Fast Fourier Transform

Andrew S. Barr

December 17, 2010

1 Introduction

It is not uncommon to hear that the most utilized algorithm of the computer age is the Fast Fourier Transform (FFT). The FFT is an extremely efficient way to perform a Discrete Fourier Transform (DFT). The DFT is an invertible transform that takes a set of data in time or space and decomposes it into a representation that corresponds to what we think of as frequency in time. As an array of length N can be seen as inhabiting a vector space of dimension N, the FFT does nothing more than perform a change of basis on [illegible] array. In the case of the FFT, the signal, which can be complex valued and thus living on $\mathbb{C}^N$, is projected onto a basis called the ω basis by means of the following inner-product: $\langle \vec{z}, \vec{w} \rangle_N = \frac{1}{N}\sum_{m=0}^{N-1} z_m \cdot \overline{w_m}$. The elements of the ω basis vectors are taken from the set of the N roots of unity, generated by $(\vec{w_k})_\ell = \left(e^{i\cdot 2\pi/N}\right)^{k\ell} = \omega_N^{k\ell}$ where k is the basis index and l is the ω index. Therefore, the magnitude of all these vector elements is 1. The principal quantity that defines the frequency of each vector is the argument or phase of the complex elements in each ω[1]. Observe the generation of the set w_4 below.

Generation of the $\vec{w_{0-3}}$ Basis Vectors

$$\vec{w}_0 = \left(\left[\omega_4^0\right]^0, \left[\omega_4^0\right]^1, \left[\omega_4^0\right]^2, \left[\omega_4^0\right]^3\right) = (1,1,1,1) \quad \Delta\text{Arg(z)} = 2\pi \equiv 0$$

$$\vec{w}_1 = \left(\left[\omega_4^1\right]^0, \left[\omega_4^1\right]^1, \left[\omega_4^1\right]^2, \left[\omega_4^1\right]^3\right) = (1,i,-1,-i) \quad \Delta\text{Arg(z)} = \tfrac{\pi}{2}$$

$$\vec{w}_2 = \left(\left[\omega_4^2\right]^0, \left[\omega_4^2\right]^1, \left[\omega_4^2\right]^2, \left[\omega_4^2\right]^3\right) = (1,-1,1,-1) \quad \Delta\text{Arg(z)} = \pi \equiv -\pi$$

$$\vec{w}_3 = \left(\left[\omega_4^3\right]^0, \left[\omega_4^3\right]^1, \left[\omega_4^3\right]^2, \left[\omega_4^3\right]^3\right) = (1,-i,-1,i) \quad \Delta\text{Arg(z)} = -\tfrac{\pi}{2}$$

(chart taken from 'Wavelets Made Easy' by Yves Nievergelt [1])

The elements of $\vec{w}_0$ are all 1 and are seperated in phase by 0 or equivalently 2π. This is sometimes called the "0-frequency" or the DC value since when our inner product is taken into account, one sees that its coefficient encodes the average value of the array. Its analog in the continuous Fourier series would be the [illegible](0) or e^0 term. The ΔArg(z) of $\vec{w}_1$ is $\frac{\pi}{2}$ while the ΔArg(z) of $\vec{w}_3$ is $\frac{-\pi}{2}$. These can be seen as the frequency corresponding to changes over 1 element. The ΔArg(z) of $\vec{w}_2$ is π and by its alternation to two values one can easily extrapolate its oscilation over 2 elements, or half of the array. This is all a result of the

[1]see `http://480.sagenb.org/pub/53` for an example of $N = 8$

cyclical nature of the complex exponentional and will be used when we consider the FFT algorithm. Though it may be hard to see with a low N example, there certainly exists a symmetry in the $\Delta\text{Arg(z)}$ that becomes clearer with higher values of N and makes it easier for one to understand the eventual shuffling of the data and why people who use FFTs often change the indexing of the transformed data.

The DFT is then accompanied by applying our inner-product to an arbitrary decomposition to resolve the weights of each of the basis vectors. This operation naively requires N complex multiplies and N complex adds for each of the N vectors, making the DFT an $O(N^2)$ operation. What the FFT does is make use of the symmetry demonstrated above to significantly reduce the computational complexity and produce the transform in $O(N \log N)$ time.

2 Basic FFT Theory

The FFT can be derived from factorizing a set of $\vec{w}_k^l$ by applying properties of the complex exponential. It is nearly universal among the literature on FFTs to denote $W_N = e^{-\frac{i2\pi}{N}}$. There are three identities I am including that I think will help the reader understand the mechanics behind how the FFT works.

$$
\begin{aligned}
W_N^2 &= e^{-\dfrac{i2\pi}{N}2} = e^{-\frac{i2\pi}{\frac{N}{2}}} = W_{\frac{N}{2}} \\
W_N^N &= e^{-\frac{i2\pi}{N}N} = e^{-i2\pi} = 1 \\
W_N^{\frac{N}{2}} &= e^{-\frac{i2\pi}{N}\frac{N}{2}} = e^{-i\pi} = -1
\end{aligned}
$$

This derivation from Andrew A. Lamb's paper, "Factoring the DFT: Derivation of the FFT Algorithm".

When we generate the set of $\vec{w}$, we can put them in a matrix, indexed exactly like in the above example with $N = 4$ and attempt a factorization. A simpler and equivalent action is to write out the formula for a DFT in a summation and then split that sum into even and odd parts.

Let $X_k = \langle \vec{z}, \vec{w} \rangle_N = \frac{1}{N}\sum_{m=0}^{N-1} x_m e^{-ikm\frac{2\pi}{N}}$ denote the DFT operation.

Then, split it into even and odd sums.

$$
\begin{aligned}
&\frac{1}{N}\sum_{m=0}^{N-1} x_n e^{-ikm\frac{2\pi}{N}} = \\
&\qquad \frac{1}{N}\sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-ik(2m)\frac{2\pi}{N}} + \frac{1}{N}\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-ik(2m+1)\frac{2\pi}{N}} \qquad (1)
\end{aligned}
$$

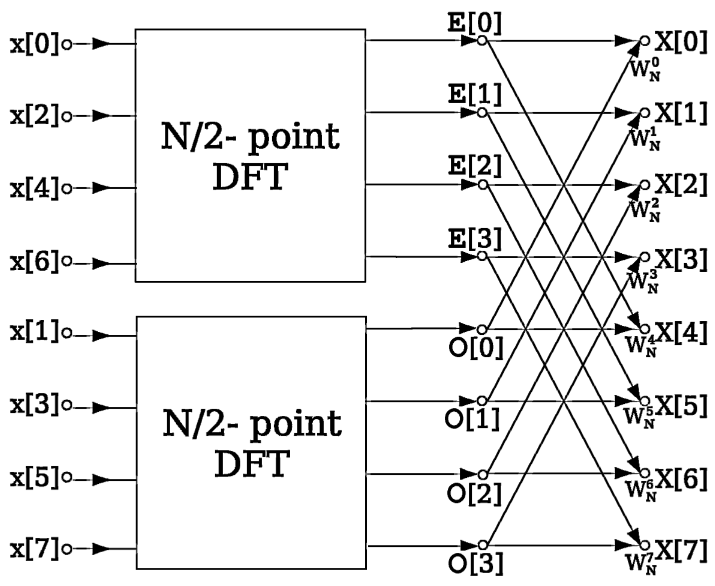

Figure 1: Butterfly diagram for length 8 FFT

Factor the exponential.

$$\frac{1}{2}\cdot\frac{1}{\frac{N}{2}}\sum_{m=0}^{\frac{N}{2}-1} x_{2m}e^{-ikm\frac{2\pi}{\frac{N}{2}}}+\frac{1}{2}\cdot\frac{1}{\frac{N}{2}}\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1}e^{-ikm\frac{2\pi}{\frac{N}{2}}}\cdot e^{\frac{-ik\pi}{\frac{N}{2}}} \tag{2}$$

These two equations are in the form of a DFT for lists of size $\frac{N}{2}$ with the addition of a constant in the odd term, referred to as a 'twiddle factor'. The splitting process can be applied recursively, until the case for which the X_k is of length 2. At that point, since the two roots of unity are 1 and (-1), we are left with an addition and a subtraction. The organization of the operations is normally represented graphically by butterfly diagrams like the one shown in Figure 1.

