Placeholder for joe's content

1 FFT with Application on Polynomial Multiplication

1.1 Introduction

The fast Fourier transform (FFT) is a name given to any algorithm that is able to compute the discrete Fourier transform (DFT) of a function over the N^{th} roots of unity in $O(N \log_2 N)$ time with regards to addition and multiplication of complex numbers. It is *fast* in the sense that under general settings to compute the DFT of any given function, one has to compute and multiply with their respective frequencies N Fourier coefficients. With each coefficient being a sum of N products of multiplications, this computation effectively takes $O(N^2)$ operations. This improvement in efficiency is achieved through realizing properties of the N^{th} roots of unity such that redundant calculations would be eliminated and thereby giving us the speed-up we desire. In this section we will first show the significance of a fast polynomial multiplication algorithm and then generalize it to the case of FFT.

1.2 Operations on Polynomials in Different Forms

The main purpose of this subsection is to provide an overview on the pros and cons of doing polynomial operations in different representations and see if we could find the most efficient method of doing these operations. In the following subsections we will devise an algorithm which will complete our search for efficiency. We first start by going over the operations we will cover on any degree $n-1$ polynomial of the general form $P(x) = \sum_{i=0}^{n-1} p_i x^i$ for scalars $(p_i)_{i=0}^{n-1}$.

- **Evaluation:** Given a polynomial $P(x)$ we can evaluate it at point x by feeding it x to receive an output y.
- **Addition:** Given two polynomials $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ we can add them together to obtain a third polynomial $C(x) = \sum_{i=0}^{n-1}(a_i + b_i)x^i = \sum_{i=0}^{n-1} c_i x^i$.
- **Multiplication:** Given two polynomials $A(x)$ and $B(x)$ as described above, we can most certainly multiply them for a third polynomial $C(x) = \sum_{i=0}^{2(n-1)} c_i x^i$ where $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. Note that this multiplication defines a vector $(c_0, c_1, ..., c_{2(n-1)})$ which describes exactly the convolution of $(a_0, ..., a_n)$ and $(b_0, ..., b_n)$ with respect to their inner products. What this implies is that if we can multiply polynomials efficiently, we can convolute vectors efficiently, and thereby process signals efficiently given their vector forms.

To optimize the computational time of these three operations it is helpful to see what different representations of polynomials could provide for our purpose. There are namely three ways of preserving the information contained in a polynomial which are listed below.

- **Coefficient vector:** For any degree $n-1$ polynomial, we can write it as a vector in a vector space with the basis $\{1, x, x^2, ..., x^{n-1}\}$. E.g. the polynomial $P(x) = 1 + 3x + 54x^2$ is equivalent to the vector $(1, 3, 54)$ in a three-dimentional vector space using the aforementioned basis.

- **Roots:** If we know all roots of a polynomial along with a scalar constant surely we can construct the polynomial as a product of $(x-r_i)$'s multiplied by the constant c with each r_i being a root of the polynomial, as guaranteed by the fundamental theorem of algebra. E.g. the polynomial $P(x) = 4x^3 - 24x^2 + 44x - 24$ could be written as $P(x) = 4(x-3)(x-2)(x-1)$.

- **Samples:** A sample of a polynomial $P(x)$ is an ordered pair (x_i, y_i) where $y_i = P(x_i)$. Say $P(x)$ has degree $n-1$, then we require exactly n distinct samples to be able to uniquely determine a $P(x)$ that is coherent with our n samples. This could be accomplished through Lagrange's polynomial interpolation or solving a simple system of equations. E.g. if given three distinct samples (x_i, y_i) we could uniquely determine a polynomial by solving the system

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

  to obtain $P(x) = c_1 + c_2x + c_3x^2$.

With these three different representations we shall see that they each have their own perks and defects as far as our operations are concerned, which will then prompt the question of "is there a middle ground covering the best cases in each form?" The answer is no, sadly. However, we may build a connection using FFT to gain access the best parts of these representations. This will allow for a close-to-ideal solution.

1.3 Runtime Analysis on Polynomial Operations

For the following estimates we will assume our polynomials are of degree $n-1$.

We start with polynomial evaluation. In coefficient form to evalute a polynomial at point x will take at most $O(n)$ calculations using Horner's rule. Simply put, calculate and record each power of x by multiplying x with itself $n-1$ times, then multiplying each power of x with the corresponding coefficient takes another n multiplications, finally adding them together takes at most $n-1$ additions, so we would end up doing $3n-2$ operations which is in the order of linear time $O(n)$. Evaluation with roots is also in linear time, as each $(x-r_i)$

can be calculated with a single addition and multiplying $n-1$ of them together with a constant will take us $n-1$ multiplications. However with samples, to evalute the polynomial we would first have to recover the polynomial through interpolation and then evaluate. The bottleneck here rests on the interpolation step which takes $O(n^2)$ calculations to compute, as can be visualized from the example of solving a system of linear equations given above.

Polynomial addition is straight forward in coefficient form, since it can be computed by simply adding the two coefficient vectors together which would take $O(n)$ computations. Addition with only the roots of two polynomials is extremely difficult, and conversion to other forms would be impossible if we want to end up with the roots of the sum, so we do not consider it as a feasible approach in this paper. Addition with samples, however, takes only linear time. The sum of two polynomials evaluated at one point is simply the sum of the values of both polynomials evaluated at that point. Hence for n samples, performing addition takes $O(n)$ calculations.

Multiplication with coefficients would take $O(n^2)$ computations as it can be written as a double summation given earlier. With roots, one simply concatenate the $(x - r_i)$'s together, which takes $O(n)$ concatenations. With samples it takes only $O(n)$ operations as well, since multiplying the second entry of both samples at the same point accomplishes polynomial multiplication.

To summarize, we have the following chart that lists the computational complexities we have considered thus far.

	Coefficients	Roots	Samples
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	N/A	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

As the chart shows, no representation is perfect and there is a price to be paid no matter which representation one adopts. However, what if there is a method of efficiently converting between representations such that it would make sense to convert and perform the opertaion rather than accepting the fate of an undesirable $O(n^2)$ complexity? Indeed, if conversion could cost significantly less, we would be able to save quite some time. Luckily, it is possible to convert between coefficient form and sample form efficiently in $O(n \log_2 n)$ time, as we will show in the following subsections.

1.4 From Coefficients to Samples

Our objective in this section is to find an algorithm which can convert coefficient vectors into a set of samples efficiently so that we may employ this conversion as an escape route from the horrid $O(n^2)$ runtime it requries to perform polynomial multiplication.

To obtain a sufficient amount of samples from a polynomial, we evaluate our polynomial of interest $P(x) = \sum_{i=0}^{n-1} a_i x^i$ at some x_k in our domain for n such distinct x_k's. This ensures that the least degree polynomial we construct from the samples obtained matches $P(x)$. The following matrix multiplication accurately describes our sampling of polynomial at n distinct points:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \tag{1}$$

If we could perform (1) efficiently and invert it efficiently as well, we could gain access to the ideal $O(n)$ running time it takes to perform polynomial multiplication with samples. Hence our goal is to figure out a way to compute (1) as efficiently as possible.

1.5 A Divide-and-Conquer Approach

The usual methods of performing matrix multiplication is to find the dot product of $\vec{a}$ with every row of the matrix it is being multiplied by. However, as each dot product takes $O(n)$ time to compute and we need to compute n of such dot products, the overall running time would be $O(n^2)$ which renders the conversion meaningless for an efficient polynomial multiplication. So at this point we will introduce a divide-and-conquer approach of computing (1) in hopes of a better running time.

The philosophy of divide-and-conquer algorithms is to break a problem down into smaller subproblems, and then break those subproblems down recursively in the same fashion until it is feasible to compute the subproblems without much effort. This approach aims to avoid direct computation of a large problem and instead shift the amount work that has to be done onto the division and combining of subproblems.

We will achieve the ability to recursively break the computation of the dot product between $\vec{a}$ and a row $\vec{x}$ in the matrix by defining the following:

$$\begin{aligned} P_{\text{even}}(x) &= \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} a_{2k} x^k \\ P_{\text{odd}}(x) &= \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} a_{2k+1} x^k \end{aligned}$$

which effectively splits $P(x)$ into two polynomials each half the size of $P(x)$ with only odd or even coefficients. However the powers of x in each sum does not match the coefficients they are multiplied with, so in order to write $P(x)$ as a combination of these two subproblems, we would have to write $P(x)$ as

$$P(x) = P_{\text{even}}(x^2) + x \cdot P_{\text{odd}}(x^2). \tag{2}$$

Recursively apply this method to P_even and P_odd and so on will allow us to compute only bite-size problems which are easily solvable.

Now it is important to note that we want to assume n is a power of two, as the smallest subproblem we want to handle is evaluation of monomials. The assumption does very little to hurt the generality of this algorithm as one can always fill a coefficient vector with trailing zeroes up to the closest power of two and be happy that the algorithm applies well to it.

With this recursion in hand, we want to analyze the resulting runtime. Let T measure the runtime of computing $P(x)$ for every x in the set X of points we want to sample our polynomial at, then we have the following recurrence equation relating the runtime of a problem to the runtime of its subproblems:

$$\begin{aligned}
T(n, |X|) &= 2T(\frac{n}{2}, |X|) + O(|X|) \\
&= 2\left[2T(\frac{n}{4}, |X|) + O(|X|)\right] + O(|X|) \\
&\vdots \\
&= 2^{\log_2 n} T(1, |X|) + \sum_{l=0}^{(\log_2 n)-1} 2^l O(|X|) \qquad (3) \\
&= n^2 + O(|X| \log_2 n) \\
&= O(n^2)
\end{aligned}$$

where n is the degree of the polynomial concerned in each stage. In the very first equality our problem on $P(x)$ gets divided into problems on $P_{even}(x)$ and $P_{odd}(x)$ at the cost of $O(|X|)$ time used to split and combine the subproblems back into the original one on $P(x)$ for every x in X. By unrolling this recurrence equation we obtain the last equality.

Although we ended with yet another $O(n^2)$ algorithm, an important insight we gained here is that we only have $O(n^2)$ runtime as a result of needing to evaluate monomials for every x in X. This is the case as in general the size of X does not reduce when one square each element in X; we only need to evaluate subproblems at x^2 for each layer of recursion we add as a result of how (2) was constructed.

Now that we have identified the root cause of the congestion in this algorithm, we look for a set which reduces in size as we square each element. Since we only need n distinct samples of a polynomial to preserve its data with no other requirements, we are justified to evaluate $P(x)$ over any set X given the size of it is n.

1.6 The Nth Roots of Unity

To find a set which reduces in size as each element is squared, we start by taking the square-root of numbers as the square-root fucntion returns two outputs which fit our purpose for every input we provide. Working reversely in this fashion, we acquire the progression below:

$$\{1\}$$
$$\{1, -1\}$$
$$\{1, -1, i, -i\}$$
$$\{1, -1, i, -i, \tfrac{\sqrt{2}}{2}(i+1), \tfrac{-\sqrt{2}}{2}(i+1), \tfrac{\sqrt{2}}{2}(i-1), \tfrac{-\sqrt{2}}{2}(i-1)\}.$$

This could be continued for as long as we would like, however at this point it is enough to highlight the important feature of these sets that is *every number listed above lies on the unit circle* on the complex plane. This property holds for as many square-roots as we would like to take on any number above.

To provide an explaination, we note that the last set above is simply the set

$$\{e^{2\pi i \frac{k}{8}}, k = 0, 1, ..., 7\} \tag{4}$$

of complex exponentials. From here we see that squaring any element in (4) simply doubles the value of k. Squaring the element four times will eventually yield

$$e^{2\pi i \frac{8k}{8}} = (e^{2\pi i \frac{8}{8}})^k = (1)^k$$

which is a result of Euler's formula. Geometrically we are spinning the point on the complex plane by its angle from $(1, 0)$ and doubling the magnitude we spin it by in every subsequent spin until it eventually rests on $(1, 0)$.

In general, the set

$$\{e^{2\pi i \frac{k}{n}}, k = 0, 1, ..., n-1\}$$

where n is a power of two reduces to $\{1\}$ when we take the n^{th} power of each element, or equivalently, square each element $\log_2 n$ times. Such set is what is known as the n^{th} roots of unity as the n^{th} power of any element evaluates to unity.

By sampling our polynomial over this specific set, we immediately see that every resulting subproblem, i.e. monomial, only has to be evaluated at $x = 1$ instead of n distinct values.

1.7 Completing the Algorithm

Revising (3) to take into account our choice of X, we have that

$$\begin{aligned}
T(n, |X|) &= 2^{\log_2 n} T(1, |X^n|) + \sum_{l=0}^{(\log_2 n)-1} 2^l O(|X|) \\
&= n \cdot T(1, 1) + \log_2 n O(|X|) \\
&= n + O(n \log_2 n) \\
&= O(n \log_2 n)
\end{aligned}$$

which is exactly the efficient conversion from coefficient vectors into samples we were looking for.

This algorithm will enable us to convert two polynomials $A(x)$ and $B(x)$ from coefficient vectors into samples in $2O(n\log_2 n)$ time and perform multiplication in $O(n)$ time.

To convert the product polynomial $C(x)$ back into coefficient form, we shall prove that the inverse operation of

$$\begin{pmatrix} 1 & (e^{2\pi i\frac{0}{N}})^1 & (e^{2\pi i\frac{0}{N}})^2 & \cdots & (e^{2\pi i\frac{0}{N}})^{N-1} \\ 1 & (e^{2\pi i\frac{1}{N}})^1 & (e^{2\pi i\frac{1}{N}})^1 & \cdots & (e^{2\pi i\frac{1}{N}})^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (e^{2\pi i\frac{N-1}{N}})^1 & (e^{2\pi i\frac{N-1}{N}})^2 & \cdots & (e^{2\pi i\frac{N-1}{N}})^{N-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix},$$

which we denote as $V\vec{a} = \vec{y}$ (V for Vandermonde), is simply

$$\frac{1}{n}\bar{V}\vec{y} = \vec{a}.$$

Or equivalently, we show that $V^{-1} = \frac{1}{n}\bar{V}$.

Proof. To show that $V^{-1} = \frac{1}{n}\bar{V}$, it is the same as showing that $V\bar{V} = nI$ where I is the identity matrix. We proceed simply by working out the product of matrices.

Let $M = V\bar{V}$, then

$$(m)_{a,b} = \sum_{k=0}^{n-1} v_{a,k}\bar{v}_{k,b} = \sum_{k=0}^{n-1} e^{2\pi i\frac{ak}{n}} e^{-2\pi i\frac{kb}{n}} = \sum_{k=0}^{n-1} e^{2\pi i\frac{ak-kb}{n}}.$$

If $a = b$, we have that $(m)_{a,b}$ is simply a sum of n e^0's, and this gives us that the diagonal entries of M must be n.

If $a \neq b$, we would have a power series to evaluate:

$$\begin{aligned} (m)_{a,b} &= \sum_{k=0}^{n-1}\left(e^{2\pi i\frac{a-b}{n}}\right)^k \\ &= \frac{\left(e^{2\pi i\frac{a-b}{n}}\right)^n - 1}{e^{2\pi i\frac{a-b}{n}} - 1} \\ &= 0. \end{aligned}$$

Now we know that any non-diagonal entries of M is 0, hence $M = nI$ as desired. □

We could apply the same divide-and-conquer technique to compute $\frac{1}{n}\bar{V}\vec{y} = \vec{a}$ and obtain our polynomial $C(x)$ as a vector of coefficients.

In summary, we pay $2O(n\log_2 n)$ to convert $A(x)$ and $B(x)$ into samples and perform multiplication in $O(n)$, then pay $O(n\log_2 n)$ time again to have our polynomial $C(x)$ in coefficient form. Here we assume $n-1$ is the degree of $C(x)$ as it takes n samples of $C(x)$ to accurately reconstruct a degree $\deg(A)+\deg(B)$ polynomial from them.

What this gives us is a way of convoluting two signals or waves represented by polynomial curves in a *fast* manner. Coupled with the Stone-Weierstrass Theorem which shows any continuous function can be approximated by polynomials, we now have a powerful tool in our hands to do greater things.

1.8 The Fast Fourier Transform

The discrete Fourier transform as described in [2] is given by

$$F(k) = \sum_{r=0}^{n-1} A_r e^{2\pi i \frac{k}{n} r}$$
$$\text{where } A_r = \frac{1}{N}\sum_{k=0}^{n-1} F(k) e^{-2\pi i \frac{k}{n} r}.$$

This correspond exactly to the equation $\frac{1}{n}\bar{V}\vec{y} = \vec{a}$ in the previous subsection if we let $\vec{a} = (F(0), F(1), \cdots, F(k)) = \vec{F}$. Following this trend, the inverse opertaion would simply be $V\vec{F} = \vec{y}$. We have already shown that this transformation could be done in $O(n \log_2 n)$ time, hence the algorithm we described above is the FFT itself.

In the case of converting representations of polynomials, the coefficient form corresponds exactly to the frequency space one would conventionally denote in Fourier analysis, as a coefficient vector tells us exactly how much of each power of x a given polynomial has. Analogously with samples, they are in the time space. Once this realization has been made, it is immediately clear that we have used exactly the convolution theorem for inverse Fourier transform:

$$\begin{aligned} &\mathscr{F}^{-1}\{A(x) * B(x)\} = \mathscr{F}^{-1}\{A(x)\} \cdot \mathscr{F}^{-1}\{B(x)\} \\ \text{or} \quad & A(x) * B(x) = \mathscr{F}\{\mathscr{F}^{-1}\{A(x)\} \cdot \mathscr{F}^{-1}\{B(x)\}\}. \end{aligned}$$

This concludes the section which has set out to demonstrate the Radix-2 form of FFT, an algorithm first used by C.F. Gauss without recognition and then discovered again by James Cooley and John Tukey in 1965.

References

[1] Demaine, Erik D. “3. Divide & Conquer: FFT.” YouTube. MIT OpenCourseWare, 04 Mar. 2016. Web. 07 Mar. 2017.

[2] Stein, Elias M., and Rami Shakarchi. “Chapter 7 Finite Fourier Analysis.” Fourier Analysis: An Introduction. Princeton: Princeton UP, 2003. N. pag. Print.

Place holder for ethan’s content

The Fourier Transform converts a domain into a frequency domain. As we saw earlier, the Fourier Transform in audio signals takes a continuous time domain and converts it into a frequency domain. In image processing, the Fourier Transform converts a spatial domain into a frequency domain (Smith 24). The Fourier Transform in image processing allows us to access the geometric characteristics of an image (Fisher 3). By having access to the geometric characteristic, we are able to modify the image by changing some of those characteristics.

In order to modify the characteristics of an image, we apply filters. Most of the time we apply filters to the original image. However, this is not always the most efficient way to manipulate an image. When we want to smooth, sharpen, and enhance an image, it is easier to take the Discrete Fourier Transform, apply the filters, and then take the inverse Discrete Fourier Transform to obtain the image with the changes (Ludwig 3). It is easier to apply the filter in the Discrete Fourier Transform because sharpening, enhancing, and smoothing an image are all consequences of convolution. The Discrete Fourier Transform allows convolution to occur easier mathematically since convolution in the spatial domain becomes simply multiplication in the frequency domain (Smith 24). Therefore, applying filters in the frequency domain is much easier than applying filters in the spatial domain.

Because images are in two dimension, we cannot calculate the Fourier Transform using a one dimension formula, like we did in audio signals. Therefore, we need to manipulate the Fourier Transform formula to take care of this extra dimension.

The one dimensional (1D) Discrete Fourier Transform formula is:

$$F(k) = \sum_{x=0}^{N-1} f(x) e^{-i2\pi kx/N}$$

We take the DFT of the rows, $F(k)$, and then the DFT of the columns, $F(l)$:

$$F(k) = \sum_{x=0}^{N-1} f(x) e^{-i2\pi kx/N} \qquad F(l) = \sum_{y=0}^{N-1} f(y) e^{-i2\pi ly/N}$$

Then we multiply them together:

$$\begin{aligned}
F(k)F(l) = F(k,l) &= \sum_{x=0}^{N-1} f(x)e^{-i2\pi kx/N} \sum_{y=0}^{N-1} f(y)e^{-i2\pi ly/N} \\
&= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi kx/N}e^{-i2\pi ly/N} \\
&= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi kx/N-i2\pi ly/N} \\
&= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-i(kx+ly)2\pi/N}
\end{aligned}$$

Therefore, we get the 2 dimensional (2D) Discrete Fourier Transform formula:

$$\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-i(kx+ly)2\pi/N}$$

where $f(x,y)$ is the image of the spatial domain.

Note that this formula only works for an $N \times N$ image, where N is a power of 2. If the image is not $N \times N$, add the necessary 0 vectors in either the rows or columns until it is.

Similarly, we can derive the 2D inverse DFT formula from the 1D inverse DFT formula:

$$f(x,y) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} F(k,l)e^{i2\pi(kx/N+ly/N)}$$

Note the presence of $1/N^2$. This is sometimes present in the Discrete Fourier Transform formula. However, $1/N^2$ should not be present in both the DFT and the inverse DFT.

The Discrete Fourier Transform is complex, therefore, having real and imaginary parts. In image processing, instead of displaying the DFT as real and imaginary images, we display the DFT using magnitude and phase.

The magnitude can be calculated by:

$$\mathrm{Mag}(F) = \sqrt{\mathrm{Real}(F)^2 + \mathrm{Imaginary}(F)^2}$$

The phase can be calculated by:

$$\mathrm{Phase}(F) = \arctan\left(\frac{\mathrm{Imaginary(F)}}{\mathrm{Real(F)}}\right)$$

The magnitude tells us how much a certain frequency is present, while the phase tells us where the frequency is in the image (Brayer 2). Even though the phase typically holds more information, we only display the magnitude image of the DFT. When applying filters, we do not want to change where the frequency is located, but manipulate the presence of the frequency themselves. Therefore, the information pertinent to us is the magnitude.

Let's look at an example. Figure 1 is our original image. While figures 2

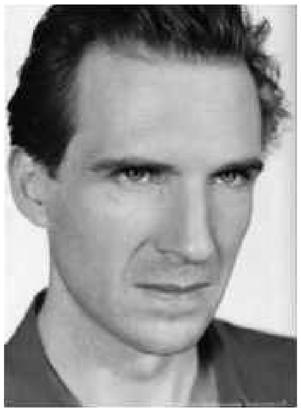
Figure 1: The Original Image

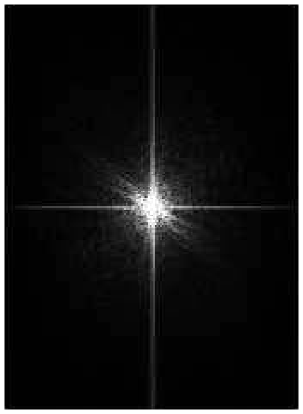
Figure 2: Magnitude of our Transform

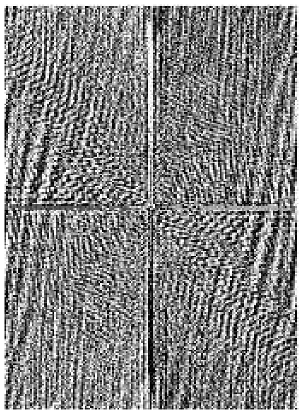
Figure 3: Phase of our Transform

and 3 show the magnitude and phase of the Fourier Transform respectively.

Even though the information pertinent to us is the magnitude, we do not want to just ignore the phase. We want to preserve the phase because that contains important information of where each frequency is located. Without the phase, our image will be corrupted. In our example, if we only reconstructed the image using the magnitude, we would end up with figure 5. Therefore, even though the magnitude contains the most information, we still must protect and preserve the phase in order to obtain our image. Taking the inverse Discrete Fourier Transform of the phase without the magnitude, we will get an outline of what the image is without any other color detail (Kundur 17-19). In our example, we would get the image in figure 6 Note that because we preserved the phase, we know where each frequency should be located, therefore we will get an outline of the image.

The results of the Discrete Fourier Transform show an image that contains components of all frequencies. The higher the frequency is, the lower the frequency magnitude is. Therefore, low frequencies contain more information about the image than the higher frequencies. In order to obtain and keep the most information that we can, we take the logarithmic of the magnitude.that we can, we take the logarithmic of the magnitude. es while compressing high frequencies. However, note that if an image contains important information in the higher frequencies, applying the logarithmic operator may lead to information loss (Fisher 3).

In most Discrete Fourier Transform images, the Fourier image is shifted such

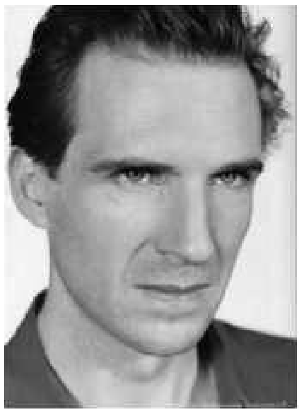
Figure 4: The Original Image

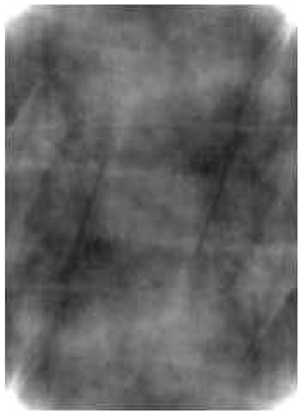
Figure 5: Inverse of Magnitude Only

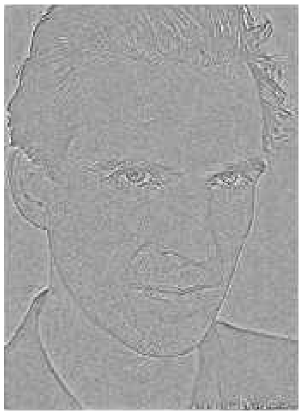
Figure 6: Inverse of Phase Only

that $F(0,0)$, the image mean, is centered in the middle (Fisher 3). Hence, the further from the origin, the higher the frequency, the lower the magnitude. In all of the pictures of the Discrete Fourier Transform in image processing in this paper, we will shift the Fourier image such that the image mean is centered in the middle. All Discrete Fourier Transform images will also represent the logarithmic of the magnitude of the Fourier Transform in 2D as well.

The purpose of using Fourier Transform in image processing is to apply filters in order to change the original image. I will give a few examples of manipulating the Fourier Transform and showing what the results would be like if we were to take the inverse Fourier Transform back to the original image.

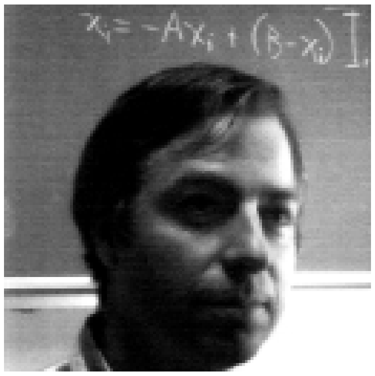

Figure 7: Original

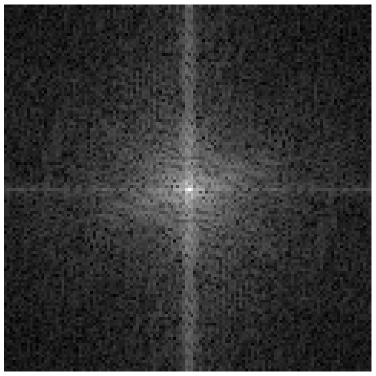
Figure 8: Spectrum

When we only take the lower frequencies and zero out the higher frequencies in the Fourier Transform image, we obtain the spectrum in figure 9 who's inverse fourier transform is shown in figure 10. Notice here in the Fourier Transform

