Efficiently Calculating Phase Data from Near and Far Field Magnitudes

Sean Howe advised by Dr. Robert Indik

May 11, 2007

1 The Problem

In optics, the near field is an area close to the source where light is arriving out of phase. The far field is the area at infinity where light is arriving in phase, or in plane-waves. Physically these can be created by diffracting light through a small aperture for the near field, and then refracting it through a lens to create the far field. This system can be seen in Figure 1. Using this set up or a similar one, we are able to measure the intensities of both the near and far fields. Our goal in this research project is to develop an efficient method that uses these measured intensities to reconstruct the full phase data of the original incoming light. In this project we assume the light is coherent, meaning essentially that phase is relevant, and monochromatic, meaning there is a single frequency component. Though we begin with this simple case, the hope is that our methods can eventually be extended to a more general approach with various practical applications, such as creating an infinite depth of field camera or helping to eliminate atmospheric interference in ground based telescopes.

In the fully developed case, the input will be large two dimensional arrays of intensity data taken along a grid of points in the near and far fields. However, in our simplified case, we will instead use one dimensional arrays filled with pre-generated test data.

To develop a method for reconstructing the phase data, we first need to have a mathematical relation between the near and far fields. The Fourier transform is just such a relation and the one dimensional version is given below.

$$\hat{F}(k) = \int_{-\infty}^{+\infty} F(x)e^{-kxi}\,dx \tag{1.1}$$

$\hat{F}$ is the Fourier transform at k and F is the original function. In our case, we take F to be the near field and $\hat{F}$ to be the far field. There is also an inverse to the Fourier Transform which is given below. Notice that the inverse is equal to the original near field function.

$$F(x) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \hat{F}(k)e^{kxi}\,dk \tag{1.2}$$

Here $\frac{1}{2\pi}$ is a normalizing factor. Depending on the application and field of study, the coefficents for the Fourier Transform and Inverse Fourier Transform can be taken differently

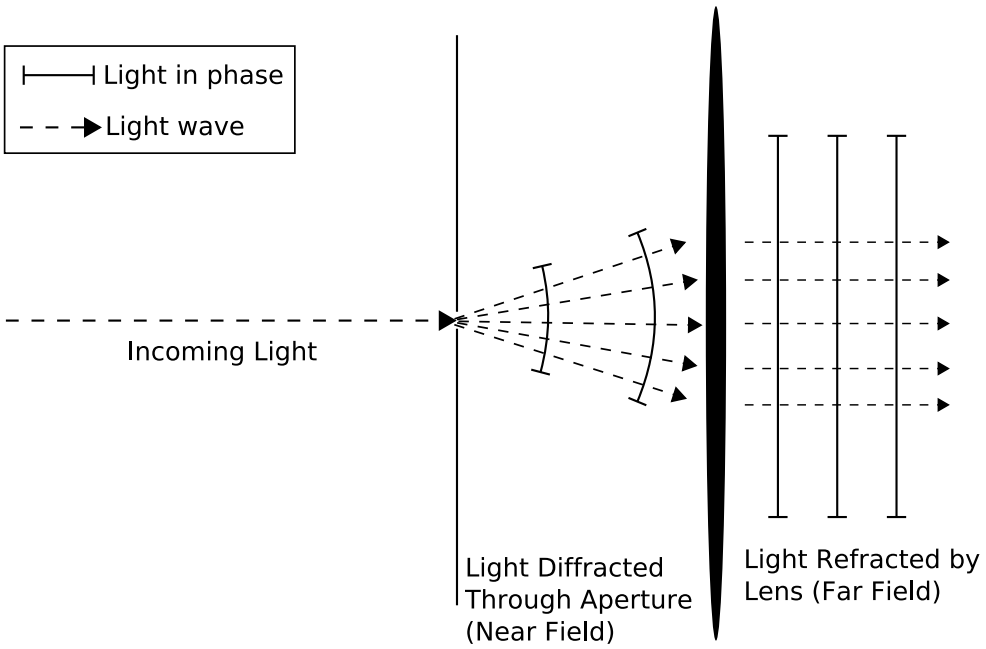

Figure 1.1: A physical setup for separating near and far fields.

so long as their product is $\frac{1}{2\pi}$. It is important to notice that in our case, we cannot use the Fourier Transform in this form because we do not have a continuous function $F(x)$ for the near field, but rather a list of discrete values where we have sampled the intensity. So we instead use a well known approximation. First, we replace the infinite bounds with bounds from 0 to 2π. We then want our discrete step size to be $2\pi ki/N$ so that when our index n iterates up to N we will hit evenly spaced values all the way from 0 to 2π. After finding this step size, simply replace the integral with a Riemann sum and you get the Discrete Fourier Transform, or DFT. A very similar process can be applied to obtain its inverse, the IDFT. Both functions are shown below

$$\hat{F}(k) = \sum_{n=0}^{N} F(n) e^{-2\pi kni/N} \tag{1.3}$$

$$F(n) = \frac{1}{N} \sum_{k=0}^{N} \hat{F}(k) e^{2\pi kni/N} \tag{1.4}$$

As you can see, both the near and far field are complex valued. The real valued measured intensities are the squares of the magnitude of the complex entries as given in equations 1.5 and 1.6.

$$|\hat{F}(k)|^2 = \mathrm{Re}(\hat{F}(k))^2 + \mathrm{Im}(\hat{F}(k))^2 = B_k \tag{1.5}$$

$$|F(n)|^2 = \mathrm{Re}(F(n))^2 + \mathrm{Im}(F(n))^2 = A_k \tag{1.6}$$

In our notation the measured intensities of the near field are contained as entries in A and the measured intensities of the far field are contained as entries in B.

