Final Report for spring 2005
Advisor: Robert Indik

Phase Reconstruction using near and far field magnitudes

Chris Summitt

1. Introduction

In optics we study light and how it propagates and interacts with matter. Light can be understood in terms of its color (frequency or wavelength), its directional propagation, its intensity and its phase. Phase can only be measured indirectly. What we can directly measure is intensity. To reconstruct an optical field means to find out what the phase of the light is. Simplified light can be fully described if its phase and intensity are known. By simplified it is meant that the light is monochromatic and moves in only one direction. Light that is monochromatic consists of only one frequency or color. Light that approaches this definition is laser light. The phase and intensity of the light can be represented as a complex number. Phase reconstruction involves using intensity information to construct the phase at some given focal plane. In this project we use intensity data from two different focal planes to try to reconstruct the phase. Phase reconstruction offers many merits, one of which being the ability to overcome optical aberration. If the phase can be reconstructed for simplified light, it is likely a similar process could be used for any type of light.

Intensity information that is needed for phase reconstruction can be gathered from the near and far field magnitudes. We can sample the light's intensities on a grid of size $N \times N$. The complex numbers representing the light's magnitude and phase can be represented as an array of numbers a_{ij}. In order to reconstruct the phase of the desired light beam we need to find the phase of each a_{ij}. We can measure is the intensity at each location, that is, the magnitude squared, $|a_{ij}|^2$, if it is measured in the near field. The far field b_{ij} is approximately the discrete Fourier transform of the near field. Here a_{ij} and b_{ij} are arrays of numbers representing the complex field amplitudes..

2. Constructing the System of Equations

It is helpful to first investigate a special case of the problem. Consider a one dimensional image with 2 pixels. In this special case the system of equations is simplified and approaches by hand can be made. The phase intensity at the near field can be represented as complex numbers Q_0 and Q_1 with real parts r_o and s_o. What can be measured is the magnitude squared of the near field and of the far field. This yields the following set of equations:

$$Q_o = r_o + is_o$$
$$Q_1 = r_1 + is_1$$
$$|Q_o|^2 = r_o^2 + is_o^2 = n_o$$
$$|Q_1|^2 = r_1^2 + is_1^2 = n_1$$

The far field magnitudes can be represented as the discrete Fourier transform of the near field magnitudes. It can be shown that:

$$|Qt_o| = 1/\sqrt{2}\ (Q_o + Q_1), \quad |Qt_o|^2 = f_o$$
$$|Qt_o| = 1/\sqrt{2}\ (Q_o - Q_1), \quad |Qt_1|^2 = f_1$$

The above series of equations can be represented as a system of four equations with four unknowns:

$$r_o^2 + is_o^2 = n_o$$
$$r_1^2 + is_1^2 = n_1$$
$$1/2\ (r_o + r_1)^2 + 1/2\ (s_o + s_1)^2 = f_o$$
$$1/2\ (r_o - r_1)^2 + 1/2\ (s_o - s_1)^2 = f_1$$

The goal now is to solve this system of equations for values of s and r. This system can be reduced to a system of two equations and two unknowns if it is recognized that there exists a redundant equation; consider only positive values for r_o and let $s_o = 0$. The system is reduced to the following:

$$r_o = \sqrt{n_o}$$
$$r_1^2 + is_1^2 = n_1$$
$$1/2\ (r_o + r_1)^2 + 1/2\ (s_1)^2 = f_o$$

3. Using Newton's Method

An obvious approach to the solving this system is to use algebra only. The solution, i.e. the roots for r and s can be found in terms of the near and far field magnitudes but it is quite long and messy, (see appendix A). Iterative techniques can be used to solve systems of equations. Newton's Method is a very effective iterative technique for solving systems of equations. There is a single variable version and a multivariable version. Initially the single variable version was explored; an example can be seen below. Newton's Method involves taking an initial guess at what the root is and refining it until it is as close to the actual root as you want it to be. The refinement is done using the following equation:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Where x_o is the initial guess and x_1 is the refinement. Each time this equation is applied the refined root converges to the actual root quadratically. It is necessary though, that the first guess is somewhat close to the actual answer. While calculating the root by hand is possible, the system of equations will get quite large when constructing a full image. A computer program can be used to execute this algorithm much faster.

Consider the polynomial $X^3 - 19 = 0$
Guess $X_o = 3$, $F'(X) = 3X^2$

$$X_1 = 3 - (3^3 - 19)/3(3)^2$$
$$X_1 = 2.7037037$$

$$X_2 = 2.7037 - (2.7037^3 - 19)/3(2.7037)^2$$
$$X_2 = 2.66886057$$

$$X_3 = 2.66886 - (2.66886^3 - 19)/3(2.66886)^2$$
$$X_3 = 2.66840172$$

Exact answer:
$X^3 = 19 \rightarrow X = 2.668401648$

After only three iterations we find a root that is 6 decimal places. We could continue iterating until a satisfactory accuracy is achieved. Matlab will be used to help solve our systems of equations. We will write a code to implement Newton's method. It is helpful to first code the single variable version of the method and test it for convergence. A code that solves the given example problem from above can be seen in Appendix B. One way to check if the code is working as it should is to look at the convergence of the solution. For Newton's Method, the degree of accuracy of the solution should be squared for each iteration performed. We can check for quadratic convergence by looking at a loglog plot of the error in step n vs. the error in step n+1. It can be shown that the plot should yield a line with slope = 2. This can be seen in figure1.

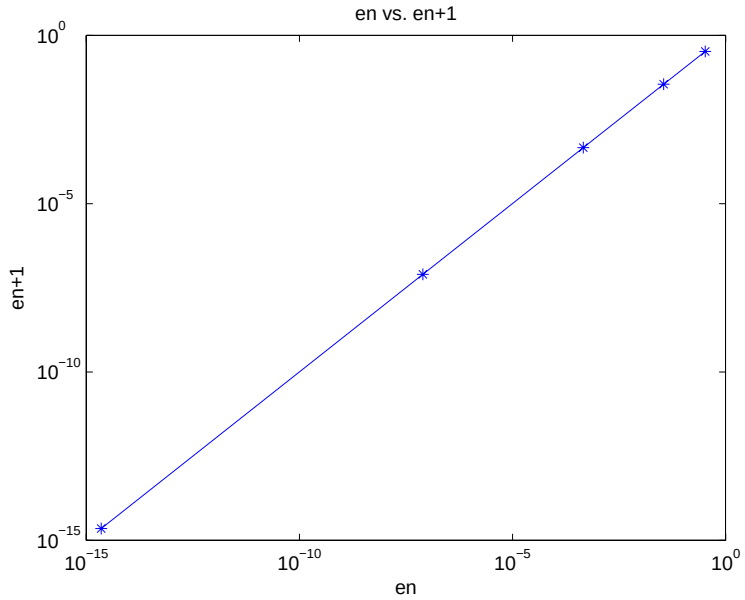


Fig. 1: plot of error in single variable version of Newton's method

The problem of phase reconstruction involves solving a system of equations with more than one variable and hence a multivariable method is necessary. This version works in a similar fashion to the single variable version. An initial guess for the unknown variables is made and then refined until the refined guesses are close to the actual roots. The initial guess must still be close to the actual root. The multivariable method is as follows:

Consider an n dimensional vector

$$y = \begin{pmatrix} y_1 \\ . \\ . \\ y_n \end{pmatrix}$$

and a function vector $F(y) = \begin{pmatrix} f_1(y_1 \ . \ . \ y_n) \\ . \ . \ . \ . \\ . \ . \ . \ . \\ f_n(y_1 \ . \ . \ y_n) \end{pmatrix}$

To solve using Newton's Method we put the problem in the form F(y)=0

The Jacobian is:

$$J = \frac{\partial f}{\partial y} = \begin{pmatrix} \frac{\partial f_1}{\partial y_1} \ldots\ldots\ldots\ldots \frac{\partial f_1}{\partial y_n} \\ \ldots\ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots\ldots \\ \frac{\partial f_n}{\partial y_1} \ldots\ldots\ldots\ldots \frac{\partial f_n}{\partial y_n} \end{pmatrix}$$

The iteration for Newton's Method will then be:

$$y^{n+1} = y^n - J^{-1}F(y^n)$$

Where x_o and y_o are the initial guesses, f_1 and f_2 are the two equations in the system to be solved.