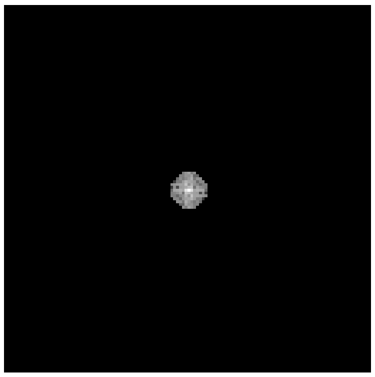
Figure 9: Low Pass Spectrum

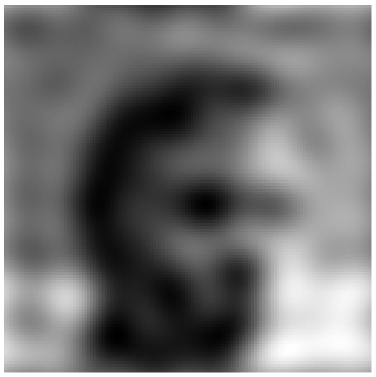
Figure 10: Low Pass Inverse

image we have the brightest spot preserved. However, we did lose quite a bit of high frequency and even some of the lower frequencies. Many of the bright spots in the original Fourier Transform image is lost. This is why the image after taking the inverse Fourier Transform of the new Fourier Transform image is so blurry. We lost too much information in the Fourier Transform.

Earlier in this section we said that the magnitude of the Fourier Transform told us how much of a frequency component was present in the image. This becomes apparent when we keep the high frequencies but zero out the lower frequencies in the Fourier Transform. This effect is highlighted in figure 12 which was obtained via the inverse of the spectrum shown in figure 11.

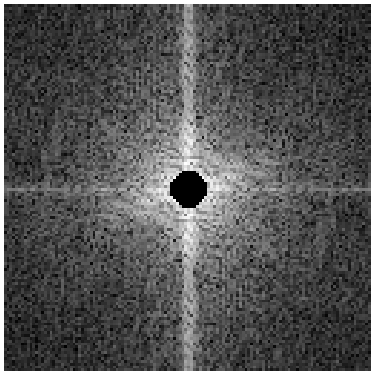
Figure 11: High Pass Spectrum

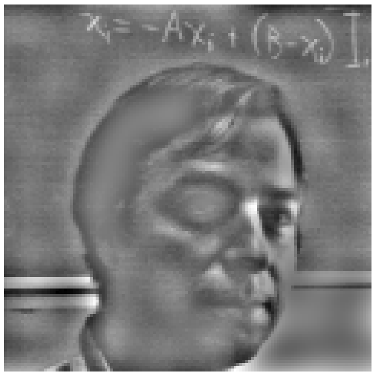

Figure 12: High Pass Inverse

Since the lower frequencies contain more information, the color of the image is mostly lost. However, because we preserved the phase and still have the higher frequencies, we are able to obtain an outline of an image. This is known as edge detection as the Fourier Transform is able to show the highly contrasted color changes of the original image.

The following images take out frequencies above and below a certain band limit. So within this band in the Fourier Transform shown in figure 13, it becomes figure 14 after an inverse fourier transform.

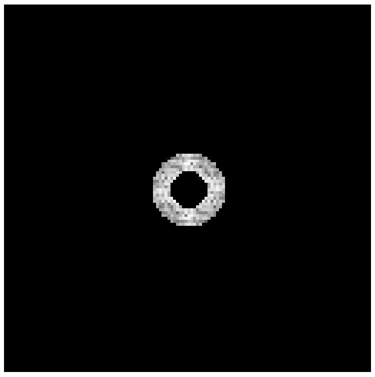
Figure 13: Band Pass Spectrum

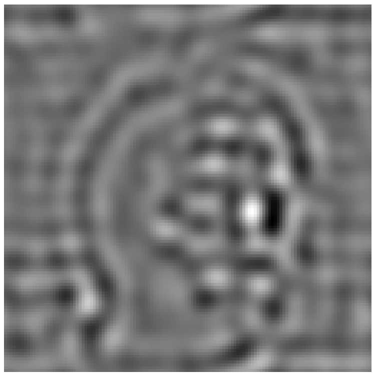
Figure 14: Band Pass Inverse

In the following image, we did the same thing. However, this time we decreased the size of the band in the Fourier Transform. Note that because the band is smaller, we have less information about the image. Therefore, the inverse Fourier Transform image in figure 16 is much blurrier than the inverse Fourier Transform of the image in figure 14

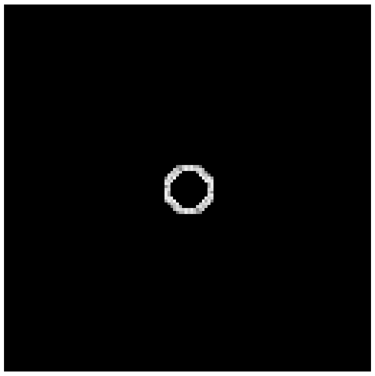
Figure 15: Band Pass Spectrum

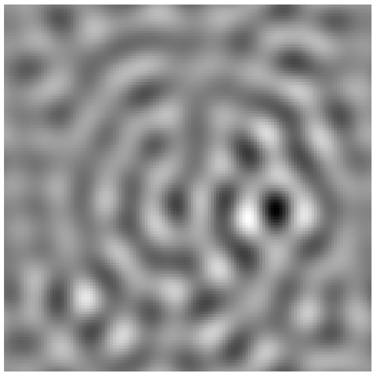
Figure 16: Band Pass Inverse

Finally we take a wide high frequency band limit in figure 17 and then reduce it's width in figure 19.

Note that the smaller band of higher frequency looks more similar to the larger band of higher frequency than the smaller band of lower frequency and the larger band of lower frequencies. This is again because lower frequencies contain more information. Therefore, the change of the band in the lower frequencies is more significant than changing the band in the higher frequencies.

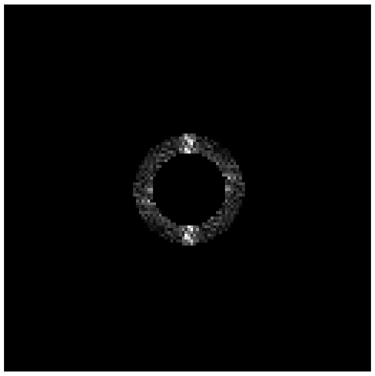
Figure 17: Band Pass Spectrum

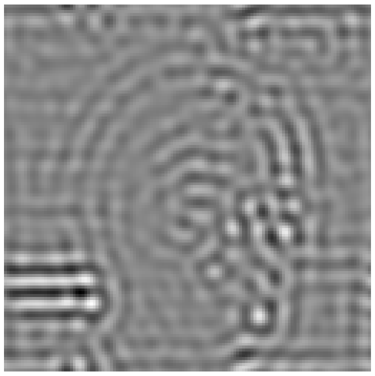
Figure 18: Band Pass Inverse

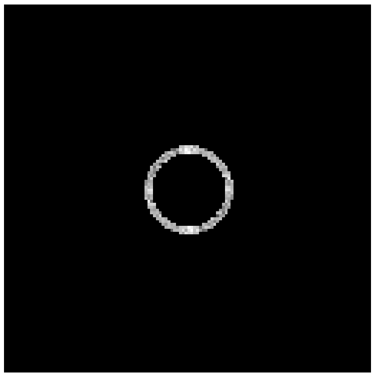
Figure 19: Band Pass Spectrum

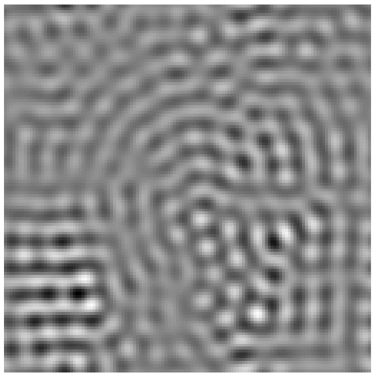
Figure 20: Band Pass Inverse

Also note that even though we lost a lot of information in the images after manipulating the original pictures Fourier Transform, we can still see a rough outline of what the image looks like. This is a result of preserving the phase.