2 Newton's Method

Using the relatons in equations 1.3-1.6, we can build a system of nonlinear equations to describe the field. In the past students have attempted to use Newton's Method to solve for the zeros. Essentially, you start with a guess close to a zero, calculate a tangent vector space and project that to zero, then repeat the process taking the spot you projected to as your new guess. Mathematically, it looks something like this (where G' is the Jacobian matrix):

$$x_{new} = x_{old} - G'^{-1}(x_{old})G(x_{old}) \tag{2.1}$$

While in many case this approach works, there are issues of convergence and more importantly effiicency[1]. Though workable for small test cases, it has at best an O(n^2) complexity which makes it unrealistic for real world applications where there will likely be hundreds of thousands of array entries.

3 The General Fixed Point Iterative Method

Because of the shortcomings of Newton's Method, the focus of this research project is on developing another fixed point iterative method that converges to the solution. Suppose we have a function $G(x)$ such that at a fixed point x, $G(x) = x$. We also say that given a guess, x_n, $G(x_n) = x_{n+1}$. Using a first order approximation we can say that with an initial guess, x_0, that is close to a fixed point,

$$G(x_0) = G(x + \Delta x_0) \approx G(x) + G'(x)\Delta x_0 = x + G'(x)\Delta x_0$$
$$x_1 = x + G'(x)\Delta x_0$$
$$x_1 - x = G'(x)\Delta x_0$$
$$\Delta x_1 = G'(x)\Delta x_0$$
$$G(x_1) = G(x + \Delta x_1) = x + G'(x)\Delta x_1$$
$$x_2 = x + G'(x)\Delta x_1$$
$$x_2 - x = G'(x)\Delta x_1$$
$$\Delta x_2 = (G'(x))^2\Delta x_0$$
Continuing like this it becomes apparent that
$$\Delta x_n = (G'(x))^n\Delta x_0$$
Therefore, if the series is converging such that
$$\lim_{n\to\infty}(G'(x))^n = 0$$
Then we can say that
$$\lim_{n\to\infty} x_n = x$$

If G' is a $\mathbb{R}^n \Rightarrow \mathbb{R}^n$ transformation then we can rewrite it as a $n \times n$ matrix and determine the convergence by its eigenvalues. We define the spectral radius, ρ, as the the largest eigenvalue

of the matrix. We can then say that it is convergent for $\rho(G') < 1$. This means we want all the eigenvalues of $G'(x)$ to have a magnitude less than 1, as seen in Figure 3.1.

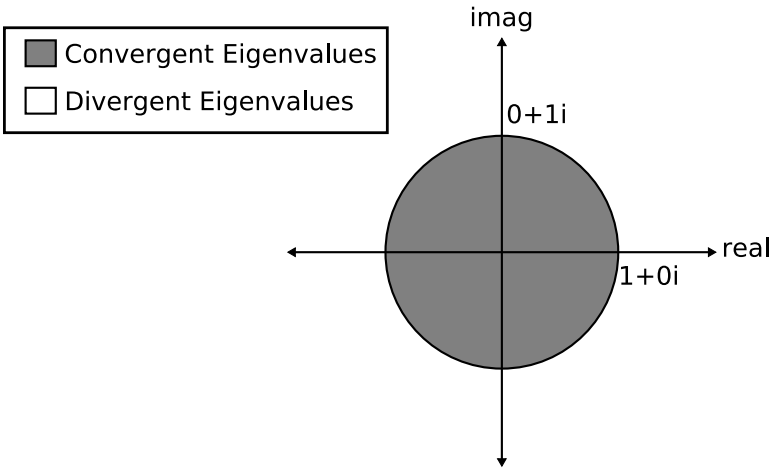

Figure 3.1: Convergent and divergent eigenvalues.

Now, what if we have a function with some divergent eigenvalues? Or what if it just has some that converge too slowly (are very close to 1)? What we need then is a way to map these diverging or slowly converging functions to quickly converging functions. Suppose we have a function $H(x)$ in terms of $G(x)$ iterates. This gives us the following equation.

$$H(x) = a_N G^N(x) + a_{N-1} G^{N-1}(x) + ... + a_1 G(x) \tag{3.1}$$

Now we rewrite

$$H(x) = a_N G(G(G(...(G(x))))) + a_{N-1} G(G(G(...(G(x))))) + ... + a_1 G(x) \tag{3.2}$$

Notice that at a fixed point $G(x) = x$ so for $H(x)$ at the fixed point we can rewrite simply as

$$H(x) = x \sum_{n=1}^{N} a_n \tag{3.3}$$

So, as long as $\sum_{n=1}^{N} a_n = 1$, $H(x)$ is also a valid fixed point iterative function. Now, to show why this is useful, we take the derivative of $H(x)$. At most points this would give us a complicated combination of $G(x)$ and $G'(x)$, but at the fixed point we can write it simply as.

$$H'(x) = a_N (G'(x))^N + a_{N-1} (G'(x))^{N-1} + ... + a_1 G'(x) \tag{3.4}$$

Then, for any given eigenvalue $\lambda_{G'}$ of $G'(x)$ we can obtain the corresponding eigenvalue $\lambda_{H'}$ of $H'(x)$ with the following equation.

$$\lambda_{H'} = a_N \lambda_{G'}^N + a_{N-1} \lambda_{G'}^{N-1} + ... + a_1 \lambda_{G'} \tag{3.5}$$

This means that we should be able to map $G(x)$ functions with divergent derivative eigenvalues to $H(x)$ functions with convergent derivative eigenvalues.

Now we need a way to build a convergent $H(x)$ function. Assume we have a $G(x)$ whose derivative $G'(x)$ has both divergent and convergent eigenvalues. Our strategy is to, one at a time, reduce all the eigenvalues to zero. We start with the first eigenvalue, λ. For any single eigenvalue it is simple to create a quadratic map that fits our criteria of the coefficients adding to one. First, we take the quadratic formula which will solve for x in any equation of the form $ax^2 + bx + c = 0$.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.6}$$

In our case, we have the quadratic equation $a\lambda^2 + b\lambda = 0$. However, since we need to have $a + b = 1$ we can rewrite $b = 1 - a$. Plugging this into equation 3.6 we get

$$\lambda = \frac{-(1-a) \pm \sqrt{(1-a)^2}}{2a} \tag{3.7}$$

Solving for a, we obtain the simple equation

$$a = \frac{-1}{-1 + \lambda} \tag{3.8}$$

So we have now created a polynomial that will reduce one eigenvalue to zero. From it we create a new function, $G_1(x)$ as such.

$$G_1(x) = aG(G(x)) + (1-a)G(x) \tag{3.9}$$

Because the coefficients add up to one and it is composed of $G(x)$ iterates, $G_1(x)$ is also a valid fixed point iterative function. Furthermore, it has special property of having reduced one of the eigenvalues from $G'(x)$ to zero. We now repeat the process to eliminate another eigenvalue from $G(x)$ by creating a $G_2(x)$ similarly composed of $G(x)$ iterates. After creating as many functions as there are eigenvalues, we simply multiply them all together and multiply the coefficients by a factor that reduces their sum to 1, giving us a valid iterative function that converges very quickly.

Unfortunately, this is not the panacea it might seem upon first inspection. While this method would converge quickly, it itself is not a quick method. One iteration of this function involves as many iterations of G as there are eigenvalues of G'. More importantly, it is dependent upon knowing the exact eigenvalues, which in a real world situation we would have no way of doing. I have included it not because it is practical in this form, but to show that something similar to this is a potential method to increase our convergence rate.

4 A Specific Fixed Point Iterative Method

To develop a specific $G(x)$ function we look back at equations 1.3 and 1.4. Imagine that we start with an arbitrary guess, x_0. We then pass it through the DFT, and normalize the result to our far field magnitudes. We then pass the result of that back through the IDFT and normalize that to the near field magnitudes. Back where we started but now with a (hopefully) different vector x_1, we repeat the process. Provided our initial guess is close enough, we hope to see convergence. The process can be seen visually in Figure 4.1.

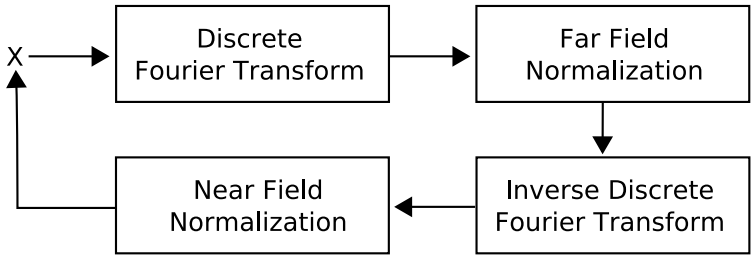

Figure 4.1: Flow diagram of our specific fixed point interative method.

To apply our iterative method, we need to first define a function $G(x)$ that performs the operations from above. We can look at it as a composite function of the DFT, the far field normalization, the IDFT, and the near field normalization as such:

$$G(x) = N \circ F \circ M \circ \hat{F}(x) \tag{4.1}$$

Where we define $\hat{F}(x)$ as the DFT, $M(x)$ as the far normalization function, $F(x)$ as the IDFT, and $N(x)$ as the near normalization function.
$\hat{F}(x)$ and $F(x)$ can be seen in equations 1.3 and 1.4.
$M(x)$ and $N(x)$ we define on a component-wise basis such that

$$x = \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_k \end{bmatrix} \qquad M(x) = \begin{bmatrix} \frac{z_0\sqrt{B_0}}{|z_0|} \\ \frac{z_1\sqrt{B_1}}{|z_1|} \\ \vdots \\ \frac{z_k\sqrt{B_k}}{|z_k|} \end{bmatrix} \qquad N(x) = \begin{bmatrix} \frac{z_0\sqrt{A_0}}{|z_0|} \\ \frac{z_1\sqrt{A_1}}{|z_1|} \\ \vdots \\ \frac{z_k\sqrt{A_k}}{|z_k|} \end{bmatrix} \tag{4.2}$$

Taking the derivative of $G(x)$ using the chain rule we get

$$G'(x) = N'(F(M(\hat{F}(x))))\hat{F}^{-1'}(M(\hat{F}(x)))M'(\hat{F}(x))\hat{F}'(x) \tag{4.3}$$

So to find $G'(x)$ we need the derivatives of $F(x)$, $\hat{F}(x)$, $N(x)$, and $M(x)$. Because these are vector valued functions the derivative will take the form of a Jacobian matrix. A Jacobian

looks like this:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} \qquad f(x) = \begin{bmatrix} y_0 \\ y_0 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \qquad J = \begin{bmatrix} \frac{\delta y_0}{\delta x_0} & \frac{\delta y_0}{\delta x_1} & \cdots & \frac{\delta y_0}{\delta x_k} \\ \frac{\delta y_1}{\delta x_0} & \frac{\delta y_1}{\delta x_1} & \cdots & \frac{\delta y_1}{\delta x_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta y_k}{\delta x_0} & \frac{\delta y_k}{\delta x_1} & \cdots & \frac{\delta y_k}{\delta x_k} \end{bmatrix} \tag{4.4}$$

For the DFT then rewriting it in vector form we get

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} \qquad \hat{F}(x) = \begin{bmatrix} x_0e^0 + x_1e^0 + x_2e^0 + \ldots + x_ke^0 \\ x_0e^0 + x_1e^{-1*1\pi i/N} + x_2e^{-1*2\pi i/N} + \ldots + x_ke^{-1*k\pi i/N} \\ x_0e^0 + x_1e^{-2*1\pi i/N} + x_2e^{-2*2\pi i/N} + \ldots + x_ke^{-2*k\pi i/N} \\ \vdots \\ x_0e^0 + x_1e^{-k*1\pi i/N} + x_2e^{-k*2\pi i/N} + \ldots + x_ke^{-k*k\pi i/N} \end{bmatrix} \tag{4.5}$$

So the Jacobian of the DFT is

$$J_{\hat{F}(x)} = \begin{bmatrix} e^0 & e^0 & \ldots & e^0 \\ e^0 & e^{-1*1*2\pi i/N} & \ldots & e^{-1*k*2\pi i/N} \\ \vdots & \vdots & \ddots & \vdots \\ e^0 & e^{-k*1*2\pi i/N} & \ldots & e^{-k*k*2\pi i/N} \end{bmatrix} \tag{4.6}$$

It is easy to see then that the Jacobian of the IDFT is

$$J_{F(x)} = \frac{1}{2\pi} \begin{bmatrix} e^0 & e^0 & \ldots & e^0 \\ e^0 & e^{1*1*2\pi i/N} & \ldots & e^{1*k*2\pi i/N} \\ \vdots & \vdots & \ddots & \vdots \\ e^0 & e^{k*1*2\pi i/N} & \ldots & e^{k*k*2\pi i/N} \end{bmatrix} \tag{4.7}$$

Now we need to find the Jacobians of the normalization functions. You can see their vectors in 4.2. Notice the presence of the complex magnitude function in the denominators of the $M(x)$ and $N(x)$ vectors. To emphasize this, we rewrite as

$$M(x) = \begin{bmatrix} \frac{z_0\sqrt{B_0}}{\sqrt{\mathrm{Re}(z_0)^2 + \mathrm{Im}(z_0)^2)}} \\ \frac{z_1\sqrt{B_1}}{\sqrt{\mathrm{Re}(z_1)^2 + \mathrm{Im}(z_1)^2)}} \\ \vdots \\ \frac{z_k\sqrt{B_k}}{\sqrt{\mathrm{Re}(z_k)^2 + \mathrm{Im}(z_k)^2)}} \end{bmatrix} \qquad N(x) = \begin{bmatrix} \frac{z_0\sqrt{A_0}}{\sqrt{\mathrm{Re}(z_0)^2 + \mathrm{Im}(z_0)^2)}} \\ \frac{z_1\sqrt{A_1}}{\sqrt{\mathrm{Re}(z_1)^2 + \mathrm{Im}(z_1)^2)}} \\ \vdots \\ \frac{z_k\sqrt{A_k}}{\sqrt{\mathrm{Re}(z_k)^2 + \mathrm{Im}(z_k)^2)}} \end{bmatrix} \tag{4.8}$$

Because the denominator is non-analytic (since it is dependent on both the real and imaginary parts) we are not able to differentiate in terms of z_k like we need to for the Jacobian. To rectify this, we double the size of the vector and split it into real and imaginary parts.

This yields

$$x = \begin{bmatrix} \mathrm{Re}(z_0) \\ \mathrm{Im}(z_0) \\ \mathrm{Re}(z_1) \\ \mathrm{Im}(z_1) \\ \vdots \\ \mathrm{Re}(z_k) \\ \mathrm{Im}(z_k) \end{bmatrix} = \begin{bmatrix} r_0 \\ s_0 \\ r_1 \\ s_1 \\ \dots \\ r_k \\ s_k \end{bmatrix} \qquad M(x) = \begin{bmatrix} \frac{r_0\sqrt{B_0}}{\sqrt{r_0^2+s_0^2}} \\ \frac{s_0\sqrt{B_0}}{\sqrt{r_0^2+s_0^2}} \\ \frac{r_1\sqrt{B_1}}{\sqrt{r_1^2+s_1^2}} \\ \frac{s_1\sqrt{B_1}}{\sqrt{r_1^2+s_1^2}} \\ \vdots \\ \frac{r_k\sqrt{B_k}}{\sqrt{r_k^2+s_k^2}} \\ \frac{s_k\sqrt{B_k}}{\sqrt{r_k^2+s_k^2}} \end{bmatrix} \qquad N(x) = \begin{bmatrix} \frac{r_0\sqrt{A_0}}{\sqrt{r_0^2+s_0^2}} \\ \frac{s_0\sqrt{A_0}}{\sqrt{r_0^2+s_0^2}} \\ \frac{r_1\sqrt{A_1}}{\sqrt{r_1^2+s_1^2}} \\ \frac{s_1\sqrt{A_1}}{\sqrt{r_1^2+s_1^2}} \\ \vdots \\ \frac{r_k\sqrt{A_k}}{\sqrt{r_k^2+s_k^2}} \\ \frac{s_k\sqrt{A_k}}{\sqrt{r_k^2+s_k^2}} \end{bmatrix} \tag{4.9}$$

Now we can take the Jacobian which gives us a deceptively complicated $J_{M(x)}$. It is a block diagonal matrix with blocks of the form

$$\begin{bmatrix} B_k\left(\frac{1}{\sqrt{r_k^2+s_k^2}} - \frac{r_k^2}{\sqrt{r_k^2+s_k^2}^3}\right) & B_k\left(\frac{-r_k s_k}{\sqrt{r_k^2+s_k^2}^3}\right) \\ B_k\left(\frac{-r_k s_k}{\sqrt{r_k^2+s_k^2}^3}\right) & B_k\left(\frac{1}{\sqrt{r_k^2+s_k^2}} - \frac{s_k^2}{\sqrt{r_k^2+s_k^2}^3}\right) \end{bmatrix} \tag{4.10}$$

Notice that the Jacobian for $N(x)$ will be the same just with A coefficients in place of B coefficients. We now have a problem, since our DFT and IDFT Jacobians are $n \times n$ whereas our normalization Jacobians are $2n \times 2n$.

