal using the next slower algorithm. What a coincidence!

0 0 Reply • Share ›

E
emre
a year ago

is there a way to code this same algorithm using for loops instead of recursion? if so, can you walk me through how to make one?

0 0 Reply • Share ›

Tianyi Wang
3 years ago

Thank you for the detailed illustration of the derivation process of all the math formula. That' s a really big help. Otherwise, without knowing exp[2π i n]=1, I have no idea about how the conclusion was drawn.

0 0 Reply • Share ›

F. S. Farimani
4 years ago

I implemented 3Blue1Brown's description of Fourier transform. would love to have your feedback.

0 0 Reply • Share ›

J
John
4 years ago

I dont understand why the algorithm reduces the complexity from O(N^2) to O(NlogN). Even if calculation of X_k is divided recursively into two sub problems, we still need to compute O(N) operations. Since do the similar computation for each X_k (k = 1, ..., N), the overall complexity is still O(N^2). I guess some computations are recursively reduced by half. What exactly are those?? Those things are not described in this post.

0 0 Reply • Share ›

U

UTN

4 years ago

What would be the explanation for the use of np.concatenate([X_even + factor[:N / 2] * X_odd, X_even + factor[N / 2:] * X_odd]) ?

0 0 Reply • Share ›

jakevdp Mod ➜ UTN

4 years ago

It implements the formula written above it. Because of aliasing, the Fourier transform of the even and odd pieces repeat themselves above k=N/2

0 0 Reply • Share ›

SR

SATASHREE ROY ➜ jakevdp

4 years ago

Please elaborate more. Why not simply use X_even + factor * X_odd ?

0 0 Reply • Share ›

jakevdp Mod ➜ SATASHREE ROY

4 years ago

That would result in a ValueError, because X_even and factor are different lengths. It would be possible to instead write

X_even = np.concatenate([X_even, X_even])
X_odd = np.concatenate([X_odd, X_odd])
return X_even + factor * X_odd

This would return the correct result as well. Is that what you had in mind?

0 0 Reply • Share ›

S

Scoodood ➜ UTN

4 years ago

Same question here. Please elaborate more.

0 0 Reply • Share ›

Themos

4 years ago

The best way I heard it explained goes like "some dense matrices break up ("factorise") into (a small number of) very sparse matrices". M, above, is one of those dense matrices so it is much faster to compute its action on a vector by multiplying by each of the very sparse factors.

0 0 Reply • Share ›

J

Johan

4 years ago

If you change the random input to `np.sin(np.pi * np.arange(128)/64)` suddenly the np.allclose() evaluates to false. Can anyone explain me why this is the case?

0 0 Reply • Share ›

nsb64

5 years ago

Hi Jake,
First of all, neat code!
Could you answer an obvious question: why have you resorted to DFT_slow for N<=32? It sure reduces the unnecessary recursions for small values, but does it reduce the speed?
Also, if I were to take an array of length 2**12, would you resort to DFT_slow for N<=64?

Thanks,
Nisarg

0 0 Reply • Share ›

jakevdp Mod ➜ nsb64

5 years ago

For arrays smaller than some number, the "fast" splitting procedure becomes slower than a simple direct approach in terms of wall-clock time. Where it becomes slower will depend on the details implementation... I chose 32 because it seemed reasonable, and I wasn't interested in doing detailed benchmarks to choose the number more carefully. You could switch that N to any number you'd like... if it's performance critical I'd suggest doing some benchmarks to choose that N optimally.

1 0 Reply • Share ›

nsb64 ➜ jakevdp
5 years ago

No, I needn't worry about any benchmarks, this code works well for me. Was simply wondering about choice of N = 32, which seems reasonable after all.

0 0 Reply • Share ›

Affa A.
6 years ago

Great post and examples!

Appreciate ;-)

0 0 Reply • Share ›

T **Tyler Matthew Harris**
6 years ago

This is great. Thank you

0 0 Reply • Share ›

ThePun Isher
6 years ago

Excelent and useful post! Thanks!

0 0 Reply • Share ›

Nhat Bui
7 years ago

Thank you for this, wished I came across it in undergrad so I can get an intuition for FFT sooner

0 0 Reply • Share ›

Sukoreno Mukti
7 years ago edited

can you explain this ?
# N_min here is equivalent to the stopping condition above,
# and should be a power of 2
N_min = min(N, 32)

and

elif N <= 32: