Phase Reconstruction Project Midterm Report

Carlos Chiquete

Supervisor: Dr. Robert Indik
12 May 2005
University of Arizona

I. Introduction

The phase reconstruction of optical image data from measured intensities is an important application. When light from astronomical objects passes through the atmosphere of Earth, images captured by CCD cameras are often blurry or distorted by atmospheric turbulence or the variety of refractive indices of the atmosphere. The problem we have been studying involves the development of an effective and efficient algorithm that will seek out a solution (i.e. reconstruct the phase) given two measured intensities. The motivation of this problem lays in its possible application in reconstructing an image that has been perturbed or interfered with such as atmospheric distortion of astronomical images. It is well known that the complex field envelope of the far field (or at infinity) is simply the Fourier transform of the near field complex field envelope (Born & Wolf). It is impractical to measure phase directly, thus it is far more convenient to sample intensities instead. Our goal is to find an algorithm that will construct the different phases associated with two sets of intensity data, one in the "near" field and one in the "far" field.

II. Physics

In the case we wish to study first, light is nearly monochromatic, having the same frequency and traveling in roughly the same direction as well as linearly polarized. In the following case, the image is two dimensional in *x* and *y*. We can represent any wave of this nature as a superposition of complex plane waves. However, since the electric field is real and not imaginary, the superposition must be real. Therefore we add the complex conjugate of the individual plane waves where the amplitude $A(x,y)$ is real valued,

$$\xi(x,y,z,t) = \sum E_j = A(x,y)\exp\left(i(\vec{k}\cdot\vec{r} - \omega t + \varphi(x,y))\right) + A(x,y)\exp(-i(\vec{k}\cdot\vec{r} - \omega t + \varphi(x,y)))$$

We can factor out some terms to get,

$$\begin{aligned}\xi(x,y,z,t) = \sum E_j = A(x,y)\exp(i(k_x x + k_y y))\exp\left(i(k_z z - \omega t + \varphi(x,y))\right) + \ldots \\ A(x,y)\exp(-i(k_x x + k_y y))\exp\left(-i(k_z z - \omega t + \varphi(x,y))\right)\end{aligned}$$

Now we define,

$$\begin{aligned}\hat{A}(x,y) &= A(x,y)\exp(i(k_x x + k_y y)) \quad \textit{and so} \\ [\hat{A}(x,y)]^* &= A(x,y)\exp(-i(k_x x + k_y y))\end{aligned}$$

The addition of the conjugate terms produces the following: simply twice the real part of the previous expression,

$$\xi(x,y,z,t) = 2\left|\tilde{A}(x,y)\right|\cos(k_z z - \omega t + \varphi(x,y))$$

The magnitude of the near field at z=0, over a time is simply proportional to $\left|A(x,y)\right|^2$. This is what we can measure initially. The particular phase is the $\exp(i(k_x x + k_y y)$ term which we cannot directly measure. The next step is to realize that as the light propagates, its far field profile can be measured and using the same process as before we obtain,

$$\xi(x,y,z_{far},t) = 2\left|\hat{f}\left\{\tilde{A}(x,y)\right\}\right|\cos(k_z z_{far} - \omega t + \varphi(x,y))$$

Here we note the complex envelope at the far field is the Fourier transform, $\hat{f}$, of the initial complex envelope (Born & Wolf). Therefore the intensity measured at that point in space is $\left|\hat{\tilde{A}}(x,y)\right|^2$ where the hat denotes the Fourier transform of $\tilde{A}(x,y)$. The phases do not appear in the magnitudes of these coefficients and lead to an ambiguity that will be discussed later. Clearly, the goal is determine the phase information $k_x x + k_y y$ from the two magnitudes that we can sample, i.e. reconstruct the phase.

III. Mathematics

The Fourier Transform

The Fourier transform is central to our problem, thus it is essential to understand the nature the transform mathematically. For now let us focus on the one-dimensional case. The Fourier transform is the generalization of the complex Fourier series. In the Fourier transform, the discrete sum of the Fourier series is replaced with an integral and the associated frequencies are continuous as well. Analogous to the coefficients in the Fourier series, the transform is a projection a set of orthogonal basis functions, $\exp(i\omega t)$. In the continuous sense of the transform, the Fourier transform yields the relative projection across the whole spectrum of frequencies of the function. The Fourier transform functions must have certain characteristics, i.e.,

$$\int_{-\infty}^{\infty}\left|f(t)\right|dt < \infty \quad and \quad \int_{-\infty}^{\infty}\left|f(t)\right|^2 dt < \infty$$

If *f* meets this criteria, the Fourier transform of *f* is,

$$F(f)(\omega) = \hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)\exp(i\omega t)dt$$

Likewise, the inverse Fourier transform is,

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{f}(\omega)\exp(-i\omega t)d\omega$$

In our case we need the discrete Fourier transform since the data we will sample is finite and therefore occurs at discrete mesh points. The discrete Fourier transform uses exponential polynomials that are orthogonal not only over continuous intervals but over discrete intervals (Arfken et al.). Therefore, instead of infinite intervals as in the continuous case outlines above, we use a finite interval and divide it into equally spaced intervals. We consider a time interval T and divide into N sections. The sites correspond to,

$$t_k = kT/N = k\Delta t$$
$$where\ k = 0,1,2 \ldots N-1$$

We then sample a function *f(t_k)* at these finite values of *t* and construct the transform in a manner analogous to the regular Fourier transform, i.e.,

$$\hat{f}(\omega_p) = \sum_{k=0}^{N-1} f(t_k)\exp(i\omega_p t_k)$$
$$where\ \omega_p = \frac{2\pi p}{\mathrm{T}} \text{ for p = 0, 1, 2... N - 1}$$

In this case the ω_p and t_k are conjugate variables. While the regular Fourier transforms measures the complex amplitude at every point of our "real" variable. The discrete transform only measures it at discreet places and for a finite length. The inverse transform is likewise similar to the continuous Fourier transform,

$$f(t_k) = \frac{1}{2N}\sum_{p=0}^{N-1} F(f)(\omega_p)\exp(-i\omega_p t_k)$$

IV. Solving the problem

Stating the Problem

As our starting point, we will attempt to solve the problem for one-dimension and *N* pixels. Therefore, our intensity data are *N* by 1 vectors of real numbers. We must satisfy *2N* equations in the near and far fields. We must have that the solution to our problem satisfy,

$$\eta_j - |Q_j|^2 = 0$$

and,

$$f_j - |\hat{Q}_j|^2 = 0$$

Where the η's are the near-field intensities, and the *f*s are the far-field intensity vectors. The *Q*'s are the elements of the solution vector and are complex valued. The "*Q* hat" terms are the elements of the discrete Fourier transform of *Q*. The formula for "*Q* hat",

$$\hat{Q}_j = \sum_{k=1}^{N} Q_k \exp\left(i \frac{2\pi}{N}(k-1)(j-1) \right)$$

and so we have 2*N* equations along with N unknown *Q*'s. However, the *Q*'s have two unknown parameters since,

$$Q_j = r_j + is_j$$

Where r_j and s_j are the j-th components of two real valued vectors, *r* and *s*. So in fact we have *2N* equations and *2N* variables. However there exists a problem with our equations. Since the sum of the near field and far field magnitudes must be equal, there is a redundant equation in our set of *2N* equations. We now have a system without a unique solution since there are *2N* variables and *2N-1* independent equations. This is important since we must form a 2*N* by 2*N* Jacobian matrix in order to apply the *N* by 1 Newton's method. Even more importantly, a matrix with a redundant equation has determinant equal to zero and therefore we cannot compute the inverse which is vital to Newton's Method. However, we know that there is an overall ambiguity in the phases. If we look at the near-field set of equations,

$$\eta_j - |Q_j|^2 = 0$$

The *Q*'s appear in magnitude, and therefore we may add a phase, $e^{i\theta}$, since

$$\eta_j - |e^{i\theta} Q_j|^2 = \eta_j - |e^{i\theta}|^2 |Q_j|^2 = \eta_j - (e^{i\theta} e^{-i\theta}) |Q_j|^2 = \eta_j - |Q_j|^2 = 0$$

Therefore if we modify all the phases of the *Q*'s with the same θ, and choose θ so that the phase angle of the first *Q* is zero, or that $s_1 = 0$. Consequently, we have that the following for the first equation,

$$\eta_1 - |Q_1|^2 = \eta_1 - ({r_1}^2 + {s_1}^2) = \eta_1 - ({r_1}^2 + (0)^2) = 0$$
$$so,\ r_1 = \sqrt{\eta_1}\ and\ s_1 = 0$$

Therefore we have now *2N-1* equations for *2N-2* unknowns.

As for the far-field equations, they also remain unchanged if we multiply by the same arbitrary factor of $e^{i\theta}$. The equations for the far-field become,

$$F_j = \eta_j - \left|\hat{f}\left(\exp(i\theta)Q_j\right)\right|^2$$

The magnitude squared of the discrete Fourier transform , $\hat{f}\left(\exp(i\theta)Q_j\right)$,

$$\left|\hat{f}\left(\exp(i\theta)Q_j\right)\right|^2 = \sum_{m=1}^{N}\left(\exp(i\theta)Q_j\right)\rho^{(j-1)(m-1)}\sum_{n=1}^{N}\left(\exp(i\theta)Q_j\right)^*\rho^{-(j-1)(n-1)}$$
$$where\ \rho = e^{\frac{2\pi i}{N}},\ and\ (\ )^*\ denotes\ the\ conjugate$$

Combining the two sums we get,

$$\left|\hat{f}\left(\exp(i\theta)Q_j\right)\right|^2 = \sum_{n=1}^{N}\sum_{m=1}^{N}\left(\exp(i\theta)Q_j\right)\left(\exp(-i\theta){Q_j}^*\right)\rho^{(j-1)(m-n)}$$

The $e^{i\theta}$ term disappears since we are multiplying it against its complex conjugate. Therefore the formula simply becomes,

$$\left|\hat{f}\left(\exp(i\theta)Q_j\right)\right|^2 = \sum_{n=1}^{N}\sum_{m=1}^{N}Q_m{Q_n}^*\rho^{(j-1)(m-n)} = \left|\hat{Q}_j\right|^2$$

As we can see the final solution to our problem will always have an overall ambiguity in its phase as it does affect the magnitudes at the near or far fields. Therefore, our removal of the ambiguity is attained by setting the first Q_j to be real in the Newton's Method.

Newton's Method

Newton's method in *N* variables is driven by the assumption that we have an initial guess that is very close to solving the *2N* equations. (Therefore, we must guess an initial set of *Q*'s that is close to the actual solution). If the solution is sufficiently close we may expand using a Taylor series in *N* variables with only linear terms where $x=x_0+\Delta x$ is the actual solution and $\Delta x=x-x_0$ is a small deviation from it,

$$F(x_0+\Delta x)=F(x_0)+J\,|_{x_0}\;\Delta x$$

Assuming that x solves the problem we can say, F(x)=0, so we can solve for x in terms of our guess x_0,

$$0=F(x_0)+J\,|_{x_0}\;(x-x_0)$$
$$0=J^{-1}\,|_{x_0}\;F(x_0)+x-x_0$$
$$x=x_0-J^{-1}\,|_{x_0}\;F(x_0)$$

Therefore we can successively guess solutions to the problem using that,

$$x_{n+1}=x_n-J^{-1}\,|_{x_n}\;F(x_n)$$

In our case the x_n's are the elements of *Q* stacked sequentially in a vector,

$$x_k=r_k \quad for \quad k=1 \;\; to\, N$$
$$x_k=s_{k-N} \quad for \;\; k=N+1 \; to \;\; 2N$$

F is constructed from the equations, $\eta_j-\left|Q_j\right|^2=0$ and $f_j-\left|\hat{Q}_j\right|^2=0$ so that,

$$F_j=\eta_j-\left|Q_j\right|^2 \quad for \quad j=1 \;\; to\, N$$
$$F_j=\,f_j-\left|\hat{Q}_j\right|^2=0 \;\; for \;\; j=N+1 \; to \;\; 2N$$

The remaining term to determine is the Jacobian. It is defined in the *m*-th row and *n*-th column by,

$$J_{mn}=\frac{\partial F_m}{\partial x_n}$$

So the J_{mn} entry in the matrix for m=1 to N is,

$$J_{mn}=-\frac{\partial}{\partial x_n}\left(\eta_m-\left|Q_m\right|^2\right)$$

Since the η_m terms are constant with respect to all x_n and $Q_m=x_m+ix_{m+N}$ the Jacobian entry becomes,

$$J_{mn}=-\frac{\partial}{\partial x_n}\left(x_m{}^2+x_{m+N}{}^2\right)$$

The x_n variables are all independent of each other so $\dfrac{\partial x_j}{\partial x_n}=\delta_{j,n}$,

$$J_{mn}=-2x_m\delta_{m,n}-2x_{m+N}\delta_{m+N,n}$$

As we can see, for the first N rows of J, there will be only 2 non-zero entries, $J_{m,m}$, and $J_{m+N,m}$. Finally, the entries for J_{mn} with *m* between *N+1* to *2N* are given by the following,

$$J_{mn} = \frac{\partial}{\partial x_n}\left(f_m - \left|\hat{Q}_m\right|^2 \right) = \frac{\partial}{\partial x_n}\left|\hat{Q}_m\right|^2$$

As we showed earlier when discussing the overall ambiguity in phase and using $Q_k = x_k + ix_{k+N}$ and $Q_l^* = x_l - ix_{l+N}$,

$$\left|\hat{Q}_m\right|^2 = \sum_{k=1}^{N}\sum_{l=1}^{N} Q_l Q_k^* \rho^{(m-1)(l-k)} = \sum_{k=1}^{N}\sum_{l=1}^{N} (x_l + ix_{l+N})(x_k - ix_{k+N})\rho^{(m-1)(l-k)}$$

Expanding the sum,

$$\left|\hat{Q}_m\right|^2 = \sum_{k=1}^{N}\sum_{l=1}^{N} x_l x_n \rho^{(m-1)(l-k)} - i\sum_{k=1}^{N}\sum_{l=1}^{N} x_l x_{k+N}\rho^{(m-1)(l-k)} + i\sum_{k=1}^{N}\sum_{l=1}^{N} x_{l+N}x_k\rho^{(m-1)(l-k)} + \sum_{k=1}^{N}\sum_{l=1}^{N} x_{l+N}x_{k+N}\rho^{(m-1)(l-k)}$$

However, we must take the derivative with respect to a general x_n , then

$$J_{mn} = \frac{\partial}{\partial x_n}\left(\sum_{k=1}^{N}\sum_{l=1}^{N} x_l x_k \rho^{(m-1)(l-k)} - i\sum_{k=1}^{N}\sum_{l=1}^{N} x_l x_{k+N}\rho^{(m-1)(l-k)} + i\sum_{k=1}^{N}\sum_{l=1}^{N} x_{l+N}x_k\rho^{(m-1)(l-k)} + \sum_{k=1}^{N}\sum_{l=1}^{N} x_{l+N}x_{k+N}\rho^{(m-1)(l-k)} \right)$$

Let us take the derivative in the first term, and using the fact that the derivative can go inside the sum,

$$T_1 = \frac{\partial}{\partial x_n}\left(\sum_{k=1}^{N}\sum_{l=1}^{N} x_l x_k \rho^{(m-1)(l-k)} \right) = \sum_{k=1}^{N}\sum_{l=1}^{N}\frac{\partial}{\partial x_n}(x_l x_k)\rho^{(m-1)(l-k)} = \sum_{k=1}^{N}\sum_{l=1}^{N}\left(x_l\frac{\partial x_k}{\partial x_n} + x_k\frac{\partial x_l}{\partial x_n} \right)\rho^{(m-1)(l-k)}$$

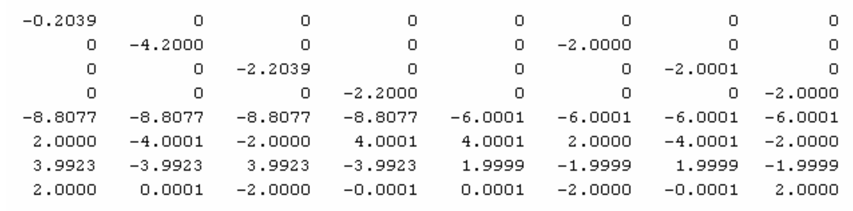
As before, $\frac{\partial x_j}{\partial x_n} = \delta_{j,n}$, therefore we can collapse the sums,

$$T_1 = \sum_{k=1}^{N}\sum_{l=1}^{N} x_l\delta_{k,n}\rho^{(m-1)(l-k)} + \sum_{k=1}^{N}\sum_{l=1}^{N} x_k\delta_{l,n}\rho^{(m-1)(l-k)} = \sum_{l=1}^{N} x_l\rho^{(m-1)(l-n)} + \sum_{k=1}^{N} x_k\rho^{(m-1)(n-k)}$$

Doing the same for the 3 others terms, we arrive at a very surprising result,

$$J_{mn} = -(\rho^{(m-1)(n-1)}\hat{Q}_m^* + \rho^{-(m-1)(n-1)}\hat{Q}_m) \quad for\ k = 1\ to\ N$$
$$J_{mn} = -i(\rho^{(m-1)(n-1)}\hat{Q}_m^* - \rho^{-(m-1)(n-1)}\hat{Q}_m) \quad for\ k = N+1\ to\ 2N$$

A typical Jacobian looks like,



```
-0.2039         0         0         0         0         0         0         0
      0   -4.2000         0         0         0   -2.0000         0         0
      0         0   -2.2039         0         0         0   -2.0001         0
      0         0         0   -2.2000         0         0         0   -2.0000
-8.8077   -8.8077   -8.8077   -8.8077   -6.0001   -6.0001   -6.0001   -6.0001
 2.0000   -4.0001   -2.0000    4.0001    4.0001    2.0000   -4.0001   -2.0000
 3.9923   -3.9923    3.9923   -3.9923    1.9999   -1.9999    1.9999   -1.9999
 2.0000    0.0001   -2.0000   -0.0001    0.0001   -2.0000   -0.0001    2.0000
```

As we can see for N=4, a 8 x 8 matrix is produced for the Jacobian. As noted before, only two entries into the top four rows are non-zero except for row 1. The first row has only one non-zero entry since we have defined $Q_1 = \sqrt{\eta_1} = r_1$. The bottom four rows are all non-zero given that they have far more complex formulae.

We need J to be invertible for Newton's method to give us meaningful answers. Consequently, we are not finished since we know that J is a 2N by 2N matrix and with a redundant column. A redundant column or row in a matrix or simply a row or column that is a multiple of another necessarily produces a matrix that cannot be inverted. Therefore, we choose to eliminate one equation and one variable in hopes of avoiding a singular Jacobian. In our case, the first row and column are deleted from J (recall the r_1 variable is known and therefore x_1 is known) to produce a potentially non-singular reduced Jacobian matrix. This means we must eliminate the first row of *x* as well as the first row of *F*. The new reduced system will produce a new guess that will potentially come closer to the actual solution to *F*. We expect then that Newton's method will converge if we are close enough to the real solution using our initial guess.

The Iterative Method

During the course of discussing the problem, it was suggested we use an iterative algorithm. The motivation for this new approach was that Newton's method was having very large problems with relatively simple functions like simple Gaussians. The new iterative method consists of guessing a solution the *F* vector defined as in the Newton's Method formulation. This guess is the simply square root of the near field intensity. This is obviously a wrong guess (although the top half of the F vector will be necessarily zero), however using a method analogous to carpet beating we can potentially approximate a solution by continuously setting the magnitude of our guess to the intensities. Our first guess is,

$$Q_j^{(0)} = \sqrt{\eta_j}$$

Our second guess is obtained by taking the Fourier transform of the first guess (Q hat) and setting a half step,

$$\hat{Q}_j^{(1/2)} = \frac{\sqrt{f_j}}{\left\|\hat{Q}_j^{(0)}\right\|}\hat{Q}_j^{(0)}$$

This forces the half step guess to have the same magnitude as the far field magnitudes of f_j. In order to arrive at the next guess we say that the next step is,

$$Q_j^{(1/2)} = \Im^{-1}(\hat{Q}_j^{(1/2)})$$

$$\textit{where } \Im^{-1} \textit{ denotes the inverse Discrete Fourier Transform.}$$

Once we have this we define the next step to have the same magnitude as the η_j. So,

$$Q_j^{(1)} = \frac{\sqrt{\eta_j}}{\left\|Q_j^{(1/2)}\right\|}Q_j^{(1/2)}$$

Therefore we have found an algorithm that we may use to try to arrive at the solution to our problem. In terms of a previous guess,

$$Q_j^{(n+1)} = \frac{\sqrt{\eta_j}}{\left\|\Im^{-1}\left(\frac{\sqrt{f_j}}{\left\|\hat{Q}_j^{(n)}\right\|}\Im\left(Q_j^{(n)}\right)\right)\right\|}\Im^{-1}\left(\frac{\sqrt{f_j}}{\left\|\hat{Q}_j^{(n)}\right\|}\Im\left(Q_j^{(n)}\right)\right)$$

The algorithm is very simple to program and execute. The hope is that it will converge to a fixed point. We expect it will take "longer" to arrive at that fixed point in comparison to Newton's Method.

IV. Results

Newton's Method

These algorithms were programmed both in Matlab. After programming and eliminating the multitude of bugs and errors in the code, we were able to get definite results from both algorithms. For Newton's method I used randomly generated sets of data and took the far field and near field magnitudes directly from it. Using these magnitudes, we used Newton's method to approximate a solution and we compared it to the actual solution. The initial guess in our solution was set to be a fixed deviation from the real test solutions.

$$x_{guess} = \mathrm{Re}(x_{actual}) + \Delta + i(\mathrm{Im}(x_{actual}) + \Delta)$$

It was found that the solution's converged very quickly after very few iterations using these randomly generated solutions.

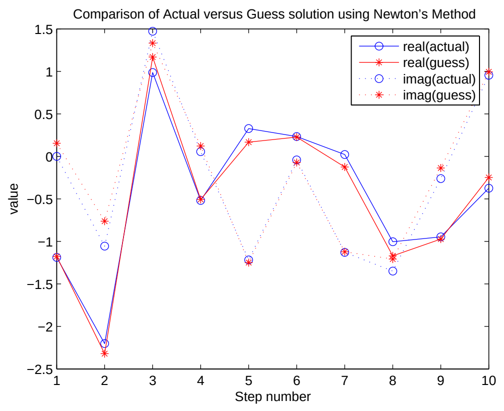


Figure 1

Figure 1 shows the convergence of Newton's method is a tad imperfect but very close overall to the real solution in blue. This lack of convergence may be a feature of the random nature of the solution and therefore its lack of smoothness.

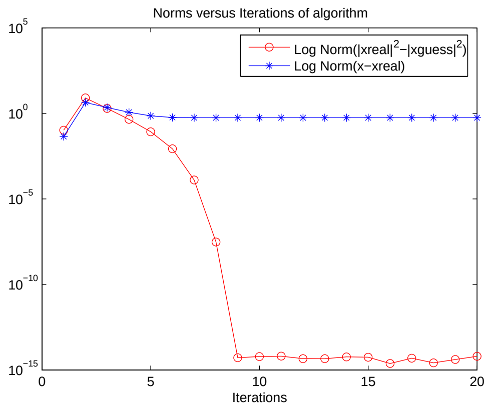


Figure 2

Figure 2 shows that the norms of the squared terms are definitely converging so Newton's Method is working very well. The problem is that the solution that it is finding is slightly different. This owes to the order of our equations gives multiple solutions. This is a problem that requires more study. Additionally, the solutions produced from Newton's Method tended to diverge if the initial offset (Δ) in our test random data was greater than about 0.1.

Next, I tried smooth functions like the following which illustrate that smooth solutions converge nicely:

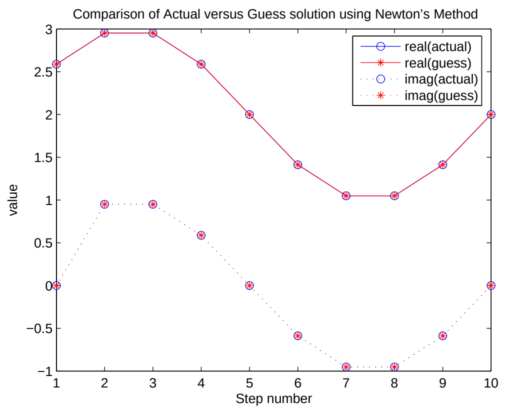


Figure 3

Figure 3 shows that Newton's method converged very smoothly and accurately to the test solution. The next figure shows the norms also converge.

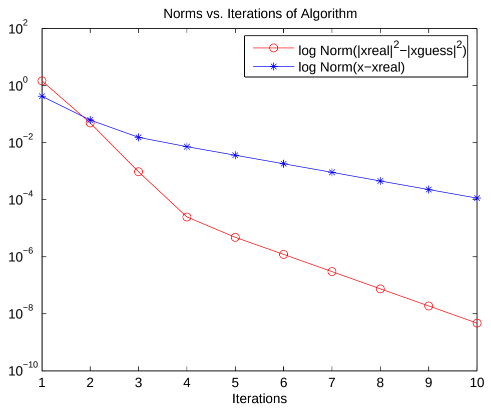


Figure 4

Then we tried a Gaussian function. This function looks like the following,

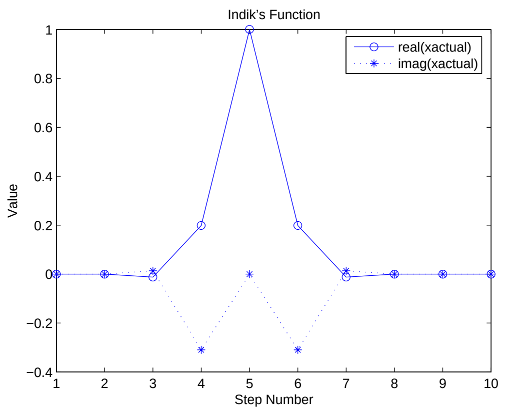


Figure 5

When we attempted to use Newton's Method on this particular function, the iterated guess that was produced diverged greatly from the actual solution.

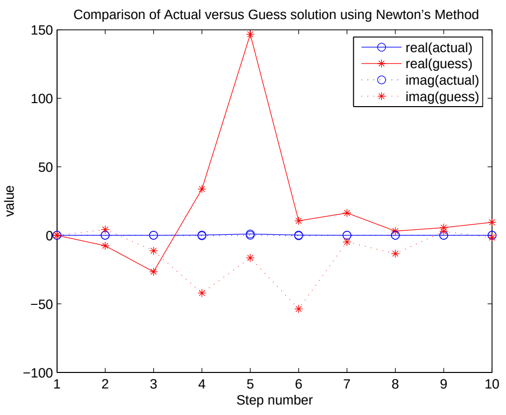


Figure 6

The norms plot (Figure 7) showed that the solution explodes after a few iterations,

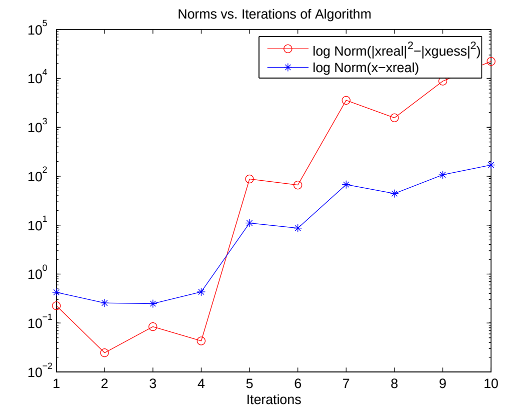


Figure 7

Through testing and debugging, we realized that the problem was that the function we were trying to converge to included one many zeros. The particular solution must be zero at those points. We realized that the Jacobian matrix becomes singular and gives very inaccurate numbers if the solution has entries that are close to zero or are zero. Since we are using the inverse of the Jacobian, the determinant of the inverse Jacobian is very large. This is equivalent to creating a row with only zero entries. This is the source of the singularity in our Jacobian matrix and thus the non-convergence of the solution.

In order to attempt to solve this particular problem I tried to add a certain offset to the real part of the solution to the problem thereby eliminating the troublesome zeros. This worked for the convergence of the solution producing the following plot and convergence,

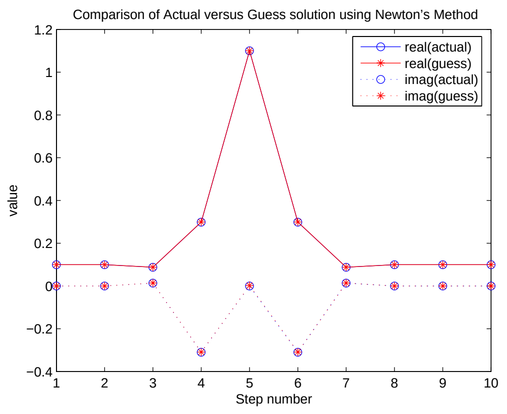


Figure 8

The norms converge very quickly just like in the sine function case (Figure 4).

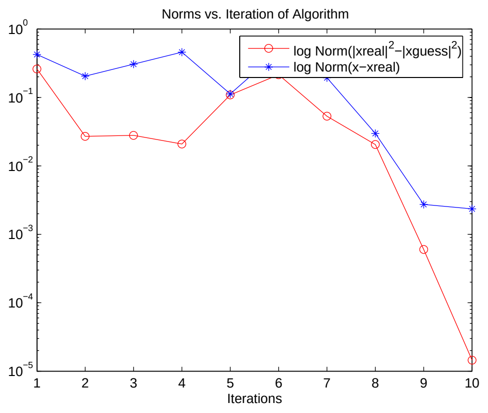


Figure 9

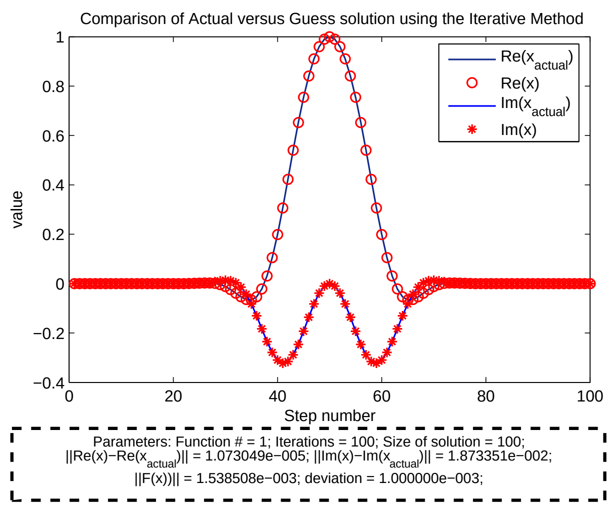
This “fix” however does not solve the problem since in a real world application the solution would be unknown and we would not be able to add such a field. However, this approach has a certain relation to holography: an unexpected result.

Overall, Newton’s Method proved to have both advantages and very serious problems. Its advantage is the speed at which it reaches a solution. Its very precise solutions can be seen in the graph of the norms of F and the difference between the solutions. However, the problems with Newton’s Method are stronger than its advantages. The method does not converge for deviations greater than 1 from the actual solution, and the possible profiles of the complex envelope that can be solved are limited to those functions that do not have zeros. This is a severely limiting characteristic since most real world application will have zeros in the tails. This in addition to the fact that we will not know the solution ahead of time, and so have no idea what the initial guess should be.

The Iterative Method

The results of this particular algorithm were more encouraging. It was found that the Iterative algorithm converges to a solution that is pretty close to the actual solution although very slowly. We built several solution functions and constructed the intensities from these functions. We then tried to find the solution using our algorithm and the intensities. This was an immediate improvement from Newton’s Method since we could approach the problem not knowing anything about the solution except for the same parameters we would know in a real application, the intensities.

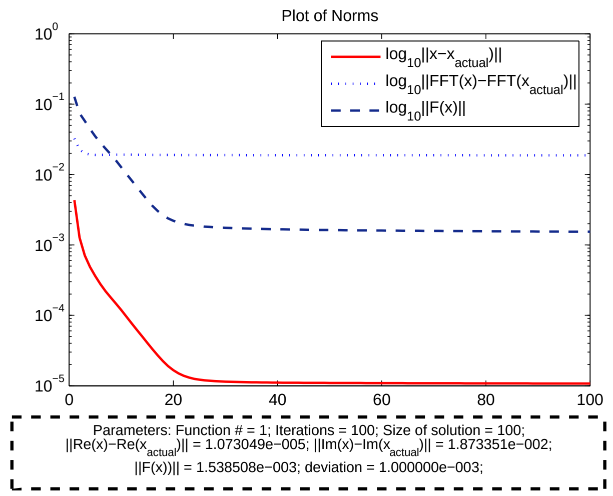
As a starting point a simple Gaussian function was used for the complex envelope. This is the same function that caused Newton’s method to explode. It produced the following plot for the solution,



Parameters: Function # = 1; Iterations = 100; Size of solution = 100;
$||Re(x)-Re(x_{actual})|| = 1.073049e-005$; $||Im(x)-Im(x_{actual})|| = 1.873351e-002$;
$||F(x)|| = 1.538508e-003$; deviation = 1.000000e-003;

Figure 10

The following graph shows the norms that result from the solution shown above,



Parameters: Function # = 1; Iterations = 100; Size of solution = 100;
$||Re(x)-Re(x_{actual})|| = 1.073049e-005$; $||Im(x)-Im(x_{actual})|| = 1.873351e-002$;
$||F(x)|| = 1.538508e-003$; deviation = 1.000000e-003;

Figure 11

As we can see the solution that is produced is not as accurate as for Newton's Method. We also see that the solution seems to level off. This may be due to the half step in the algorithm getting "stuck". However, it also might be possible that as the Iterative method approaches the solution, the rate of convergence is very close to one. This can be seen in the following plot of the spectral maximum of the method's Jacobian,

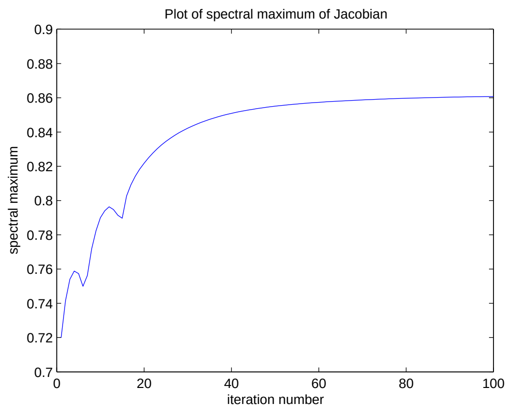
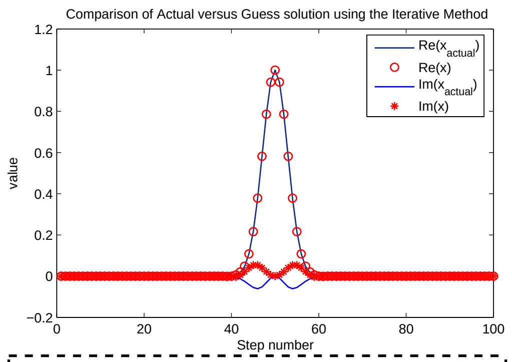


Figure 12

As we can see the spectral maximum changes as the iterations increase. This plot was produced at an initial deviation of 0.1 from the actual solution. We then used the Iterative Method on this initial guess and plotted the spectral maximum for 100 iterations. This graph shows that as we get closer to the solution (as we iterate our method) we are getting slower and slower convergence. It is still converging but very slowly.

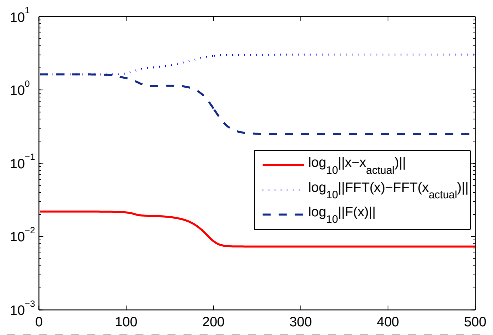
The Iterative method has problems converging to certain functions that are symmetric across its central vertical axis. If the function is symmetric across its center, the near and far field magnitudes have two related solutions. The solution and its conjugate satisfy the equations of *F*. The following Gaussian with a narrower spike than the previous function exhibits this behavior,



Parameters: Function # = 2; Iterations = 500; Size of solution = 100;
$||Re(x)-Re(x_{actual})|| = 7.323048e-003$; $||Im(x)-Im(x_{actual})|| = 3.023523e+000$;
$||F(x))|| = 2.507086e-001$; deviation = 1.000000e−003;

Figure 14

As we can see the solution in read (the iterated guess) is the complex conjugate of the actual solution. This can also be seen in the norms plot,

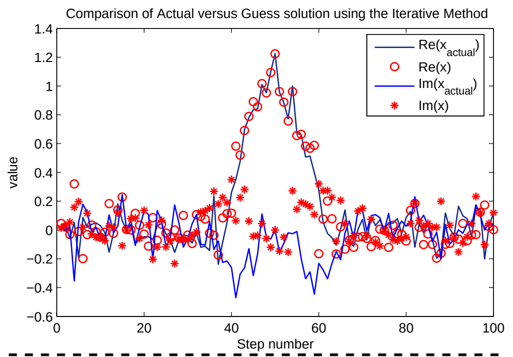


Parameters: Function # = 2; Iterations = 500; Size of solution = 100;
$||Re(x)-Re(x_{actual})||$ = 7.323048e−003; $||Im(x)-Im(x_{actual})||$ = 3.023523e+000;
$||F(x))||$ = 2.507086e−001; deviation = 1.000000e−003;

Figure 15

As we can see the norm of F goes to zero slowly even though the complex part has diverged from the actual complex part.

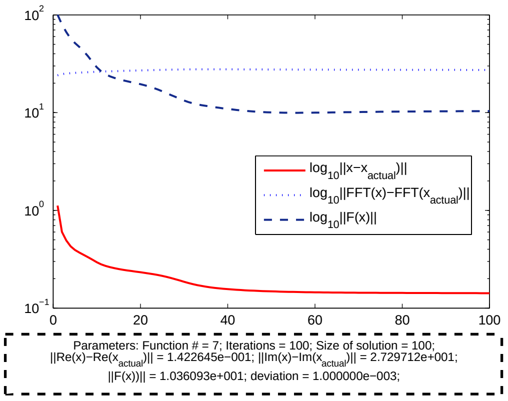
Another type of function that is hard for the Iterative Method to assimilate is solution with random variance. More specifically, a Gaussian with random variance added. The following plot and resultant guess from the Iterative method are shown,



Parameters: Function # = 7; Iterations = 100; Size of solution = 100;
$||Re(x)-Re(x_{actual})||$ = 1.422645e−001; $||Im(x)-Im(x_{actual})||$ = 2.729712e+001;
$||F(x))||$ = 1.036093e+001; deviation = 1.000000e−003;

Figure 16

The norms are not significantly close to zero, especially the norm of F,



Parameters: Function # = 7; Iterations = 100; Size of solution = 100;
$||Re(x)-Re(x_{actual})||$ = 1.422645e−001; $||Im(x)-Im(x_{actual})||$ = 2.729712e+001;
$||F(x))||$ = 1.036093e+001; deviation = 1.000000e−003;

Figure 17

A possible explanation for the lack of convergence of the random solution is that if the actual solution has large random variance. It picks out high frequencies in the Fourier transform. This might cause the Fourier transform to have large values at large frequencies and create problems with the algorithm.

The advantages of the Newton's Method are stronger than the Newton's methods. The fact that there is no requirement for the initial guess to be close to the actual solution is the main advantage over Newton's Method. Our ability to apply this method immediately to a certain set of magnitudes is not available with respect to Newton's method. The problem with convergence to the complex conjugate solution is unavoidable. It is an inherent feature of the way we formulated the problem and we would have to deal with it even with Newton's method. A disadvantage with respect to Newton's method is that the random variance seems to pose no special problem for Newton's while it does pose a problem for the Iterative Method.

VI. Conclusion

Although our problem was restricted to one dimension and particular functions, we can conclude that Newton's method seems to have too many problems with the initial guess problem. The radius of convergence for the Newton's method seems too small to expect any convergence in a real world setting. The Iterative Method holds more promise since it does not need to have an initial guess close the actual solution. In terms of computing, Newton's method requires massive amounts of calculation. For a system with 500 pixels, the inverse of the Jacobian must calculate one million entries.

There are improvements that can be made for Newton's Method that could eliminate the lack of convergence with the zeros in the complex envelope. Just like we removed a variable because of the redundant equation, if an entire row is zero in the full Jacobian, then we know automatically that the associated entry for the associated solution vector is simply zero. The only way for the magnitude of a complex number to be zero is if both real and imaginary parts are zero. Therefore, we can expect to produce a potentially non-zero Jacobian.

If we had further time, we could test a combination of this new and improved Newton's method described above and the Iterative method. If the Iterative method was used for some iterations and hopefully approach the actual solution, we could then use Newton's method with the approximated solution produced by the Iterative method. Potentially, this could allow the combined method to converge much faster to the solution and allow us to sidestep the main flaw of Newton's method. Therefore we could use the advantages of both methods and get rid of a few disadvantages.

There are other considerations that could be pursued with more time. If the solution is very narrow as in the case outline earlier, the solution that is chosen is the conjugate solution. While for the simple Gaussian described before the narrow case, the solution is also symmetric but we almost always get the actual solution. The reasons for this are worth studying. Also, the zero problem for Newton's method arises for numbers that are small, not only for exactly zero. This problem could also be explored in the context of the Jacobian being close to ill-conditioned.

References

M. Born & E. Wolf, *Principles of Optics*, Pergamon Press, New York (1959).

G.B. Arfken, & H.J. Weber, *Mathematical Methods for Physicists*, Academic Press, New York (2000).

Appendix: The Matlab Code

Explanation

(1) newton2.m is simply the algorithm for Newton's method with associated parameters.

(2) plotnorms.m plots the norms associated with our method using various parameters.

(3) plotnewton.m plots the solution and our iterated solution for newton's method

(4) iterativemethod.m is the iterative method algorithm with its associated input parameters.

(5) F.m calculates the F vector for an input vector. Used in calculating its norm.

(6) plotspecial2.m plots the solution and iterated solution for the iterative method.

(7) iterativemethod2 is another version of the iterative method algorithm used in calculating the spectral maxium.

(8) jacobianplot.m plots the range in spectral maximum as a function of iterations of the Iterative Method algorithm.

(9) jacobianim.m calculates the spectral maximum for a single interation.

(10) funct.m is the catalog of functions that is used by the Newton and Iterative Method algorithms.

****

newton2.m

```
function newton2(N,M,e,c);%N-number of pixels, M-Numer of iterations of
algorithm, e-deviation from solution in initial guess, c-switch for type
of plot
clear results;
clear results2;

i=sqrt(-1);
%defining the test function

dx=10/N;%step size (old messed up function)
for j=1:N
    xreal(j,1)=exp(-(1+i)*((j-N/2)*dx)^2)+.1;
end
xreal(1,1)=real(xreal(1,1));
%defining the real solution/function

% dx=3/(N);%step size
% for j=1:N
%     xreal(j,1)=(j)*dx*(j*dx-3)+i*(j)*dx*((j)*dx-3);
% end
% xreal(1,1)=real(xreal(1,1));

%defining the 'measured' magnitudes
nf=conj(xreal).*xreal;
ff=conj(fft(xreal)).*fft(xreal);
```

```
%creating a for loop to guess various answers to the problem and seeing
if
%they converge and producing plot

%for w=1:100 (for testing convergence over a spread of value)
%defining our initial guess for x (or Q). In this case it is a special
case
%that is slightly off the real guess.

r=real(xreal)+(e)*ones(N,1);
s=imag(xreal)+(e)*ones(N,1);
r(1,1)=xreal(1,1);
s(1,1)=0;

%defining matrices and vectors

y=zeros(2*N,1);
yr=zeros(2*N-1,1);
J=zeros(2*N,2*N);
F=ones(2*N,1);
xtest=zeros(N,1);
x=zeros(N,1);

y=[r(1:N);s(1:N)];

yr=[y(2:2*N)];

for n=1:M;%iterations of newton's algorithm
    %forming the solution vector from yr

    for k=2:2*N
        y(k,1)=yr(k-1,1);
    end

    for k=1:N
        x(k,1)=y(k,1)+i*y(k+N,1);
    end
    fftxconj=fft(conj(x));
    fftx=fft(x);
    %defining F
    for j=1:(2*N);
        if j<=N
            F(j,1)=nf(j,1)-power(abs(x(j)),2);
        end

        if j>N
            F(j,1)=ff(j-N,1)-power(abs(fftx(j-N)),2);
        end
    end

    %defining the jacobian, J
    for j=1:2*N;
        %upper half
        if j<=N;
            J(j,j)=-2*real(x(j,1));
            J(j,j+N)=-2*imag(x(j,1));
        end

        %lower half
        if j>N;
            for k=1:N
                J(j,k)=-(exp(-2*pi*(j-1)*(k-1)*i/(N))*conj(fftx(j-
N))+exp(2*pi*(j-1)*(k-1)*i/(N))*fftx(j-N));
            end
            for k=(N+1):2*N
                J(j,k)=-i*(+exp(-2*pi*(j-1)*(k-1)*i/(N))*conj(fftx(j-
N))-exp(2*pi*(j-1)*(k-1)*i/(N))*fftx(j-N));
            end
        end
    end

J



    %defining the reduced J
    Jr=[J(2:2*N,2:2*N)];
    %defining the reduced vector F
    Fr=[F(2:2*N)];
    %creating new reduced solution vector and starting the process over
    %again

    yrnew=yr-Jr\Fr;
    yr=yrnew;


    results(n,1)=norm(nf-conj(x).*x);
    results(n,2)=norm(x-xreal);

end
x(1,1)=real(x(1,1));
%svdx=svd(Jr)
%f=svdx(1,1)/svdx(N-1,1)
    plotnewton([real(xreal),real(x),imag(xreal),imag(x)])
```

****

plotnorms.m

```
function plotnorms(y1,t)
%CREATEFIGURE(Y1)
%  Y1:  matrix of y data

%  Auto-generated by MATLAB on 07-Mar-2005 10:38:12

%% Create figure
figure1 = figure;
```

```
%% Create axes
axes1 = axes('Position',[0.13 0.2262 0.775 0.6988],'Parent',figure1);
s=sprintf('Norms of difference between real and imaginary as well the
residual norm F(x)');
title(axes1,s);
xlim(axes1,[1 t(2)]);
xlabel(axes1,'Iteration number');
ylabel(axes1,'Log_1_0');
box(axes1,'on');

%% Create mutliple lines using matrix input to semilogy
semilogy1 = semilogy(y1);
set(semilogy1(1),...
  'LineWidth',1.5,...
  'LineStyle','-',...
  'Color',[1 0 0]);
set(semilogy1(2),...
  'LineWidth',1.5,...
  'LineStyle',':',...
  'Color',[0 0 1]);
set(semilogy1(3),...
  'LineWidth',1.5,...
  'LineStyle','--',...
  'Color',[0.07843 0.1686 0.549]);
%% Create legend

%legend1 = legend(axes1,{'log_1_0||Re(x)-
Re(x_a_c_t_u_a_l)||','log_1_0||Im(x)-
Im(x_a_c_t_u_a_l)||','log_1_0||F(x)||'});
legend1 = legend(axes1,{'log_1_0||x-x_a_c_t_u_a_l)||','log_1_0||FFT(x)-
FFT(x_a_c_t_u_a_l)||','log_1_0||F(x)||'});

s2=sprintf('Parameters: Function # = %d; Iterations = %d; Size of
solution = %d;\n||Re(x)-Re(x_a_c_t_u_a_l)|| = %e; ||Im(x)-
Im(x_a_c_t_u_a_l)|| = %e;\n||F(x))|| = %e; deviation =
%e;',t(1),t(2),t(6),t(3),t(4),t(5),t(7));

annotation1 = annotation(...
  figure1,'textbox',...
  'Position',[0.04498 0.02143 0.8818 0.1428],...
  'BackgroundColor',[1 1 1],...
  'LineWidth',2,...
  'LineStyle','--',...
  'FitHeightToText','off',...
  'FontName','Arial',...
  'FontSize',9,...
  'HorizontalAlignment','center',...
  'String',{s2});
hold(axes1,'all');
```

****

plotnewton.m

```
function plotnewton(y1)
%CREATEFIGURE(Y1)
%  Y1:  matrix of y data
%  Auto-generated by MATLAB on 07-Mar-2005 16:04:13

%% Create figure
figure1 = figure('FileName','C:\MATLAB7\work\figure2.fig');

%% Create axes
axes1 = axes('Parent',figure1);
title(axes1,'Comparison of Actual versus Guess solution using Newton''s
Method');
xlabel(axes1,'Step number');
ylabel(axes1,'value');
box(axes1,'on');
hold(axes1,'all');

%% Create mutliple lines using matrix input to plot
plot1 = plot(y1);
set(plot1(1),...
  'Color',[0 0 1],...
  'Marker','o');
set(plot1(2),...
  'Color',[1 0 0],...
  'Marker','*');
set(plot1(3),...
  'Color',[0 0 1],...
  'LineStyle',':',...
  'Marker','o');
set(plot1(4),...
  'Color',[1 0 0],...
  'LineStyle',':',...
  'Marker','*');

%% Create legend
legend1 =
legend(axes1,{'real(actual)','real(guess)','imag(actual)','imag(guess)'}
);
```

****

iterativemethod.m

```
function iterativemethod(N,M,t);
%N number of pixels
%M number of iterations
%t number of function, see funct.m
clear xreal;
clear x;
clear results;
results=zeros(N,3);
i=sqrt(-1);
%defining the real solution/function

xreal=funct(N,t);
if xreal==zeros(N,1);
    disp('**************************************');
    disp('************ E R R O R ! *************');
    disp('**************************************');
    disp('Function number not valid');
    return
end

[xrealmax,p]=max(xreal);

xreal=xreal*((((real(xreal(p,1)))^2+(imag(xreal(p,1)))^2)^(1/2))/xreal(p
,1));

nf=conj(xreal).*xreal;
ff=conj(fft(xreal)).*fft(xreal);
e=.001;
%notes: Randomness seems to not matter, same behavior.. appears to
simply
%depend on proximity to original solution. If I start at the solution
the
%solution moves away from it instead of staying put. It goes "up". To
%where? The mystery solution?
%x=xreal+e*(randn(N,1)+i*randn(N,1));
x=sqrt(nf);
%initializing variables

fftx=zeros(N,1);
xf=zeros(N,1);
xn=zeros(N,1);
xnew=zeros(N,1);

for n=1:M
    fftx=fft(x);
    for k=1:N
        if norm(fftx(k,1))==0
            xf(k,1)=0;
        else
            xf(k,1)=(fftx(k,1)*sqrt(ff(k,1)))/(norm(fftx(k,1)));
        end
    end

    xn=ifft(xf);
    xn=xn*((((real(xn(p,1)))^2+(imag(xn(p,1)))^2)^(1/2))/xn(p,1));

    for k=1:N
        if norm(xn(k,1))==0
            xnew(k,1)=0;
        else
            xnew(k,1)=(xn(k,1)*sqrt(nf(k,1))/(norm(xn(k,1))));
        end
    end

%     normxold=norm(F(x,nf,ff));
%     normxnew=norm(F(xnew,nf,ff));

xnew=xnew*((((real(xnew(p,1)))^2+(imag(xnew(p,1)))^2)^(1/2))/xnew(p,1));
    %results(n,1)=(norm(ff-conj(fft(x)).*fft(x)));
    %results(n,1)=norm(imag(xreal)-imag(x));
    %results(n,2)=norm(real(xreal)-real(x));
%     if norm(x-xnew)>e
%        x=xnew;
%     else
%        x=(xn+xnew)/2;
%     end
    %x=xnew;%old way of defining next step
%     if norm(imag(x)-imag(xnew))<.1
%         x=conj(xnew);
%     end


    x=xnew;%new way of defining step
%     if n==M/2
%         x=conj(xnew);
%     end


    results(n,1)=norm(xn-xnew);
    results(n,2)=norm(fft(x)-fft(xreal));
    results(n,3)=norm(F(x,nf,ff));

end




%defining parameters
parameters=zeros(6,1);
parameters(1)=t;%function number
parameters(2)=M;%number of iterations
parameters(3)=results(M,1);%final norm of (real(x-xreal)
parameters(4)=results(M,2);%final norm of (imag(x-xreal)
parameters(5)=results(M,3);%final norm of F(x)
parameters(6)=N;%size of solution
parameters(7)=e;%deviation from solution
plotnorms(results,parameters);%plotting the norms
plotspecial2([real(xreal),real(x),imag(xreal),imag(x)],parameters);
%plotspecial([real(fft(xreal)),real(fft(x)),imag(fft(xreal)),imag(fft(x)
)])


hold off;

************************************************************************
```
F.m

```
function [ans] = F(x,nf,ff)
%Evaluates the F vector at whatever guess for the solution
%inputs is simply teh vector we will evaluate at and the near and
farfield vectors.
dim=size(x);
N=dim(1,1);
fftx=fft(x);
```

```
for j=1:(2*N);
        if j<=N
            ans(j,1)=nf(j,1)-power(abs(x(j)),2);
        end

        if j>N
            ans(j,1)=ff(j-N,1)-power(abs(fftx(j-N)),2);
        end
end
```

****
plotspecial2.m

```
function plotspecial2(y1,t)
%CREATEFIGURE(Y1)
%  Y1:  matrix of y data

%  Auto-generated by MATLAB on 31-Mar-2005 11:22:57

%% Create figure
figure1 = figure('FileName','C:\MATLAB7\work\figure.fig');

%% Create axes
axes1 = axes('Position',[0.13 0.2555 0.775 0.6861],'Parent',figure1);
s = sprintf('Comparison of Actual versus Guess solution using the
Iterative Method','fontsize',14);
title(axes1,s);
xlim(axes1,[0 t(6)]);
xlabel(axes1,'Step number');
ylabel(axes1,'value');
box(axes1,'on');
hold(axes1,'all');

%% Create mutliple lines using matrix input to plot
plot1 = plot(y1);
set(plot1(1),...
  'Color',[0.07843 0.1686 0.549],...
  'LineWidth',1);
set(plot1(2),...
  'Color',[1 0 0],...
  'LineStyle','none',...
  'LineWidth',1,...
  'Marker','o');
set(plot1(3),...
  'Color',[0 0 1],...
  'LineWidth',1);
set(plot1(4),...
  'Color',[1 0 0],...
  'LineStyle','none',...
  'LineWidth',1,...
  'Marker','*');

%% Create legend
```

```
legend1 =
legend(axes1,{'Re(x_a_c_t_u_a_l)','Re(x)','Im(x_a_c_t_u_a_l)','Im(x)'});

%% Create textbox
s=sprintf('Parameters: Function # = %d; Iterations = %d; Size of
solution = %d;\n||Re(x)-Re(x_a_c_t_u_a_l)|| = %e; ||Im(x)-
Im(x_a_c_t_u_a_l)|| = %e;\n||F(x))|| = %e; deviation =
%e;',t(1),t(2),t(6),t(3),t(4),t(5),t(7));
annotation1 = annotation(...
  figure1,'textbox',...
  'Position',[0.04498 0.02143 0.8818 0.1428],...
  'BackgroundColor',[1 1 1],...
  'LineWidth',2,...
  'LineStyle','--',...
  'FitHeightToText','off',...
  'FontName','Arial',...
  'FontSize',9,...
  'HorizontalAlignment','center',...
  'String',{s});
```

****

iterativemethod2.m

```
function [ans] = iterativemethod2(N,M,x,nf,ff,p);
%N number of pixels, M number of iterations (1), x current solution,
%nf near field intensities, ff farfield intensities.
i=sqrt(-1);

%initializing variables
fftx=zeros(N,1);
xf=zeros(N,1);
xn=zeros(N,1);
xnew=zeros(N,1);

for n=1:M
    fftx=fft(x);
    for k=1:N
        if norm(fftx(k,1))==0
        xf(k,1)=0;
        else
        xf(k,1)=(fftx(k,1)*sqrt(ff(k,1)))/(norm(fftx(k,1)));
        end
    end

    xn=ifft(xf);
    xn=xn*((((real(xn(p,1)))^2+(imag(xn(p,1)))^2)^(1/2))/xn(p,1));
    for k=1:N
        if norm(xn(k,1))==0
            xnew(k,1)=0;
        else
            xnew(k,1)=(xn(k,1)*sqrt(nf(k,1))/(norm(xn(k,1))));
        end
    end


xnew=xnew*((((real(xnew(p,1)))^2+(imag(xnew(p,1)))^2)^(1/2))/xnew(p,1));


    x=(xnew+xn)/2;%new way of defining step
end

ans=x;
```

****

jacobianplot.m

```
function [ans]= jacobianplot(N,M,e,e2);
% N number of pixels (should be a power of 2 i.e. 128
% M number of iterations i.e. data points for max(abs(eig(J)))
% e initial deviation of x (initial guess) in both real and imaginary
parts
% e2 deviation used to calculate J... i.e. df/dx=f(x+e2)-f(x-e2)/(2*e)



%declaring particular function
xreal=funct(N,1);
dx=10/N;

%calculates maximum entry of xreal
[maxxreal,p]=max(abs(xreal));

xreal=xreal*((((real(xreal(p,1)))^2+(imag(xreal(p,1)))^2)^(1/2))/xreal(p
,1));

nf=conj(xreal).*xreal;
ff=conj(fft(xreal)).*fft(xreal);

%first guess of x, slightly deviated from real solution
x=xreal-e*(1+i);
for m=1:M
    x=iterativemethod2(N,1,x,nf,ff,p);
    results(m,1)=jacobianim(N,x,nf,ff,e2,p);
end
plot(results);
```

****

jacobianim.m

```
function [ans]= jacobianim(N,x,nf,ff,e,p);
%N number of pixels
%M number of iterations
i=sqrt(-1);
```

```
%defining the real solution/function INDIK FUNCTION


J=zeros(2*N);

x1=x;
x2=x;

for j=1:N
    x1(j)=x1(j)+e;
    x2(j)=x2(j)-e;

    colj=(iterativemethod2(N,1,x1,nf,ff,p)-
iterativemethod2(N,1,x2,nf,ff,p))/(2*e);
    J(1:N,j)=real(colj);
    J(1:N,j+N)=imag(colj);
    x1=x;
    x2=x;

    x1(j)=x1(j)+i*e;
    x2(j)=x2(j)-i*e;
    colj=(iterativemethod2(N,1,x1,nf,ff,p)-
iterativemethod2(N,1,x2,nf,ff,p))/(2*e);
    J(N+1:2*N,j)=real(colj);
    J(N+1:2*N,j+N)=imag(colj);

    x1=x;
    x2=x;
end

ans=max(abs(eig(J)));
```

****

funct.m

```
function [ans] = funct(N,c)
%  Produces function to be used in iterativemethod.m program. All
%  observations refer to that program and not a succesive "better"
program
%  in terms of convergence etc...
%  The function chosen depends on Carlos' whim.
%  N : number of pixels
%  c : choice of function
%  Note: Most of these are derived from initial Gaussian shape
if c==1
    %Function 1:The one that started it all, Indik's symmetric function
(Gaussian)
    %Function properties:
    %Convergence of norm(x-xreal)=0 : Yes
    %Convergence of norm(F(x))=0: Yes
    %Symmetry: Yes
    %Other Notes: interesting flat shape in main hump at center.
Imaginary
```

```
    %part has plenty of curves.
    dx=10/N;
    for j=1:N
        xreal(j,1)=1.0*exp(-(1+i)*((j-2*N/4)*dx)^2);
    end
    ans=xreal;
elseif c==2
    %Function 2 Description: Just like 1, except the hump is much more
    %narrow.
    %Function properties:
    %Convergence of norm(x-xreal)=0 : No
    %Convergence of norm(F(x))=0: No
    %Symmetry: Yes
    %Other Notes: does not converge to real solution due probably to the
    %fact that symmetric solutions will produce two distint solutions
    %(complex conjugates of each other).
    dx=10/N;
    for j=1:N
        xreal(j,1)=exp(-(6+i)*((j-N/2)*dx)^2);
    end
    ans=xreal;
elseif c==3
    %Function 3 Description: flat gaussian thing in the middle, and wide
wings... imaginary part kind of weird
    %Function properties:
    %Convergence of norm(x-xreal)=0 : Yes
    %Convergence of norm(F(x))=0: Yes
    %Symmetry: Yes
    %Other Notes: interesting flat shape in main hump at center.
Imaginary
    %part has plenty of curves.
    dx=10/N;
    for j=1:N
        xreal(j,1)=1.0*exp(-(1+i)*((j-2*N/4)*dx)^2)+(.5)*exp(-(1+i)*((j-
2.5*N/4)*dx)^2)+(.5)*exp(-(1+i)*((j-1.5*N/4)*dx)^2);
    end
    ans=xreal;
elseif c==4
    %Function 4 Description: 3 gaussian humps, clearly separated...
    %Function properties:
    %Convergence of norm(x-xreal)=0 : No
    %Convergence of norm(F(x))=0: Yes
    %Symmetry: Yes
    %Other Notes: convergence seems to depend on the separation of the
    %humps... that is, if the humps are close together there is a chance
    %the conjugate solution is actually encountered. This is the case
here.
    dx=10/N;
    for j=1:N
        xreal(j,1)=1.0*exp(-(1+i)*((j-3*N/6)*dx)^2)+(.5)*exp(-(1+i)*((j-
1.5*N/6)*dx)^2)+(.5)*exp(-(1+i)*((j-4.5*N/6)*dx)^2);
    end
    ans=xreal;
elseif c==5
    %Function 5 Description: a gaussian hump, superposition of 3 smaller
    %humps with weird shape....
    %Function properties:
```

```
    %Convergence of norm(x-xreal)=0 : Yes
    %Convergence of norm(F(x))=0: Yes
    %Symmetry: No
    %Other Notes: A unsymmetric function that converges exactly and
    %quickly. I
    dx=10/N;
    for j=1:N
        xreal(j,1)=1.0*exp(-(1+i)*((j-3*N/6)*dx)^2)+(1.0)*exp(-
(1+i)*((j-2.2*N/6)*dx)^2)+(1.0)*exp(-(1+i)*((j-2.8*N/6)*dx)^2);
    end
    ans=xreal;
elseif c==6
    %Function 6 Description: A decaying exponential modulated by a
simple
    %sine function.
    %Function properties:
    %Convergence of norm(x-xreal)=0: No, real part only.
    %Convergence of norm(F(x))=0: Hell No.
    %Symmetry: No
    %Other Notes: this function really messes with the algorithm. I
think
    %it is because the edges are not exactly zeros.. on both sides. The
    %real part does converge though. A common occurrence.
    dx=10/N;
    for j=1:N
        xreal(j,1)=5+0*i+sin((j-1)*dx)*exp(-j*dx)+i*cos((j-1)*dx)*exp(-
j*dx);
    end
    ans=xreal;
elseif c==7
    %Function 7 Description: just like Funct(1) but with a uniform
random
    %variance.
    %Function properties:
    %Convergence of norm(x-xreal)=Real part only, Imaginary sometimes:
    %Convergence of norm(F(x))=sort of:
    %Symmetry: No, but envelope is.
    %Other Notes: As this is a random solution although carries on a
symmetric envelope (funct(1)),
    %the solution sometimes converge but not very well. It seems about
half the time we
    %have convergence in both norms sometimes not. Additionally, the
norms stall and stop converging very rapidly.
    dx=10/N;
    for j=1:N
        xreal(j,1)=1.0*exp(-(1+i)*((j-2*N/4)*dx)^2);
    end
    for j=1:N
        noise=sqrt(.01)*randn(N,2);
        xreal(j,1)=xreal(j,1)+[noise(1)]+i*[noise(2)];
    end

    ans=xreal;
 elseif c==8
    %Function 8 Description: just like Funct(5) but with a uniform
random
```

```
    %variance.
    %Function properties:
    %Convergence of norm(x-xreal)=0:Real part only consistently. The
    %imaginary part most time converges. I don't know why it finds the
    %conjugate solution...
    %Convergence of norm(F(x))=0: Not really, stalls pretty clearly
above
    %1. I think the randomness is hard to pin down, and there are tiny
    %variations although the guess solution approximates the shape very
    %well. Therefore, there are small differences throughout the
solution.
    %These add up.
    %Symmetry: No, not even the envelope.
    %Other Notes:
    dx=10/N;
    for j=1:N
       xreal(j,1)=1.0*exp(-(1+i)*((j-3*N/6)*dx)^2)+(1.0)*exp(-(1+i)*((j-
2.2*N/6)*dx)^2)+(1.0)*exp(-(5+i)*((j-2.8*N/6)*dx)^2);
    end

    for j=1:N
        noise=sqrt(.1)*randn(N,2);
        xreal(j,1)=xreal(j,1)+[noise(1)]+i*[noise(2)];
    end

    ans=xreal;
    elseif c==9
    %Function 9 Description: just like Funct(5) but with a uniform
random
    %variance and a little hump at the extreme edge.
    %Function properties:
    %Convergence of norm(x-xreal)=0:
    %Convergence of norm(F(x))=0:
    %Symmetry: No.
    %Other Notes:Depending on how much variance, if it is small enough
    %the norms converge, otherwise no. Around .001 I guess.
    dx=10/N;
    for j=1:N
       xreal(j,1)=1.0*exp(-(1+i)*((j-3*N/6)*dx)^2)+(1.0)*exp(-(1+i)*((j-
2.2*N/6)*dx)^2)+(1.0)*exp(-(1+i)*((j-2.8*N/6)*dx)^2)+(.1)*exp(-
(1+i)*((j-5.8*N/6)*dx)^2);
    end

    for j=1:N
        noise=sqrt(1)*randn(N,2);
        xreal(j,1)=xreal(j,1)+[noise(1)]+i*[noise(2)];
    end

    ans=xreal;
    elseif c==10
    %Function 10 Description: Chris' mystery fuction.

    %Function properties: Randomness and an envelope.
    %Convergence of norm(x-xreal)=0:No
    %Convergence of norm(F(x))=0:No
    %Symmetry: No.
    %Other Notes: Using this to test initial small deviations and to see
if
    %the algorithm does anything weird. It does. Near the solution the
    %algorithm reaches a fixed point not affiliated with the actual
    %solution
    t=linspace(-1,1,N);



    x=rand(N,1)+i*rand(N,1);
    for j=1:N
       xreal(j,1)=exp(-10*(t(1,j))^2)*(x(j,1));
    end

    ans=xreal;
else
    ans=zeros(N,1);
end
```