To rectify this we note that $ze^{\theta i} = (r\cos\theta - s\sin\theta) + (r\sin\theta + s\cos\theta)i$ where $r = \mathrm{Re}(z)$ and $s = \mathrm{Im}(z)$. Using this we recalculate the DFT Jacobian as a $2n \times 2n$ matrix.

$$J_{\hat{F}(x)} = \begin{bmatrix} \cos 0 & -\sin 0 & \cos 0 & -\sin 0 & \dots & \cos 0 & -\sin 0 \\ \sin 0 & \cos 0 & \sin 0 & \cos 0 & \dots & \sin 0 & \cos 0 \\ \cos 0 & -\sin 0 & \cos\frac{-1*1*2\pi}{N} & -\sin\frac{-1*1*2\pi}{N} & \dots & \cos\frac{-1*k*2\pi}{N} & -\sin\frac{-1*k*2\pi}{N} \\ \sin 0 & \cos 0 & \sin\frac{-1*1*2\pi}{N} & \cos\frac{-1*1*2\pi}{N} & \dots & \sin\frac{-1*k*2\pi}{N} & \cos\frac{-1*k*2\pi}{N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \cos 0 & -\sin 0 & \cos\frac{-k*1*2\pi}{N} & -\sin\frac{-k*1*2\pi}{N} & \dots & \cos\frac{-k*k*2\pi}{N} & -\sin\frac{-k*k*2\pi}{N} \\ \sin 0 & \cos 0 & \sin\frac{-k*1*2\pi}{N} & \cos\frac{-k*1*2\pi}{N} & \dots & \sin\frac{-k*k*2\pi}{N} & \cos\frac{-k*k*2\pi}{N} \end{bmatrix} \tag{4.11}$$

The Jacobian of the IDFT is nearly identical, just with negative signs removed from the inside of the cos and sin functions and the normalization factor. In fact, if you use equal normalization factors on the DFT and IDFT then the one is actually the transpose of the other.

Now that we have the Jacobians, we need matrix representations of the actual functions. For the DFT and IDFT these are actually the same as the Jacobians, since as you

can see they are all just constant terms. The normalization functions take the following form.

$$M(x) = \begin{bmatrix} B_0(\frac{1}{\sqrt{r_0^2+s_0^2}}) & 0 & 0 & \dots & 0 \\ 0 & B_0(\frac{1}{\sqrt{r_0^2+s_0^2}}) & 0 & \dots & 0 \\ 0 & 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \dots & B_k(\frac{1}{\sqrt{r_0^2+s_0^2}}) & 0 \\ 0 & 0 & \dots & 0 & B_k(\frac{1}{\sqrt{r_0^2+s_0^2}}) \end{bmatrix} \tag{4.12}$$

$N(x)$ looks the same only with A coefficients.

Now that we have all these matrices, we can multiply them together as in equation 4.3 to create our $G'(x)$ matrix. It is this matrix whose eigenvalues we want to study. Let's take a look at a typical near field, its corresponding far field, and the eigenvalues of G' evaluated at the nearfield.

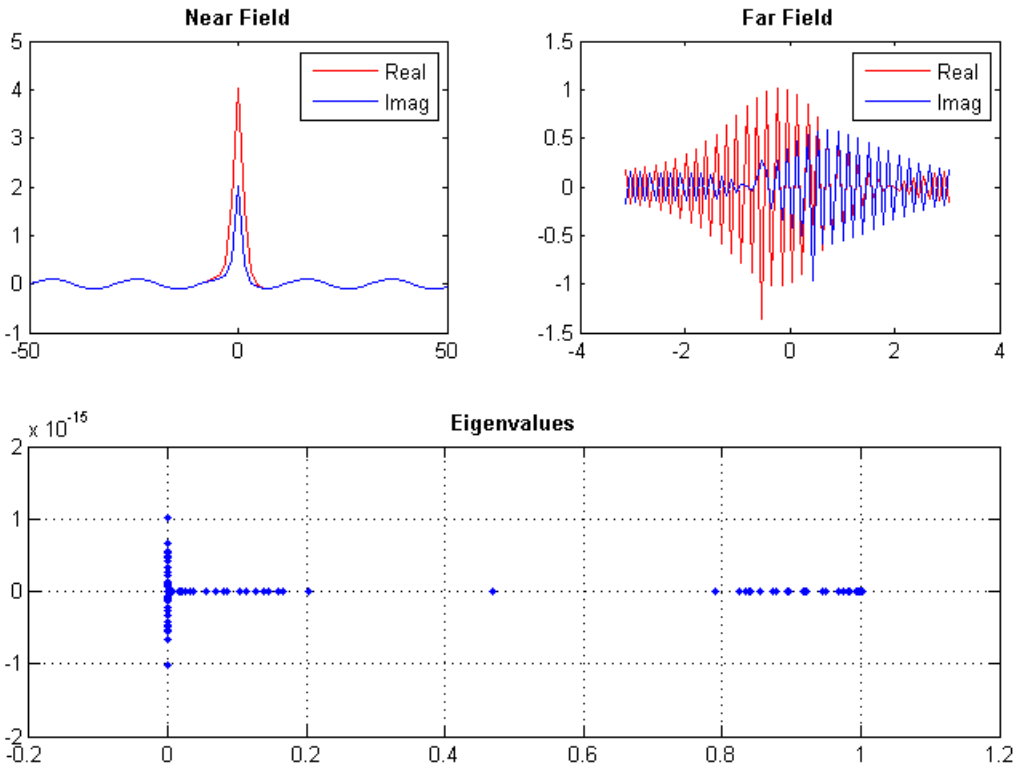

Figure 4.2: A typical near field.

Note the scale on the imaginary axis is very small and that within computational error we have all real eigenvalues between 0 and 1. How exciting! The only non-convergent eigenvalue we have is at exactly one. Experimentally we've seen a situation similar to this for every

near field we've investigated. In fact, we can prove that all of the eigenvalues will always be between -1 and 1.