Note that a downward arrow indicates a subtraction of that element. The shuffling of the elements is a natural result of the continued recursive splitting and can be equivalently described by taking the binary representation of each index, reversing that, and then grouping in twos to generate the bottom level of the butterfly diagram.

3 List of Arbitrary Length

While I only mentioned the case of an array of length $N = 2^k$, there exist algorithms, referred to as 'mixed-radix FFTs', which can be used to calculate the FFT of an array of arbitrary length. These will essentially factor an FFT of length $N \cdot M$ into N FFTs of length M or vice-versa. The general workings of these FFTs are the same but with changes to the 'twiddle factors' and of course the mechanics of the shuffling. An important thing to consider is that this decomposition is not as fast as the $N = 2^k$ case. In the past, due to hardware limitations that have mostly disappeared by 2010, it was faster to compute the transform by simply zero padding the array and using the standard radix-2 transform. However, this is not always an acceptable option, due to limitations of size, processing, and a user not wanting to consider normalizing the scale of the frequencies. A step by step example of a length 21 FFT can be found at `http://www.engineeringproductivitytools.com/stuff/T0001/PT07.HTM`. The author of that page brings up the point that for large and multidimensional cases, the zero padding becomes increasingly inefficient. For example, if one had a 400 x 800 pixel image (stored as a two dimensional array), an algorithm requiring 2^k length inputs would require a conversion into a 512 x 1024 pixel image, leading to a storage requirement 1.6 times greater than the original allocated amount, not to mention more calculations, many of which are multiplies by zero. A worst case for this type of naive implementation would approach 4 times the storage space as a result of zero padding (257 x 513 being padded to 512 x 1024) and require over twice as many operations, many of which are trivial. All of the most optimized and modern general-use implementations use solved base cases of low prime number FFTs to perform the mixed-radix routines. However, for an implementation to work purely by mixed-radix routines, it would need to have solutions for every array of length(array) = prime. For the case of large primes, there exists Rader's FFT algorithm which uses a convolution, but is still able to solve FFTs in approximately $O(N \log N)$ time (in practice, taking a few times longer than an FFT of a comparable but more factorable length). One thing that the reader should note if interested in studying hardware or embedded implementations of FFTs is that it has been a standard practice in electrical engineering to pad data with zeros to reach length $2^k, k \in \mathbb{N}$ or to simply design systems that take in data with the 2^k restriction.

4 Optimizations

Apart from the reduction of complexity from $O(N^2)$ to $O(N \log N)$, there are a number of ways to speed up the evaluation of the FFT. For example, an FFT of a purely real input will be Hermitian symmetric so we can get a factor of 2 speed increase by stopping calculations after obtaining half of the FFT. In many implementations there are command flags, different method calls, or options to detect and take care of this. Parallel FFTs have been researched and implemented, though their use is of limited value in the 1-dimensional case. While it is tempting to think that the divide and conquer nature and butterfly structure of the FFT will lend itself to parallelizability, there are not many operations to be performed at each stage in the algorithm (two adds and two complex multiplies in the radix-2 case) so the extra overhead that comes from the splitting and messaging between threads ends up negating time savings. The reason that the 1-dimensional FFT makes little sense to parallelize is that every term in the input contributes to each term in the output, though parallelized algorithms do certainly exist[2]. Parallel FFT algorithms become useful in the multidimensional case, which are performed by taking one dimensional FFTs along each axis, for each dimension. For example, a 10 x 10 array will have its FFT taken by performing 10 FFTs in one dimension, and then doing the same along the other dimension. These 10 FFTs in each dimension are unrelated so there are no race-conditions to worry about. Thus we can certainly thread this procedure and make each thread perform an FFT, an option that is available in FFTW and FFTPACK. There have been many papers and theses on parallel FFTs and one can find a number of implementations using MPI and OMP libraries available for those interested in parallel code. For those who work in image processing and particle simulation, in-place algorithms have been developed to control memory use, though with indexing schemes that can be much more complex. The inherent recursion in the FFT algorithm can also be optimized, depending on the language used for the implementation and the architecture of the machine. Many of the papers regarding the optimization of recursion in an FFT deal with caching and memory management issues. For small base cases, there are also iterative solutions (see the FFTW section for more details) though one will not find anyone advocating an iterative solution for a large case.

