Sparse Fourier transform Final Project Report

Andrii Hlyvko
Rutgers University
ah619@scarletmail.rutgers.edu

Wentao Zhu
Rutgers University
wz220@scarletmail.rutgers.edu

Abstract

One of the most important algorithms in signal processing is the Fast Fourier transform, which allows us to compute the DFT of the input signal in $O(NlogN)$ time. With the rise of big data we often need to process terabytes of data in very short time. Thus, computing the DFT of big amount of data in very short time is critical in many applications. Very often our input signal contains very few frequencies so we can approximate the DFT using subsampled input of small length. The goal of this project is to implement the sparse Fourier transform, which approximates the DFT in $O(KlogN)$ where $K << N$.

1. Introduction

The emergence of big data problem make FFT no longer fast enough and furthermore, in many applications it is hard to acquire a sufficient amount of data to compute the desired Fourier transform.So we switched our attention to signal itself and found that lots of signal are sparse which means in frequency domain most points' value is zero except for a small fragment of points.Base on this quality of the signal, sparse Fourier transform derived.In this algorithm we only use small subset of input data.

2. Theory

The most common case for a signal is that its spectrum is dominated by several non-zero frequencies and for some spectra of the frequencies are really close to each other.What we need to do is to reduce this problem to several subproblem involving a single non-zero frequency.SFT is a kind of algorithm that can sense the position of the non-zero frequency points,(the position and the value of the nonzero frequency can be found using only two samples of the pure signal).The key is we use certain rule ($\Gamma^{-1}(\cdot)$) to group the spectrum of the original vector into a bin(total number is B which should be far smaller than the original number N)and each bin correspond to frequency band .Since the frequency domain is sparse,it would be very likely for each non-zero frequency point to stay in their own bin.We add up point in each bin to switch the N-long sequence to a short B-length sequence and do the FFT and according to the results of our computation we will ignore all the bin which do not contain the non-zero frequency point.And we design a method to reconstruct the frequency of original N point.

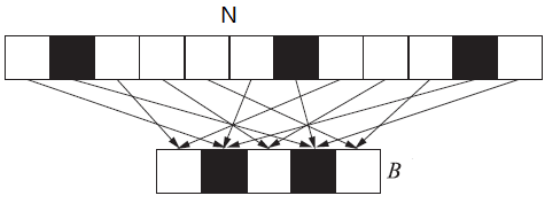


Fig. 1 Representation of the binning process

We use filter to divide 'N' bins to 'B' bins.To avoid multiple non-zero points appear in the same 'bin' since it could be possible that spectra in which two nonzero frequencies are very close to each other.First we need to do is *randomly binning the frequency* so that non-zero frequencies could be separated.Then we use a filter which bandwidth is N/B to *isolate the frequency* .The convolution of frequency domain can make each parts superposed .Finally we down sample the frequency domain at the frequency of N/B.Then we suppose to get all non-zero frequency .We can get all the position and value of non-zero frequency through the FFT of B.At last ,we can use *simple-frequency recovery* to get the original position of each non-zero frequency .

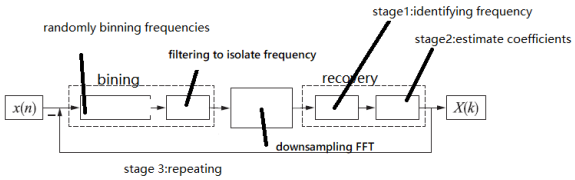


Fig. 2 Theoretical framework of SFT

2.1. The Prototypical SFT

Sparse FFTs improve on the runtime of traditional FFTs for such Fourier sparse signals by focusing exclusively on identifying energetic frequencies, and then estimating their Fourier coefficients. This allows sparse FFTs to avoid "wasting time" computing the Fourier coefficients of many insignificant frequencies. Although several different sparse FFT variants exist, they generally share a common three-stage approach to computing the sparse DFT of a vector: Briefly put, all sparse FFTs (repeatedly) perform some version of the three following steps:
step 1) identification of frequencies whose Fourier coefficients are large in magnitude (typically a randomized subroutine). Roughly speaking, stage 1 works by randomly binning the Fourier coefficients of f into a small number of "bins" (i.e., frequency bands), and then performing a single frequency recovery procedure (as described in the section "Phase Encoding") within each bin to find any energetic frequencies that may have been isolated there.
step 2) accurate estimation of the Fourier coefficients of the frequencies identified in the first step Recall that stage 2 involves estimating the Fourier coefficient of each frequency ω identified during stage 1 of the sparse FFT.
step 3) subtraction of the contribution of the partial Fourier representation computed by the first two steps from the entries of f before any subsequent repetitions.

2.2. Randomly binning frequencies

Just the simple use two basic properties of FFT,we can get (2) from (1) and get (4) from (3).

$$x_1(n) = x(n)W_N^{bn} \quad \text{(the scaling property)} \quad (1)$$
$$x_1(k) = x(k+b) \quad (2)$$
$$x_2(n) = x(\sigma n) \quad \text{(The modulation property)} \quad (3)$$
$$x_2(k) = x_2(\sigma^{-1}k) \quad (4)$$

σ^{-1}means the number-theoretic reciprocal of modulo N.And this is the concept of Chinese remainder theorem.According to the concept of complete system of residues,$\sigma(k-b)$ mod N still has completeness.And $\sigma(k-b)$ mod N is the new position of original K.

2.3. Filtering to isolate frequency

Since each repetitive or iterative cycle involves multiplying the filter, in order to improve the efficiency of the algorithm, the time domain and the frequency domain of the filter are required to be sparse, that is, the effective length of the time domain is only a small part, Can be ignored outside.

After we use the techniques for filtering the general vector,what we need to do now is to do the *single frequency recovery*.Some techniques are mentioned below.And we assume the length of the signal N is integer power of 2 .If not ,we add zero signal to make it be.

2.4. Phase encoding

It is derived from the OFDM trick.Suppose the single frequency signal $x(n) = x(\omega)e^{j2\pi n\omega/N}$, $\omega \in [0, N-1]$,$x(n+1)/x(n) = e^{j2\pi\omega/N}$So we can easily get .Although fast, this straightforward technique is not generally very robust to noise.Consider$x(n) = x(\omega)e^{j2\pi n\omega/N} + \epsilon_j$,when we do x(n+1)/x(n) again we could not expect to get accurate w unless ϵ_j is far less than x(w)/N .

2.5. Binary search technique

For simple signal with containing noise,this technique has better robust comparing to phase encoding whose time complex is O(klbN NlbN/k).For example ,let us set $x(n) = x(\omega)e^{j2\pi\omega/N}$,$\omega \in [0,7]$,let us randomly pick $n \in [0,7]$. consider

$$|e^{j2\pi\omega/8} - i| < |e^{j2\pi\omega/8} - (-i)| \quad (5)$$
$$|e^{j2\pi\omega/8} - 1| < |e^{j2\pi\omega/8} - (-1)| \quad (6)$$

when we multiply —x(n)—on both sides of both equation we can get

$$|x(n+1) - ix(n)| < |x(n+1) + ix(n)| \quad (7)$$
$$|x(n+1) - x(n)| < |x(n+1) + x(n)| \quad (8)$$

Then we know which quadrant is the frequency in. Then we set signal $x'(n) = x(2n)e^{-j2\pi 4\times 2n/8} = x(\omega)e^{j2\pi(\omega-4)n/4}, n \in [0,3]$; consider the picture below

$$|x'(n+1) - ix'(n)| < |x'(n+1) + ix'(n)| \quad (9)$$
$$|x'(n+1) - x'(n)| < |x'(n+1) + x'(n)| \quad (10)$$