First, we show that our normalization Jacobian matrices are projection matrices. To do this we factor out some things and rewrite each block:

$$\frac{B_k}{\sqrt{r_k^2+s_k^2}}\begin{bmatrix} 1-\frac{r_k^2}{r_k^2+s_k^2} & \frac{-r_k s_k}{r_k^2+s_k^2} \\ \frac{-r_k s_k}{r_k^2+s_k^2} & 1-\frac{s_k^2}{r_k^2+s_k^2} \end{bmatrix}$$

Using the formula for eigenvalues $\lambda^2 - \mathrm{tr}A\lambda - \det A = 0$ we get

$$\lambda^2 - (2 - \frac{r^2+s^2}{r^2+s^2})\lambda + (1 - \frac{r^2+s^2}{r^2+s^2}) = 0$$
$$\lambda^2 - \lambda = 0$$
$$\lambda(\lambda - 1) = 0$$
$$\lambda = 1, 0$$

So all of its eigenvalues will be either 0 or 1. Notice that with Eigen-decomposition[4] $A = PDP^{-1}$ and $A^2 = PD^2P^{-1}$. Since all entries of the diagonal matrix D in this case will be either 0 or 1, $A = A^2$ and it is therefore by definition a projection matrix. In fact it is actually an orthogonal projection since if you examine the form it is apparent that $A = A^T$. Now, we rewrite equation 4.3 with N referring generically to any normalization function.

$$G'(x) = N(...) \cdot F \cdot N(...) \cdot \hat{F}$$

Notice that $F \cdot N(...) \cdot \hat{F}$ is a similarity transformation of $N(...)$ and so it will have the same eigenvalues and therefore be a projection as well[6]. So $G'(x)$ is the product of two projection matrices. If you imagine a vector going through this transform one step at a time, then it is projected into one vectorspace and then projected again into another. By the very nature of a projection then this process couldn't possibly elongate a vector and so none of the eigenvalues can have an absolute value greater than 1.

We also know that there will be at least one eigenvalue equal to 1 due to the potential for a global phase shift. Our method can only determine relative, not absolute phase, so a solution with a uniform phase shift would be indistinguishable from the actual solution.

Experimentally what we've also found is that all the eigenvalues are positive and that there is only the one eigenvalue actually equal to one. Unfortunately, there seems to be a clump of eigenvalues very close to one, as can be seen in Figure 4.3.

While initially we see fast convergence with this method as the small eigenvalues fall out, the presence of these eigenvalues close to one causes this convergence to slow rapidly to a tortoise crawl. A graph of typical convergence using a guess that differs with small random error at each point in a 127 point near field can be seen in Figure 4.4.

While the exact point it levels out at depends on the sampling size, the field, and the guess, it always levels out to a very slow convergence after a brief spurt of quick convergence. Essentially, the first few iterations are making a big difference but the rest are not. As it stands, this method is unacceptably slow. Part of the difficulty in developing it is we do

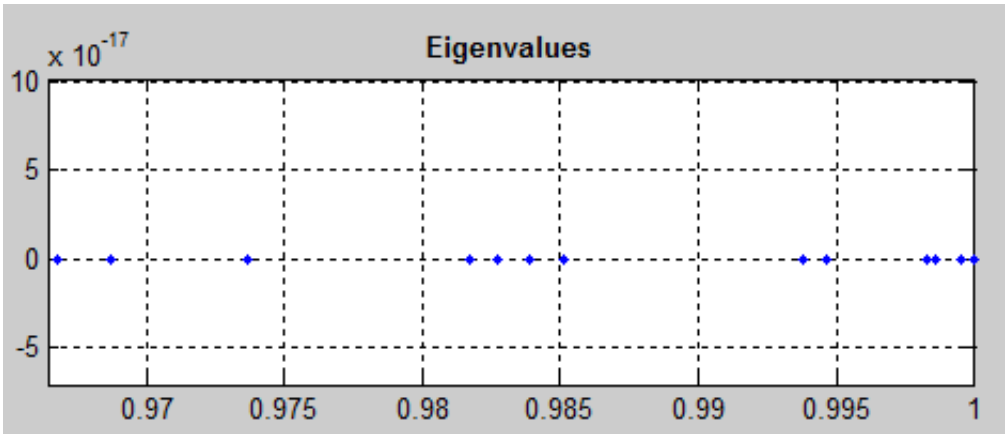

Figure 4.3: Zoomed-in view of eigenvalues close to one.

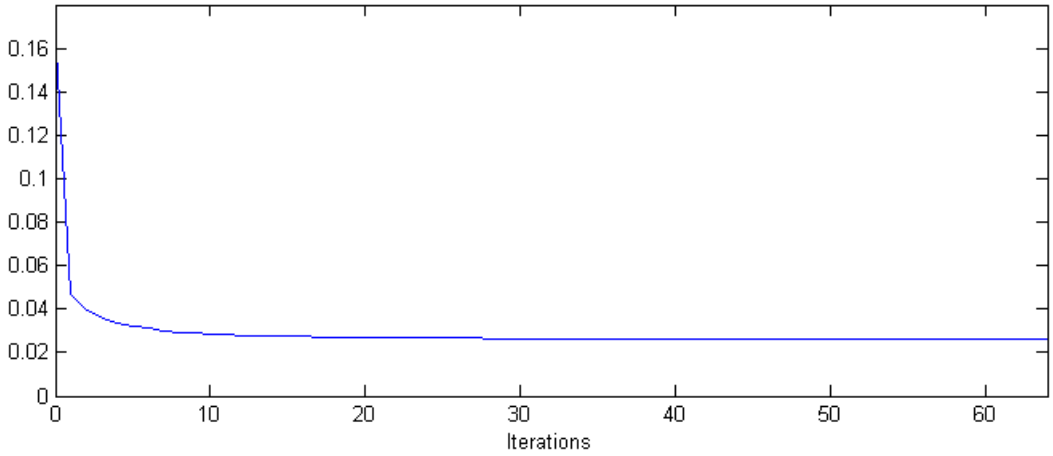

Figure 4.4: Convergence.

not fully understand the effect going through the DFT and IDFT will have, yet we depend on their interaction with the normalization functions to bring each point to the correct rotation around the complex circle. One method we have tried to improve the convergence is to include some forced rotation in the normalization function. While in some (seemingly rare) cases it helps, in many cases it actually makes the function divergent. It has proven elusive so far to find any governing rules for its behavior, but I still maintain an outside hope that we will be able to and then possibly mix the two methods to generate some better convergence. If not, there may be a way to modify the normalization method to take into account more than just one point at a time. Another potential method for obtaining better convergence would involve the polynomial method discussed earlier in this paper, possibly with Tchebyshev polynomials.