5 Using the FFT

The FFT has an enormous range of applications that span a variety of fields. It has found an especially large amount of use from electrical engineers. For them, many operations are simplified by directly manipulating the frequency components of a signal. Since the transform of a signal into its frequency components is easily and quickly invertible, it is easy to obtain the effect of that manipulation in the time domain. Electrical engineers are particularly interested in two operations that are difficult in the time domain but simple in the frequency domain, designing filters and performing convolutions. Since we often care about isolating a certain frequency component of a signal, a filter would simply be an array that stores weights of how much of each frequency it would allow to pass. This is applicable to both the discrete and continuous cases but since in practice we almost always deal with discrete data (like an mp3 sampled at 192kHz), we here limit our concern to the discrete $\vec{w}$ decomposition provided by the FFT. We could thus take the FFT of a sound, perform a pointwise multiplication with our filter, and then take the inverse transform to return to the time domain where we are left with a filtered sound. This is something that is very easy to do, both conceptually and computationally. If we were limited to working in the time domain, to analyse the effect of a system on an input signal, we would have to perform a conceptually and computationally more difficult convolution of the input signal with the transfer function of the system to find the output of the system (a transfer function summarizes the relation of a system's input to output). The convolution theorem establishes the equivalence of multiplication in frequency to convolution in time, making the FFT extremely useful in signal analysis[3].

[2] `http://beige.ucs.indiana.edu/B673/node16.html`

[3] In this case, time could just as easily be spaced, and similar ideas are used to make image filters. Zeroing out the large frequency coefficients of an image would lead to an edge detecting filter, while zeroing

When using an FFT implementation, it is extremely important to read the documentation as conventions vary on both the actual transform and the form of output of the data. A common ocurrence with the continous Fourier transform is that someone's forward transform could be someone else's backwards transform. In the discrete case, the positive exponential for the generating vector $W_N = e^{-\frac{i2\pi}{N}}$ is nearly universal for the FFT. There is a scale factor that comes out of the transform so one could have a problem if care is not taken. Some algorithms like FFTW scale the result so one could transform back and forth without a loss of data. While the standard DFT and FFT algorithms provide for a particularly ordered output, many people prefer to 'center' their output by changing the location of the DC-frequency to correspond to the center. From there, many often take the magnitude of the frequency data since the magnitude may be what we are sensitive to (like sound and light). Another important consideration comes from a result in general information theory which relates the rate of sampling to attainable detection of frequency components (see the Nyquist-Shannon Sampling Theorem). The Nyquist rate puts a bound based on the input array's bandwidth (the width of nonzero elements in the frequency domain) of the frequency components one can detect as a function of the sampling rate. In order to perfectly resolve a frequency, one must sample at twice the rate of the frequency to avoid something called aliasing. Aliasing is extremely undesired and is the signal processing equivalent to a medical false positive. It leads to detection of something that is not actually there, such as moire patterns in images or the wagon-wheel effect in video. Our decomposition into basis vectors was done without ambiguity so it seems strange to think that there could be problems with our obtained frequencies. The important insight is that the Nyquist rate extends in the negative direction and so FFT actually returns a range of frequencies from the negative Nyquist frequency to the positive Nyquist frequency. One should also not be alarmed if the result of a transform of a purely real signal results in a transform with complex coefficients. This is a natural result of the choice of basis and signifies modulation or shift in the time domain.

6 FFTW: The Fastest Fourier Transform in the West

No paper on FFT implementations would be complete without mentioning FFTW, the Fastest Fourier Transform in the West. FFTW was developed by Matteo Frigo and Steven Johnson at MIT nearly 15 years ago. It is totally open under the GPL[4] for use in free software, only requiring a license for use in non-GPL software. Its use is extremely widespread and the website has interesting and well-written papers describing the use and development of FFTW[5]. The FFTW is a set of C libraries that work to generate very fast code to calculate FFTs of an arbitrary sized, arbitrary dimension array. In order to do that, it factors an array (recall the discussion on mixed-radix FFTs) and then attacks the FFT a

the small frequency coefficients will lead to a blurring. See `sites.google.com/a/iupr.com/ipiu-course/sage-notebooks` for image processing examples using Sage.

[4]`http://www.fftw.org/fftw2_doc/fftw_8.html`

[5]http://www.fftw.org/fftw-paper-ieee.pdf

few different ways to see which approach performs best. While the factorization of a number is certainly unique, FFTW uses its ‘planner’ to generate multiple plans for a given input, attempting optimizations based on the ”’shape’ of the input data”[2] which it will then internally compare and choose. To run quickly for a large class of input data, FFTW has a large number of resolved and optimized base cases (it lists cases 1-16,32, and 64 for complex transforms and 2,3,5,7,11, and 13 for purely real transforms) before having to appeal to Rader’s arbitrary length FFT algorithm. One who reads FFTW’s release notes will see a number of fixes indicating past bugs involving prime factors as small as 19, where FFTW was generating incorrect output.

When FFTW generates a ’plan’ for an FFT, it uses ’codelets’, corresponding to the aforementioned cases, which are essentially direct solutions implemented in iterative code. The benefit to generating this plan is that it is reusable, so in the common case where one would be taking many FFTs of the same length, we can have a net speed up after the time-consuming plan generation. The second level of optimization is specific to the machine and involves the generation of SIMD[6] instructions. I do not know much about this but it is well documented on FFTW’s website.

FFTW supports a large variety of transforms including multidimension and parallel transforms and does everyt[illegible] extremely quickly. However, its use can be difficult and one would have to spend a bit of going over the examples to use it effectively as opposed to NumPy where the most simple operations are straightforward. Thankfully, FFTW is very well documented and it is easy to [illegible]amples online to get started. Support is also available from many users and it is not uncommon to see one of the FFTW authors, Steven Johnson, replying to questions on forums.

7 FFTPACK

FFTPACK is a highly optimized and widely used library developed by NCAR (National Center for Atmospheric Research) that has been around for over 20 years. It is a set of FORTRAN subroutines that are in the category of being essentially done (the latest update by NCAR was written in 1995) yet continues to be ported to other languages, notably NumPy/SciPy. It is very much open, with its unique license on the site[7] giving permission for anyone to use it, as long as they don’t make any claims that NCAR endorses their product. The openness, combined with high performance, good documentation, and its early creation made it the first widely adopted implementation and remains used by many, in its original form and in NumPy/Scipy.

[6]Single Instruction, Multiple Data. These instructions are for hardware level optimizations.

[7]`http://www.cisl.ucar.edu/css/software/fftpack5/ftpk.html`

8 NumPy/SciPy

The FFT can be applied to any vector object in NumPy and SciPy via the fft routine. It includes transformations for 1,2, and n-dimensional FFTs. In addition, it allows for the user to call on special methods for real and Hermitian symmetric inputs. One thing that makes using the NumPy version easy ~~to use~~ are the numerous operations attached to the vector object. Being implemented in python also lowers the barriers to entry low and easy for those who are using the same vectors with another python library or Sage. The documentation for the NumPy/SciPy FFTs are well written and include examples for each type of FFT available in the libraries. Considering that the NumPy/SciPy fft is just a porting of FFTPACK from FORTRAN, it makes it very easy for one to obtain a high speed FFT in a high level language and is my preferred FFT to use from Sage.

9 GSL: Sage

GSL (Gnu Scientific Library), a C library available in Sage, was developed by number of physicists, originating with the work of Mark Galassi and James Theiler of Los Alamos National Laboratory. From its conception in 1996 it was designed to be a powerful, free, and open library for scientists to handle numerical calculations. Its implementation of the FFT is essentially the standard version used in Sage (called by using FastFourierTransform). Using it is sometimes awkard since it creates an object for which the forward and backward transforms are methods. The data are stored as tuples (before the C99 complex type that was a common way to store complex numbers), and their immutability makes desired operations on a transformed array difficult. To manipulate the data, there is a set(i,j) method, but it is not as easy as working with lists or NumPy vectors. On the other hand, the GSL implementation is fast and, like all modern implementations, has mixed-radix routines available to compute the transform of arbitrary length arrays. Something that has made its use difficult in Sage is the very minimal documentation and lack of substantive examples. I have searched quite a bit for use of the GSL FFT and have found little to suggest that it is widely used. Even the GSL documentation suggests using FFTW for more involved applications. According to the GSL documentation, the mixed-radix routines of the GSL FFT are actually a ”reimplementation of the FFT routines in the Fortran FFTPACK library” [3]. There is no such note for the radix-2 implementation so I assume that that part is unique to GSL, but unless length is controlled by the user, it is likely that the vast majority of FFTs will call the mixed-radix versions. The GSL implementation features a ”ComplexWavetable” class that is a ”trigonometric look up table for a complex FFT of length n”[8] to reduce redundant calculations. This is a feature that I have not found in any part of FFTPACK so I assume that it is original to GSL.