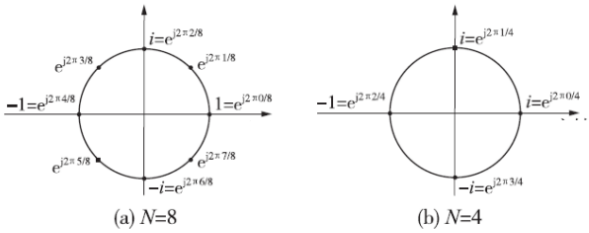
if (10) is true and (11) is not true ,then $\omega \in 1,2$ belongs to the second quadrant. Then we make

$$x"(n) = x'(2n)e^{-j2\pi 1\times 2n/4} = x(\omega)e^{j2\pi(\omega-5)n/2}, n \in [0,1] \quad (11)$$

if

$$|x"(n+1) - x"(n)| - |x"(n+1) + x"(n)| = 0 \quad (12)$$

Then $\omega - 5 = 0$, and $\omega = 5$.



(a) N=8 (b) N=4

Recovering a single frequency via a binary search

2.6. Aliased based search

Suppose the length of a signal is N which could be represented by the multiplication of coprime integers.For example:N=2x5x7. Let M be divisible by N.
(a) Randomly selected offset para $\tau \in [0, N/M - 1]$;
(b)let $z(n) = x(nM/M + \tau), n \in [0, M - 1]$;
(c)do DFT to z(n) of M points to get Z(k);
(d)The number of 2K largest entries Z(k) are grouped into set T.

$$z(k) = \Sigma_{n=0}^{N/M-1} x(k + nM) W_N^{-\tau(k+nM)} \qquad (13)$$

according to this formula we know that the non-zero value coordinate will appear in new set T in the form of k mod M.Set M into different value and we will get a set of formula and we can use Chinese remainder theorem to get the position of non-zero value.The negative part of this technique is that we can never avoid collided(e.g congruent to the same residue modulo)

3. Results

Experiment 1

In the first experiment we keep the sparsity of the input signal constant at $K = 50$ and change the N from the range $N = [8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576]$. We see from the error plot that as we increase the N to be large the error stays almost on the same level. This is because the accuracy of our algorithm relies on the sparsity of the signal K and the level of noise in the signal. The runtime of our algorithm for all values of N did not outperform the scipy fft algorithm, which is to be expected since the scipy fft was implemented in C. We give a comment about the differences in runtime and improving it in the conclusion. We give several examples of running the algorithm for some N.

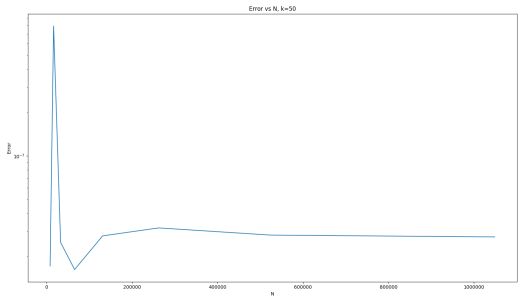


Figure 1. Experiment 1 error for various N.

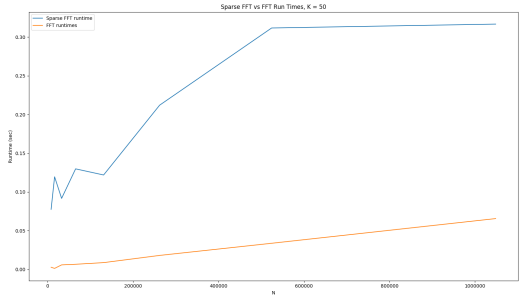


Figure 2. Experiment 1 runtime for various N.

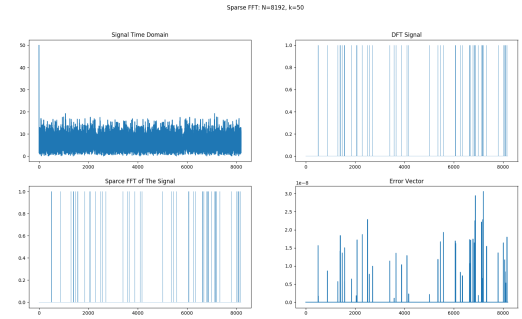


Figure 3. Experiment 1 for N=8192, K=50.