Although it is nice to know how the changes in the Fourier Transform affects the images, typically in image processing, we wont be omitting many important frequencies, but enhancing and sharpening them.

In the next example, I will be using figure 21 as the original photo while figure 22 is it's spectrum. Then we apply our filters to these images to obtain the sharpened image in figure 23 with it's corresponding spectrum in figure 24 Notice that the frequencies of the higher magnitude frequencies are lighter than the original high frequencies in the original Fourier Transform image. In order to obtain this, we kept the lower frequencies the same, but every frequency above 96, we multiplied by the Fourier Transform coefficients by 4.

The Fourier Transform is also used to reduce noise. In figure 25, we have a picture of goofy. He has a cosine function embedded in him, which is why we have the "noise" or the rolls in the image. The cosine function can be seen in

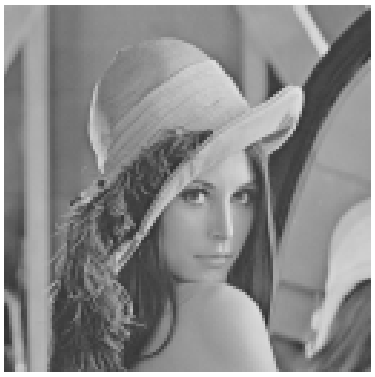
Figure 21: Original Image

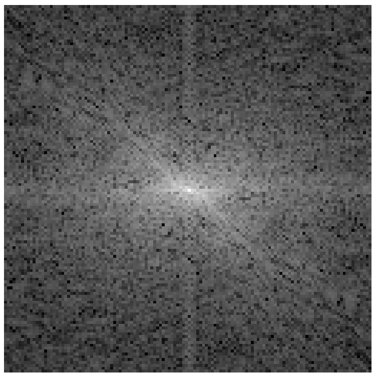
Figure 22: Original Spectrum

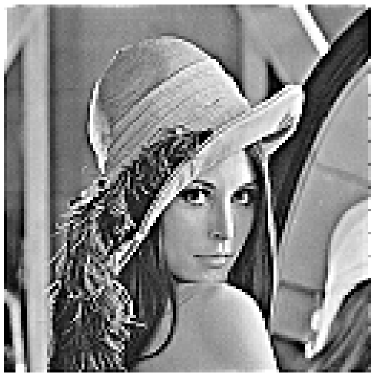
Figure 23: Filtered Image

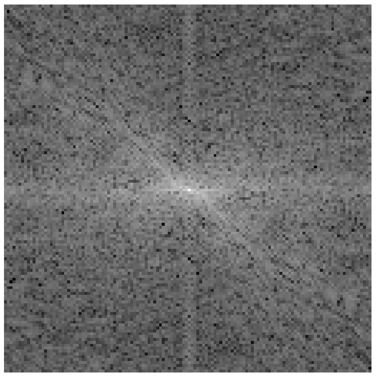
Figure 24: Filtered Spectrum

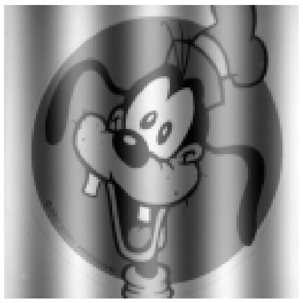
Figure 25: Original Image

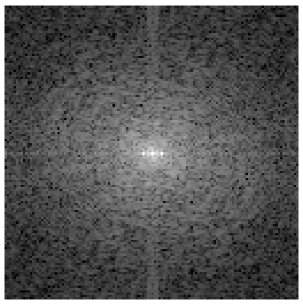
Figure 26: Original Spectrum

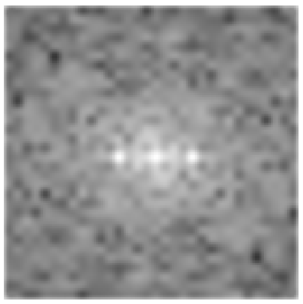
Figure 27: Zoomed Spectrum

the Fourier Transform image (figure 26). The center of the Fourier Transform image has three bright dots. In order to see the cosine function more clearly in the Fourier Transform, we zoomed in on the center of the Fourier Transform (figure 27). The outer dots represent the cosine function. Therefore, in order to remove the noise, we must take out the cosine function. So, we zero out the dots in the Fourier Transform. When we do that, we get figures 28-30 Notice

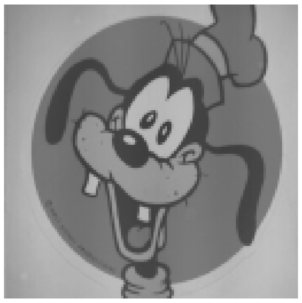
Figure 28: Filtered Image

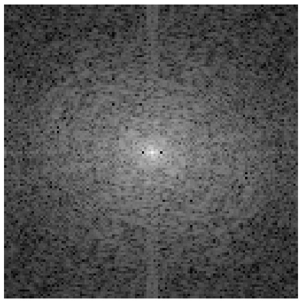
Figure 29: Filtered Spectrum

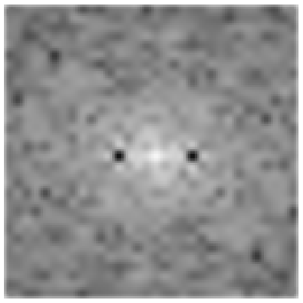
Figure 30: Zoomed Spectrum

that the outer bright dots are removed from the Fourier Transform. Therefore, the original image is smoother than before.