[8]Source: `http://rb-gsl.rubyforge.org/files/rdoc/fft_rdoc.html#2.3.1`

10 Benchmarks

Due to the differences of implementations, there are many things to take into account if one desires to create an accurate and meaningful benchmark. For example, FFTW creates incredibly fast 'plans' for doing an FFT of a particular length, but if you only need to do 1 FFT calculation, the generation of the 'plan' and associated code could take more time than running the FFT using some other implementation. In addition to that, the source for FFTW consists of 988 files and takes nearly 16 Megabytes of space[9]. Even with this taken into account, the creators of FFTW expect "most of the [other] FFTs to slip quietly into oblivion"[2]. But if the FFTW was the fastest code for every possible case, then why would people even bother with other implementations? The FFTW website has a detailed section[10] where they do many benchmarks with an extensive amount of cases on a number of different machines. They even have something you can download called BenchFFT[11] that will run the tests on your personal machine. Their conclusion, and the consensus of those who have to calculate many FFTs, particularly large, parallel, and/or multidimensional is that the FFTW, in general, can be considered to be the fastest implementation.

11 Conclusion

As of now, if one wants to use Sage for the calculation of large and/or many FFTs, it is possible to build Sage with FFTW. However, for most casual and non large-scale use, the NumPy implementation is sufficiently fast and easy to work with. Something that I am very interested in developing is an open-source digital-signal/image processing package to be usable specifically with Sage. There are libraries such as Python Imaging Library (PIL), PyWavelets, and Encortex (for digital signal processing written in C) available, but it would be nice to be able to use the functionality of each library on common objects. This could take advantage of things like fast large number and matrix multiplication, plotting, and list operations in Sage. I would certainly add a number of helper methods to enhance the use of the numpy FFT so that a user could easily perform common signal operations like modulation and convolution with another signal. I think this could lower the barrier to entry to signal processing work since it would not only be free, but also much easier to use than Matlab, whose use is often taught to undergraduates since is use is widespread in industry. One thing that I have learned from studying various libraries is that the key for a successful implementation is not only performance, but openness and quality of documentation and support. For example, getting FFTW to run as desired is not straightforward but is extremely well documented and thus not difficult to find help. I hope that I can apply what I have learned from studying implementations to create a useful library where one can apply

[9]This storage requirement may be negligible for a modern personal computer but for some engineering applications involving microcomputing it is an important consideration.

[10]`http://www.fftw.org/speed`

[11]`http://www.fftw.org/benchmark/benchmark.html`

these open algorithms to make it easier to study and apply principles of digital and image processing.

References

[1] Nievergelt, Yves. ”Wavelets Made Easy,”, Birkhuser; 1999.

[2] Matteo Frigo and Steven G. Johnson, ”The Design and Implementation of FFTW3,” Proceedings of the IEEE 93 (2), 216-231 (2005). Invited paper, Special Isue on Program Generation, Optimization, and Platform Adaptation. Accessed 17-12-2010. Available: http://www.fftw.org/fftw-paper-ieee.pdf

[3] Mark Galassi and James Theiler. ”The GSL Reference Manual: Fast Fourier Transforms (FFTs)”. Accessed 17-12-2010. Available: http://www.gnu.org/software/gsl/manual/html_node/Fast-Fourier-Transforms.

[4] ”Discrete Fourier Transform (numpy.fft)”. Accessed 12-17-2010 Available: http://docs.scipy.org/doc/numpy/reference/routinces.fft.html

[5] Swarztrauber, Paul. ”FFTPACK5”. University Corporation for Atmospheric Research. Accessed 17-12-2010. Available: http://www.cisl.ucar.edu/css/software/fftpack5