For the system of equations so be solvable we need the form: $Ax = B$, where x must be an invertible matrix. In our system we must require that the Jacobian is a nonsingular matrix. To use this method it appears that the inverse of the Jacobian must be computed. When using Matlab to compute the inverse of the Jacobian multiplied by the function matrix, the inverse is not actually computed; Matlab can perform a more efficient operation that involves Gaussian elimination called LU factorization. This gets harder and harder as the number of equations and variables are increased. Performing the

calculation by hand becomes infeasible and the use of a computer is necessary. We used Matlab to program the two variable version of Newton's Method. The code and the hand worked version can be seen in appendix D. The code involves three parts: the main code and two functions, which define the equations and the Jacobian. The partial derivatives must still be computed by hand but the computer can solve the inverse much faster than can be done by hand.

4. New Iteration

Dr. Indik has proposed a possible new iterative method that does not involve calculating the Jacobian. This method is begun by making an initial guess for the phase intensity at the near field. A good guess is, as stated earlier is to guess the phase is initially zero and to take the square root of the near field intensity.

$$Q_0 = \sqrt{n}$$

This guess for the phase is not correct initially but can be 'moved' to the correct value by using a series of normalizations. As was said before, the far field can be represented as the Fourier Transform of the near field. So the first normalization will be take the Fourier transform of our first guess and multiply by its magnitude.

$$Q_1 = \frac{\sqrt{f}}{\|Q_0\|}\widetilde{Q}_0$$

In the next step the near field is calculated again by taking the inverse Fourier transform of the previous step.

$$Q_2 = F^{-1}(Q_1)$$

Finally, we normalize as before by multiplying by the magnitude:

$$Q_2 = \frac{\sqrt{n}}{\|Q_1\|}\widetilde{Q}_1$$

This algorithm is repeated until the phase achieved matches a known phase that is constructed using the known far field and near field magnitude values. This algorithm was programmed using MATLAB and tested for convergence. The code for this method can be seen in the appendix D. It was hoped that there will be at least linear convergence. While this method does not provide the speed of convergence that Newton's Method does, the problem of taking the inverse of the Jacobian is overcome.

5. Future Work

There is still much work to be done. There are many ideas that have not been tested. The iterative technique devised by Dr. Indik has many parameters that can be varied. Thus far there has only been time to experiment with the variation of one

parameter. By experimenting with the code written to implement this method it is hoped that some kind of pattern can be found. The possibility of finding a solution to this system of equations is still in debate. There are results that suggest there may be an infinite number of solutions around the point where the true, i.e. the desired solution, exists. These results have not yet been confirmed or denied. If this is the case, then there might not be a method for reconstructing the phase using only the near and far field magnitude information. The codes that were written to implement the new iterative technique will undergo further investigation to determine whether or not there is a problem with the way the method was coded or the method itself.

An alternative that was proposed earlier on is to solve the system using Newton's method. This approach is not completely abandoned. There still exists the problem of the existence of a singular matrix, namely the Jacobian. There may be ways around this. A modification to Newton's method that would allow the iteration to be completed without determining the inverse of the Jacobian is desired. This will be one of the driving motivations upon further investigation of this method.

Appendix A

First attempt to solve the one dimensional case by hand using algebra only

System of Equations:

$r_o = \sqrt{n_o}$

$r_1^2 + s_1^2 = n_1$

$1/2\ (r_o + r_1)^2 + 1/2\ (s_1)^2 = f_o$

Algebra:

Consider the first equation: $r_1^2 + s_1^2 = n_1$

by rearranging we get: $r_1^2 = n_1 - s_1^2$

We can now substitute the value for r_1^2 into the second equation:

$½(\sqrt{n_o} + r_1)^2 + ½(s_1)^2 = f_o$

Factoring the first term gives:

$½\ (n_o + 2\ r_1\sqrt{n_o} + r_1^2) + ½(\ s_1^2) = f_o$

Distributing the ½ through gives

$½n_o + r_1\sqrt{n_o} + 1/2\ n_1 - 1/2\ s_1^2 + 1/2\ s_1^2 = f_o$

$=½n_o + r_1\sqrt{n_o} + 1/2\ n_1 = f_o$

$r_1\sqrt{n_o} = f_o - 1/2n_o - 1/2n_1$

Isolating the r_1 term gives:

$r_1 = (f_o - ½n_o - 1/2n_1)/\ \sqrt{n_o}$

Now we need to solve for the second root s_1

$s_1^2 = n_1 - r_1^2$ To do this we need to find r_1^2

Using the r_1 we found earlier we get:

$r_1^2 = [(f_o - 1/2n_o - 1/2n_1)/\ \sqrt{n_o}]\ .\ [(f_o - 1/2n_o - 1/2n_1)/\ \sqrt{n_o}]$

Multiplying the two terms yields:

$r_1^2 = f_o^2 - 1/2n_of_o - 1/2n_1f_o - 1/2n_of_o + 1/4n_o^2 + 1/4n_on_1 - 1/2n_1f_o + 1/4n_on_1 + 1/4n_1^2$

$r_1^2 = (f_o^2 - n_of_o - n_1f_o + 1/2n_on_1 + 1/4n_o^2 + 1/4n_1^2)/\ n_o$

Grouping terms and isolating s_1^2 will give

$s_1^2 = n_1 - (f_o^2 - n_o f_o - n_1 f_o + 1/2 n_o n_1 + 1/4 n_o^2 + 1/4 n_1^2)/ n_o$

Hence, our second root is :

$S_1 = \sqrt{(n_1 - (f_o^2 - n_o f_o - n_1 f_o + 1/2 n_o n_1 + 1/4 n_o^2 + 1/4 n_1^2)/ n_o)}$

Appendix B

This is the one variable version of Newton's Method written for Matlab

```
x=0;
x0=3;
epsilon=1e-12;   %set the decimal accuracy
fx=x0;
fpx=x0;
steps=0;
steparray=[];
errorarray=[];
error2array=[];
exact=19^(1/3);  %determine exact solution
while fx > epsilon
   error=x0-exact;
   errorarray = [errorarray,error];
   fx=x0^3-19;    %function we want root of
   fpx=3*x0^2;    %derivative of function
   xn=x0-fx/fpx;  %newton's method
   x0=xn;
   steps=steps+1;
   steparray = [steparray,steps];
   error2array = [error2array,error];
end
loglog(error2array, errorarray,'*-')
xlabel('en')
ylabel('en+1')
title('en vs. en+1')
```

Appendix C

Using the Multivariable Newton's Method to solve our system of equations

1. Solving the System by Hand

Recall the form of Newton's Method:

$$\begin{bmatrix} r_1 \\ s_1 \end{bmatrix} = \begin{bmatrix} r_0 \\ s_0 \end{bmatrix} - J^{-1}(r_0, s_0) \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} (r_0, s_0)$$

Recall also the system of equations we have:

$$\begin{Bmatrix} r_0 = \sqrt{n_0} \\ r_1^{\,2} + s_1^{\,2} = n_1 \\ \frac{1}{2}(r_0 + r_1) + \frac{1}{2} s_1^{\,2} = f_0 \end{Bmatrix} \text{ this can be written as } \begin{Bmatrix} f_1 = r_1^{\,2} + s_1^{\,2} - n_1 = 0 \\ f_2 = \frac{1}{2}(\sqrt{n_0} + r_1) + \frac{1}{2} s_1^{\,2} - f_0 = 0 \end{Bmatrix}$$

Where r_0 and s_0 are initial guesses and r_1 and s_1 are values closer to the actual root

Recall that J is the Jacobian and is of the form:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial r} & \frac{\partial f_1}{\partial s} \\ \frac{\partial f_2}{\partial r} & \frac{\partial f_2}{\partial s} \end{bmatrix}_{(r_0, s_0)}$$

The partial derivatives can easily be found as:

$$\frac{\partial f_1}{\partial r} = 2r \qquad \frac{\partial f_1}{\partial s} = 2s_1$$

$$\frac{\partial f_2}{\partial r} = r_0 + \sqrt{n_0} \qquad \frac{\partial f_2}{\partial s} = s_1$$