The Fourier Transform in image processing allows us to change some characteristics of an image in the spatial domain. By taking the Fourier Transform of every row of an N x N image and taking the Fourier Transform of every column, we are able to obtain a formula to calculate the Fourier Transform of a 2D image. By taking the magnitude, we can display the Fourier Transform image. In order to change the characteristics of an image in the spatial domain, we use the multiplication operator in the frequency domain. Sometimes we multiply the Fourier coefficients by zero in certain frequencies in order to remove those frequencies. By doing so, we can smooth out an image. By strategically multiplying certain Fourier coefficients in the frequency domain, we are able to enhance images. However, we cannot just focus on the magnitude of the Fourier Transform. We must preserve the phase as well. Preserving the phase of every image is important as the phase tells us where each frequency is located. A real world application of Fourier Transform is when looking for patterns within what looks like useless or confusing data. In a blog, a person by the name Chad Orzel, decided to explain this exact use to those reading the blog. He gathered up data on the traffic the blog he writes in receives daily in the past one thousand and twenty-four days or about three years (to the power of two so it would work nicely with the transform). From the picture they show

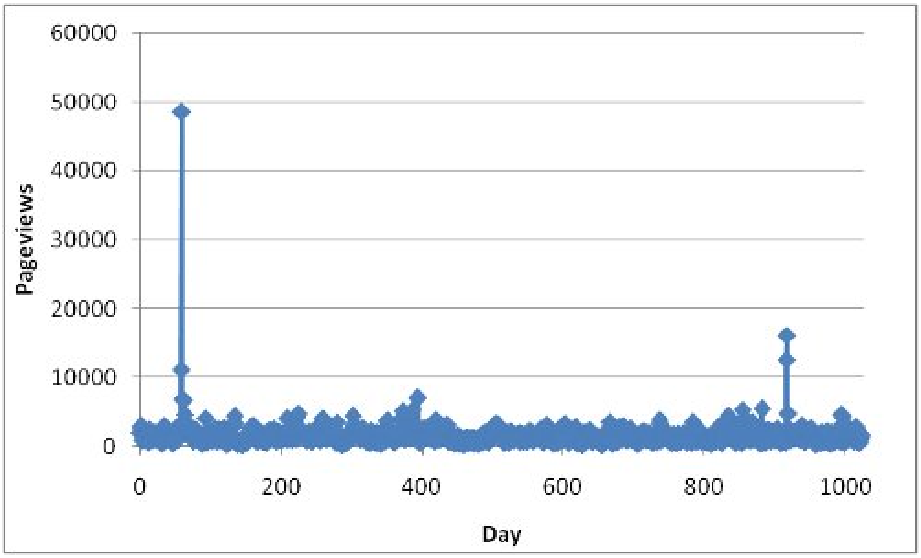

Figure 31: Views Per Day.

you can not really discern anything special about this graph at all except for those two peaks where they had very popular blog posts at those times. With just looking at this graph alone and doing nothing else with it we do not learn much about this traffic. Now we apply fourier transform to this data set we get

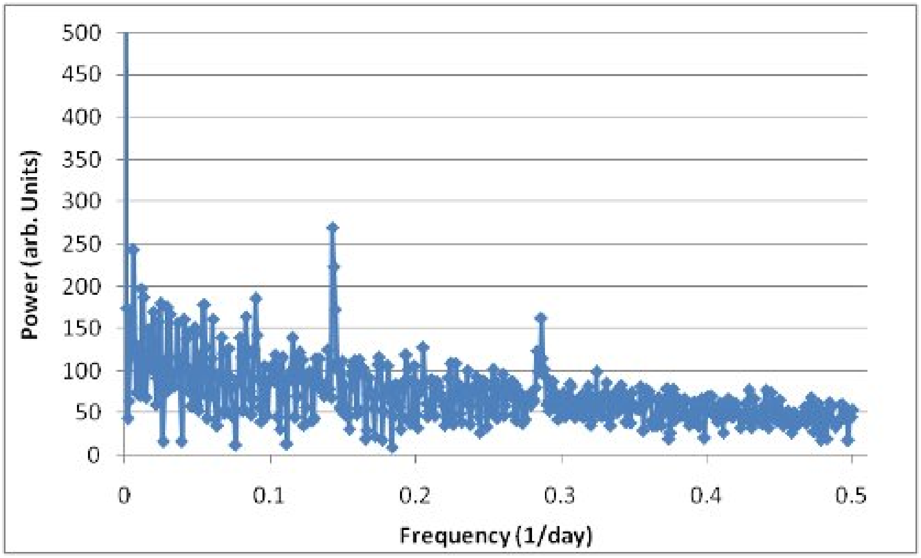

Figure 32: View Frequency Spectrum.

This graph tells us how much on a certain sin wave they are using in order to re create the first graph shown. As you can see there are two relatively larger spikes in the graph excluding the one at 0 as that is another aspect of fourier transformation. These two spikes to tell us however that these two certain sin waves are occurring more often and such a possible pattern. In

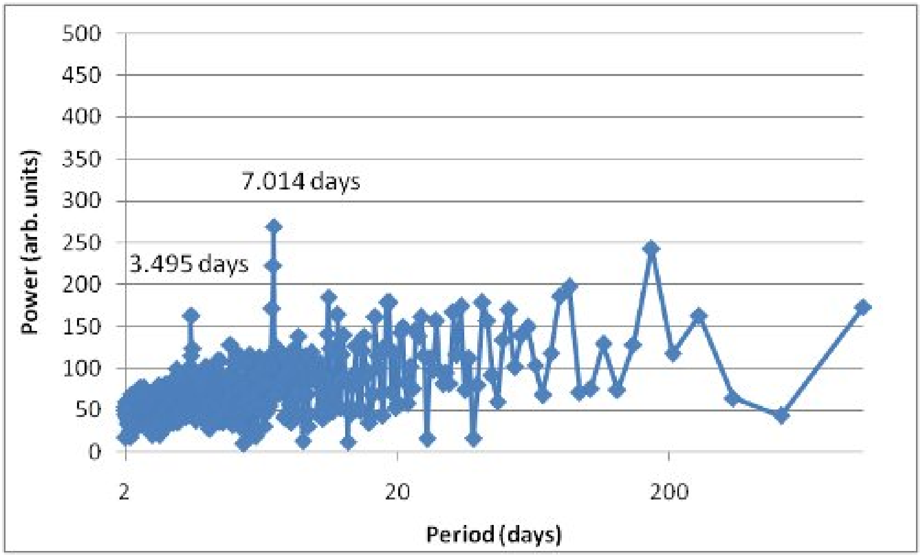

Figure 33: View Period Strength.

the person plotted the function as one over the frequency as well as setting it up with a log scale. From this graph we see that through fourier transformation we can notice that a very strong recurring pattern is occurring every week as well as a smaller one every half week.

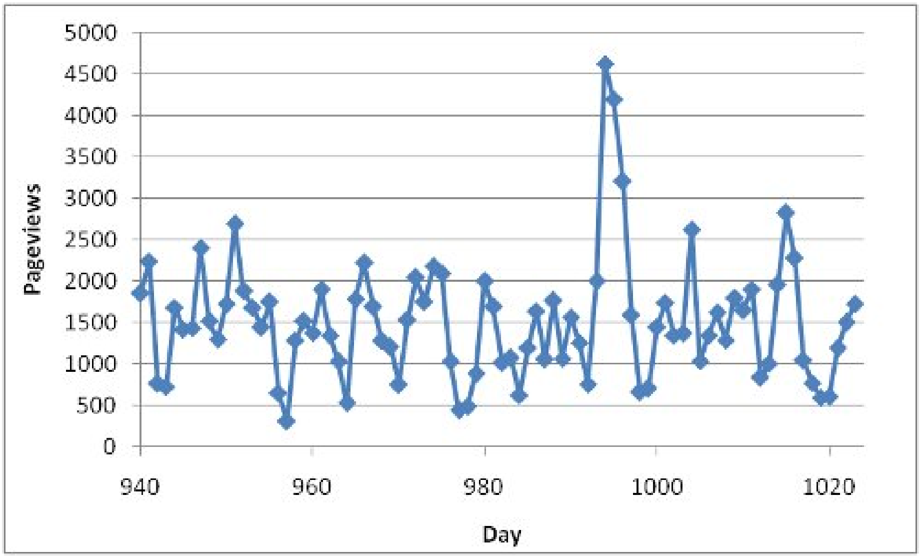

Figure 34: Views Per Day Closeup

Is a zoom in on our first graph showing this representation very well. Every seven days there is a strong dip downwards which is why in the fourier transform graph there is a large appearance of a certain sin wave as well as the impact of the smaller wave every half week.