5 Acknowledgements

I would like to thank Dr. Indik for his time and effort advising me on this project as well as the University of Arizona and the NSF for sponsoring it through the VIGRE grant.

References

[1] Carlos Chiquette, undergraduate research project, University of Arizona, 2005, available at http://math.arizona.edu/~ura/051/Chiquette.Carlos/Final.pdf

[2] Chris Summitt, undergraduate research project, University of Arizona, 2005, available at http://math.arizona.edu/~ura/051/Summitt.Chris/Final.pdf

[3] Weisstein, Eric W. "Discrete Fourier Transform." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/DiscreteFourierTransform.html

[4] Weisstein, Eric W. "Eigen Decomposition." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/EigenDecomposition.html

[5] Weisstein, Eric W. "Newton's Method." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NewtonsMethod.html

[6] Weisstein, Eric W. "Similarity Transformation." From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/SimilarityTransformation.html

A Source Code

```
%{
 IterateWithRot.m
 Description - DFT->Far Norm->IDFT->Near Norm (with Rot)
 Inputs - inVec: the vector you start with
          NearMag: magnitudes of the near field
          FarMag: magnitudes of the far field
 Outputs - the vector having passed through an iteration
%}
function [outVec] = IterateWithRot(inVec, NearMag, FarMag)
vec=Complexify(inVec);
fftVec=fft(vec);
gamma=-0.01;
for a=1:1:length(fftVec)
    if FarMag(a)~=0
       N=abs(fftVec(a));

       if(N==0)
```

```
            fNormVec(a)=fftVec(a);
        else
    %        fnormVec(a)=fftVec(a)*(FarMag(a)/N)*exp(gamma*i*(N-FarMag(a)));
             fnormVec(a)=fftVec(a)*(FarMag(a)/N);
        end
     else
        fnormVec(a)=0;
     end
end

ifftVec=ifft(fnormVec);

for a=1:1:length(ifftVec)
     if NearMag(a)~=0
        N=abs(ifftVec(a));
        if (N==0)
            nnormVec(a)=ifftVec(a);
        else
            nnormVec(a)=ifftVec(a)*(NearMag(a)/N)*exp(gamma*i*(N-NearMag(a)));
        end
     else
        nnormVec(a)=0;
     end
end

outVec=Decomplexify(nnormVec);


%{
 Complexify
 Description - Takes a nx1 vector and splits it into alternating real and imaginary part
 Inputs - inVec - vector to be complexified
 Outputs - outVec - complexified vector
%}
function outVec=Complexify(inVec)
outVec=zeros(length(inVec)/2,1);
for a=1:1:length(outVec)
     outVec(a)=inVec(2*a-1)+inVec(2*a)*i;
end


%{
 Decomplexify
 Description - Takes a nx1 vector and splits it into alternating real and imaginary part
 Inputs - inVec - vector to be decomplexified
```

```
 Outputs - outVec - decomplexified vector
%}
function outVec=Decomplexify(inVec)
outVec=zeros(2*length(inVec),1);
for a=1:1:length(inVec)
    outVec(2*a-1)=real(inVec(a));
    outVec(2*a)=imag(inVec(a));
end


%{
 CreateNearField
 Description - Creates a Near Field Vector
 Inputs - inFieldParams - a parameters vector containing the following entries
            1 - index of field
            2 - index of imaginary field
            3 - number of elements to place in vector
            4 - magnitude of field
            5 - magnitude of imaginary field
            6 - radius around origin of field
            7 - shift of real field
            8 - shift imaginary field
            9 - number of wavelets
            10 - magnitude of wavelets
             ... more possible for specific fields.
 Outputs - outDerivative
           outXAxis - X axis values for a graph
%}
function [outNearField, outIndices] = CreateNearField(inFieldParams)
numEle=inFieldParams(3);
mag=inFieldParams(4);
magIm=inFieldParams(5);
rad=inFieldParams(6);
shiftReal=inFieldParams(7);
shiftImag=inFieldParams(8);
numWave=inFieldParams(9);
magWave=inFieldParams(10);
firstX=0-rad;
stepX=2*rad/(numEle-1);
outNearField=zeros(numEle,1);

X=firstX+shiftReal;

for a=1:1:numEle
    if inFieldParams(1)==1
```

```
        outNearField(a)=mag*exp(-abs(X+shiftReal));
    elseif inFieldParams(1)==2
        outNearField(a)=mag*1/cosh(X+shiftReal);
    elseif inFieldParams(1)==3
        outNearField(a)=mag*exp(sin(X+shiftReal)*2*pi/(rad*2));
    end

    if inFieldParams(2)==1
        outNearField(a)=outNearField(a)+magIm*exp(-abs(X+shiftImag))*i;
    elseif inFieldParams(2)==2
        outNearField(a)=outNearField(a)+magIm*1/cosh(X+shiftImag)*i;
    elseif inFieldParams(2)==3
        outNearField(a)=outNearField(a)+magIm*exp(sin(X+shiftImag)*2*pi/(rad*2))*i;
    end

    outIndices(a)=X;
    X=X+stepX;
end

t=0;
stepT=((2*pi*numWave)/numEle)
for a=1:1:numEle
    outNearField(a)=outNearField(a)+magWave*sin(t)+magWave*sin(t)*i;