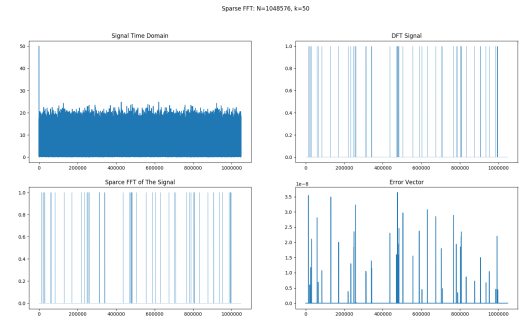


Figure 4. Experiment 1 for N=8192, K=50.

Experiment 2

In the second experiment we keep the length of the input signal constant at $N = 1048576$ and change the K from the range $K = [50, 100, 200, 500, 1000, 2000]$. Making the K large makes the error of the sparse Fourier transform larger. This is because to approximate the FFT with low probability of error we assume $K << N$. So as K gets large we get more error.

Experiment 3

In this experiment we keep the length of the signal at $N = 262144$ and $K = 50$ and add noise to the signal for different SNR radios.

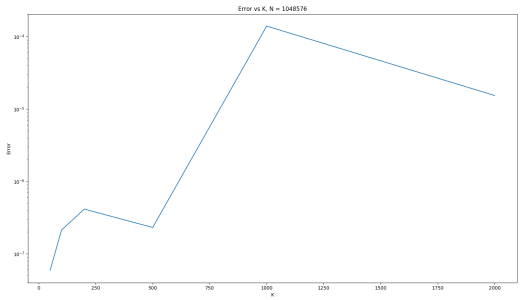


Figure 5. Experiment 2 error for various K.

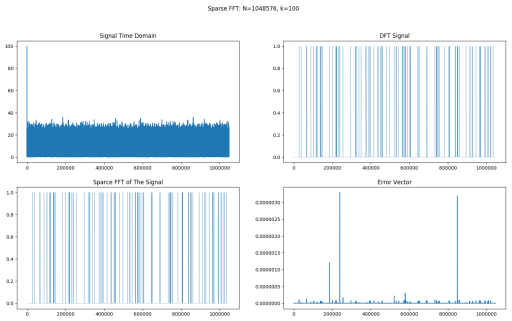


Figure 6. Experiment 2 for N=1048576, K=100.

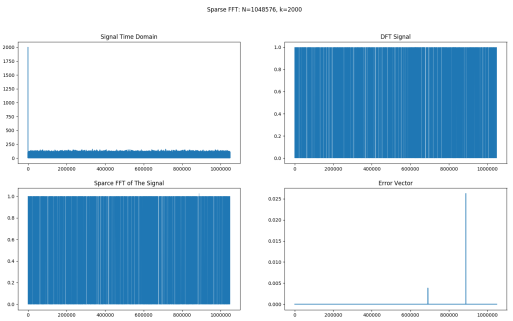


Figure 7. Experiment 2 for N=1048576, K=2000.

We add the SNR from the range of $SNR = [-20, -10, -7, -3, 0, 3, 7, 10, 20, 30, 40, 50, 60, 120]$ in dB. We observe that for adding noise -20 dB SNR generates a lot of error in the sparse Fourier transform. This is because the noise introduces frequencies that add to the large signal frequencies and can make some frequencies seem large in the noisy signal. As we reduce the noise level the error gets smaller.