In another example we are going to look at data over a certain amount of time, much like we did in the previous example. A person going by the name of Greg recorded the total number of user actions on their website. By looking at

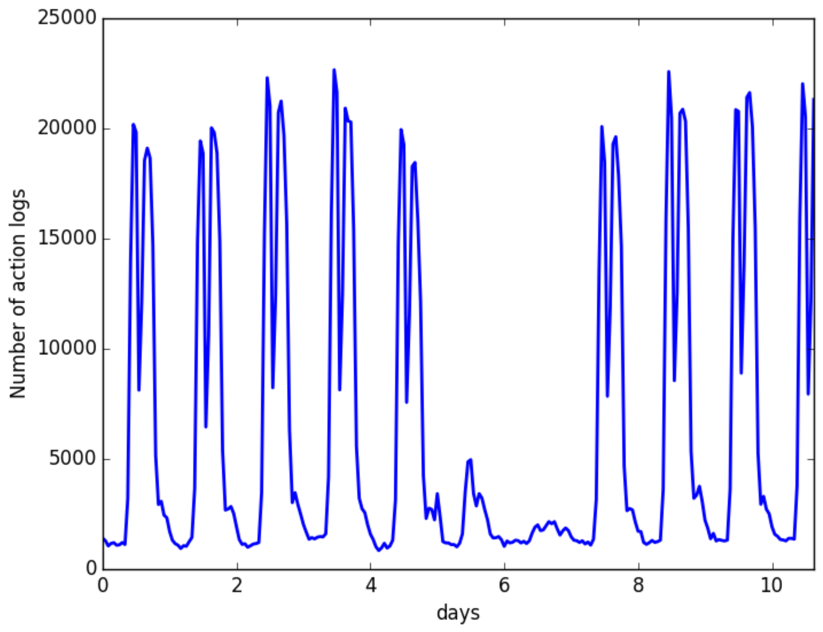

Figure 35: User Actions.

you are easily able to identify a somewhat periodic pattern in the graph, but is there anything else that may be missing? Perhaps there is something we can not notice, so we use the fourier transform on the set. In

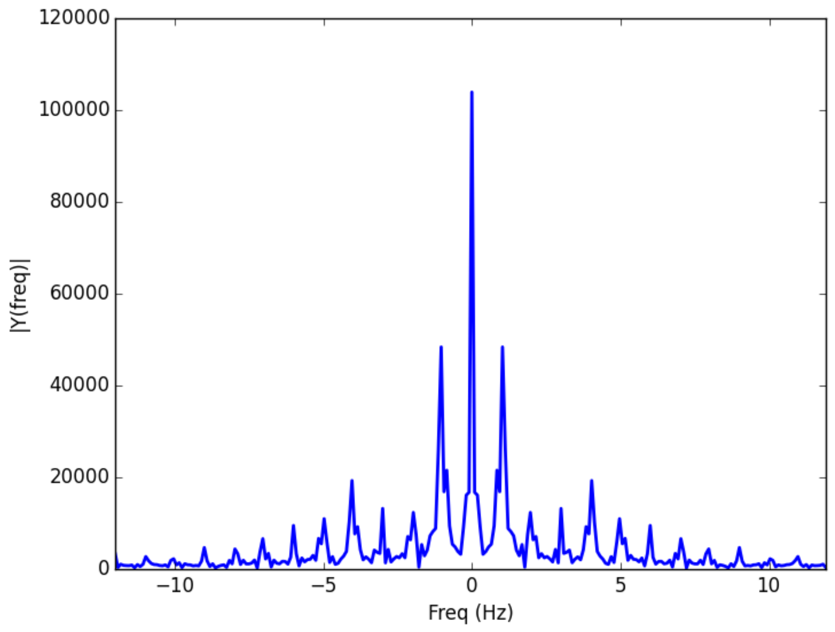

Figure 36: User Action Spectrum.

we notice that most of the actions on the site are using a low frequency sin wave. Now that the fourier transform in available we can directly compare the frequencies of the components to the original graph.

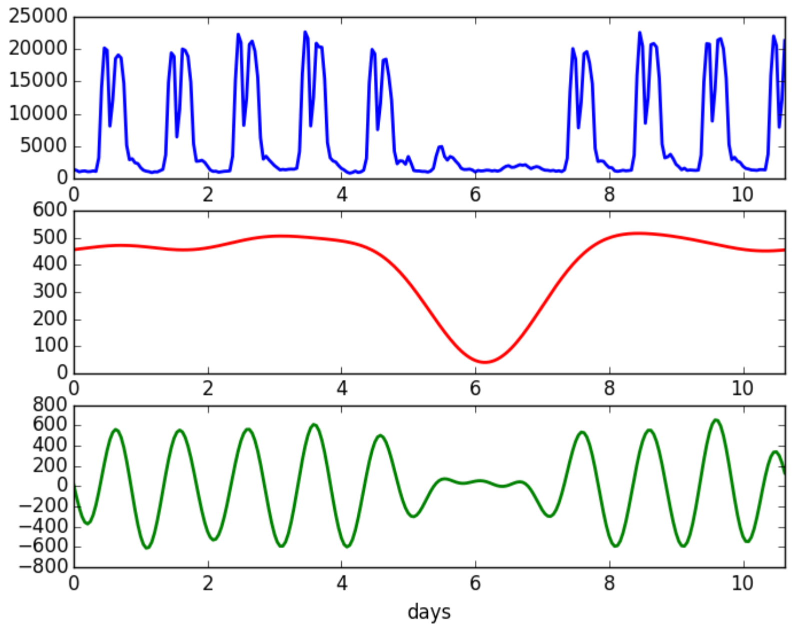

Figure 37: I don't know.

The second graph in the third image is the components of the transform with frequency less than —.46— and under that is components with frequency between —.46— and —1.40—. The author notes that the graph with the red line really shows how on weekends the amount of actions is extremely low and how the green graph shows how nice and fluid the dip is that represents when people are working and when they are not. We get a clear understanding of what exactly is going on if it it were possible to have access to more graph and at more frequencies we would be able to exactly replicate the original graph.

With fourier transform we are able to see things we may not have been able to so in the past and even if we could allows us to understand it even more. With this technique people are able to base their programs and experiments with a much better evidence and data. It is also a really enjoyable thing to do in your free time.