    t=t+stepT;
end


%{
 Iterate.m
 Description - DFT->Far Norm->IDFT->Near Norm
 Inputs - inVec: the vector you start with
          NearMag: magnitudes of the near field
          FarMag: magnitudes of the far field
 Outputs - the vector having passed through an iteration
%}
function [outVec] = Iterate(inVec, NearMag, FarMag)
vec=Complexify(inVec);
fftVec=fft(vec);
for a=1:1:length(fftVec)
    if FarMag(a)~=0
       fnormVec(a)=fftVec(a)*FarMag(a)/(abs(fftVec(a)));
    else
       fnormVec(a)=0;
    end
end
```

```
ifftVec=ifft(fnormVec);

for a=1:1:length(ifftVec)
    if NearMag(a)~=0
       nnormVec(a)=ifftVec(a)*NearMag(a)/(abs(ifftVec(a)));
    else
       nnormVec(a)=0;
    end
end

outVec=Decomplexify(nnormVec);


%{
 IterativeDerivative.m
 Description - Takes a vector of eigenvalues and generates a polynomial using multiplica
 Inputs - NearField: The Near Field you want to evaluate at (decomplexified)
          FarField: The Far Field you want to evaluate at (decomplexified)
          NearMag: The actual near magnitudes
          FarMag: The actual far magnitudes
 Outputs - outDerivative - the derivative evaluated at that point
%}
function [outDerivative, NormNearDerivMat, NormFarDerivMat,NormNearMat,NormFarMat] = Ite
NumEnt=length(NearFieldIn)/2;
FFTMat=zeros(2*NumEnt, 2*NumEnt);
IFFTMat=zeros(2*NumEnt,2*NumEnt);
NormFarMat=zeros(NumEnt*2, NumEnt*2);
NormFarDerivMat=zeros(NumEnt*2, NumEnt*2);
NormNearMat=zeros(NumEnt*2, NumEnt*2);
NormNearDerivMat=zeros(NumEnt*2, NumEnt*2);

%Create DFT, IDFT Mats

for a=1:1:(NumEnt)
    for b=1:1:(NumEnt)
       FFTMat(2*a-1,2*b-1)=cos(-1*2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);
       FFTMat(2*a-1,2*b)=-sin(-1*2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);
       FFTMat(2*a,2*b-1)=sin(-1*2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);
       FFTMat(2*a,2*b)=cos(-1*2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);


       IFFTMat(2*a-1,2*b-1)=cos(2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);
       IFFTMat(2*a-1,2*b)=-sin(2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);
       IFFTMat(2*a,2*b-1)=sin(2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);
```

```
        IFFTMat(2*a,2*b)=cos(2*pi*(a-1)*(b-1)/NumEnt)/sqrt(NumEnt);

    end
end

FarField=FFTMat*NearFieldIn;

%Create far normalization matrix

for a=1:1:NumEnt
    MagConst=FarField(2*a-1)^2+FarField(2*a)^2;
    NormFarMat(2*a-1, 2*a-1)=MagConst^(-1/2)*FarMag(a);
    NormFarMat(2*a-1, 2*a)=0;
    NormFarMat(2*a, 2*a-1)=0;
    NormFarMat(2*a, 2*a)=MagConst^(-1/2)*FarMag(a);

    NormFarDerivMat(2*a-1, 2*a-1)=FarMag(a)*(MagConst^(-1/2)- FarField(2*a-1)^2*MagConst
    NormFarDerivMat(2*a-1, 2*a)=-FarMag(a)*FarField(2*a-1)*FarField(2*a)*MagConst^(-3/2)
    NormFarDerivMat(2*a, 2*a-1)=-FarMag(a)*FarField(2*a-1)*FarField(2*a)*MagConst^(-3/2)
    NormFarDerivMat(2*a, 2*a)=FarMag(a)*(MagConst^(-1/2)- FarField(2*a)^2*MagConst^(-3/2
end

NearField=IFFTMat*NormFarMat*FarField;


%Create near normalization matrix
for a=1:1:NumEnt
    MagConst=NearField(2*a-1)^2+NearField(2*a)^2;
    NormNearMat(2*a-1, 2*a-1)=MagConst^(-1/2)*NearMag(a);
    NormNearMat(2*a-1, 2*a)=0;
    NormNearMat(2*a, 2*a-1)=0;
    NormNearMat(2*a, 2*a)=MagConst^(-1/2)*NearMag(a);


    NormNearDerivMat(2*a-1, 2*a-1)=NearMag(a)*(MagConst^(-1/2)- NearField(2*a-1)^2*MagCo
    NormNearDerivMat(2*a-1, 2*a)=-NearMag(a)*NearField(2*a-1)*NearField(2*a)*MagConst^(-
    NormNearDerivMat(2*a, 2*a-1)=-NearMag(a)*NearField(2*a-1)*NearField(2*a)*MagConst^(-
    NormNearDerivMat(2*a, 2*a)=NearMag(a)*(MagConst^(-1/2)- NearField(2*a)^2*MagConst^(-
end

outDerivative=NormNearDerivMat*IFFTMat*NormFarDerivMat*FFTMat;


%{
 MagicPlot
```

```
 Description - Plots Everything We Want Plotted
 Inputs - inNearField - near field to be plotted
          inFarField - far field to be plotted
          inEigs - eigenvalues to be plotted
 Outputs - outDerivative
%}
function MagicPlot(inNearField,inXNear,inFarField, inEigs)
farX=(-length(inFarField))/2:1:(length(inFarField)/2)-1;
farX=farX*2*pi/length(inFarField);

subplot(2,2,1);
plot (inXNear,real(inNearField),'r');
hold on
plot (inXNear,imag(inNearField),'b');
title('Near Field','FontWeight','bold');
legend('Real','Imag','Location','NorthEast');
hold off;
subplot(2,2,2);
plot (farX,fftshift(real(inFarField)),'r');
hold on
plot (farX,fftshift(imag(inFarField)),'b');
title('Far Field','FontWeight','bold');
legend('Real','Imag','Location','NorthEast');
hold off;
subplot(2,1,2);
plot(real(inEigs), imag(inEigs), '.');
grid on
title('Eigenvalues','FontWeight','bold');
```