4. Conclusion

The sparse fourier transform is a fast algorithm that can approximate the fft algorithm with high probability if the input signal has $K << N$ frequencies. Since the signal has

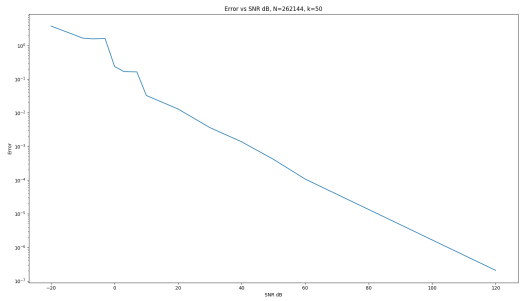


Figure 8. Experiment 3 error for various SNR.

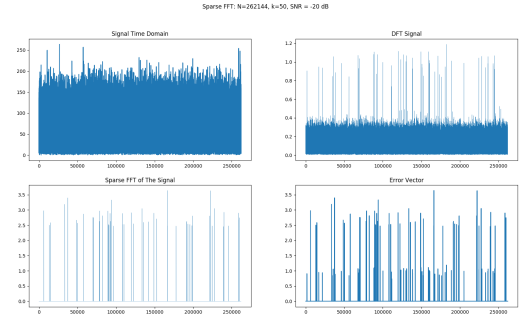


Figure 9. Experiment 3 for N=262144, K=50, SNR=-20dB.

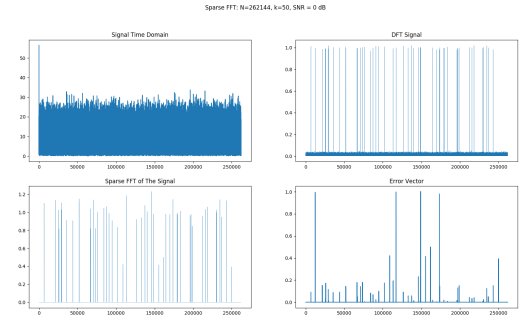


Figure 10. Experiment 3 for N=232144, K=50, SNR=0dB.

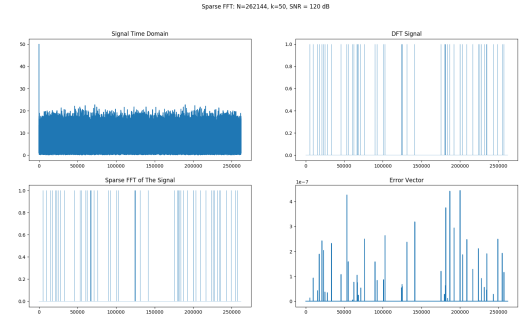


Figure 11. Experiment 3 for N=262144, K=50, SNR=120dB.

only a few frequencies we can approximate the fft from a down sampled signal. Using the flat window filter for binning the frequencies allows us to get the large frequencies without any iteration process. Our implementation was getting the same level of error for signals with different levels of noise as the implementation in the paper. Since Python is a high level interpreted language our implementation did not run faster than the scipy fft implementation. This is because scipy fft algorithm was actually implemented in C and heavily optimized for cache locality and relies on vectorized computation. Implementing an algorithm like the sparse fourier transform in Python allows rapid implementation but it sacrifices the efficiency that could be gained by implementing it in C. In conclusion, we recommend implementing the sparse fourier transform in Python first to rapidly develop a working implementation as well as gaining theoretical insights that would be otherwise hard to get in C. But, once a working algorithm is implemented in Python it needs to be ported into C to gain performance improvements.

5. Future Work

Our current Python implementation is slower than the scipy fft implementation. This is because the scity fft was actually implemented in C and heavily optimized for cache locality. To make our Python sparse fourier transform faster we could implement it in C and port it into Python using shared libraries.

6. Project Code GitHub Link

All code for this project can be found at GitHub https://github.com/AndriiDSD/SparseFFT.

References

[1] Anna C Gilbert, Piotr Indyk, Mark Iwen, and Ludwig Schmidt. Recent developments in the sparse fourier transform: a compressed fourier transform for big data. *IEEE Signal Processing Magazine*, 31(5):91–100, 2014.

[2] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse fourier transform. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1183–1194. Society for Industrial and Applied Mathematics, 2012.