Hence the Jacobian is:

$$J = \begin{bmatrix} 2r_0 & 2s_0 \\ r_0 + \sqrt{n_0} & s_0 \end{bmatrix}$$

And the inverse of the Jacobian can be found to be

$$J = \begin{bmatrix} \dfrac{-1}{2\sqrt{n_0}} & \dfrac{1}{\sqrt{n_0}} \\ \dfrac{r_0 + \sqrt{n_0}}{2 s_0 \sqrt{n_0}} & \dfrac{-r_0}{s_0 \sqrt{n_0}} \end{bmatrix}$$

Now we can use Newton's Method

$$\begin{bmatrix} r_1 \\ s_1 \end{bmatrix} = \begin{bmatrix} r_0 \\ s_0 \end{bmatrix} - \begin{bmatrix} \dfrac{-1}{2\sqrt{n_0}} & \dfrac{1}{\sqrt{n_0}} \\ \dfrac{r_0 + \sqrt{n_0}}{2 s_0 \sqrt{n_0}} & \dfrac{-r_0}{s_0 \sqrt{n_0}} \end{bmatrix} \begin{bmatrix} {r_0}^2 + {s_0}^2 - n_1 \\ \dfrac{1}{2}(\sqrt{n_0} + r_0) + \dfrac{1}{2}{s_0}^2 - f_0 \end{bmatrix}$$

Which can be simplified as:

$$\begin{bmatrix} r_1 \\ s_1 \end{bmatrix} = \begin{bmatrix} r_0 \\ s_0 \end{bmatrix} - \begin{bmatrix} \dfrac{2 r_0 \sqrt{n_0} - 2 f_0 + n_1 + n_0}{2\sqrt{n_0}} \\ \dfrac{{r_0}^2 \sqrt{n_0} + (2 f_0 - n_1 - n_0) r_0 + \sqrt{n_0}({s_0}^2 - n_1)}{2 s_0 \sqrt{n_0}} \end{bmatrix}$$

Where r_0, s_0, f_0, n_0, and n_1 are known values and r_1 and s_1 are the initial guesses after one iteration of Newton's Method. Each iteration the initial guess values will converge to the actual root. It is evident that with each iteration the algebra involved gets very messy. This process is easily executed by a computer program. A simple system of equations can be solved with the Matlab code below.

2. Solving the system using Matlab

Main code:

```
x1=10;
y1=10;
N=[x1;y1];
xo=1;
yo=2;
G=[xo;yo];
d=1e-14;
step=0
exact=[1;1]
while (norm(G-N)>d)
    G=N;
```

```
    step=step+1;
    steps(step)=step;
    error(step)=norm(N-exact);
    f=F(G);
    J=Jac(G);
    N=G-J\f;
end
error(step+1)=norm(N-[1;1]);
steps(step+1)=step+1;
G-N
f
N
```

Functions:

```
function  f=F(G)
f1=3*G(1)^2+G(2)^2-2;
f2=G(1)^2-2*G(2)^2;
f=[f1;f2];

function J=Jac(G)
f1px=[3*2*G(1)];
f1py=[2*G(2)];
f2px=[2*G(1)];
f2py=[-2*2*G(2)];
J=[f1px f1py ; f2px f2py];
```

I used this code as a guide for writing one specific to our problem. The modified code can be seen below.

Main code:

```
global n0 n1 f0
q0=randn(1);
q1=randn(1)+i*randn(1);  %generates a known q value
%q0=1;                   %to test against
%q1=1;
qt0=(q0+q1)/sqrt(2);
qt1=(q0-q1)/sqrt(2);
near=[abs(q0)^2;abs(q1)^2];  %calculate near and far
far=[abs(qt0)^2;abs(qt1)^2]; %field magnitudes
n0=near(1);
n1=near(2);
f0=far(1);
```

```
for count = 1:5    %set number of iterations to perform
    r0=randn(1);
    s0=randn(1);
    r1=randn(1);
    s1=randn(1);
N=[r1;s1];
G=[r0;s0];
d=1e-14;
step=0
%exact=[0;0]
t=cputime;       %calculate program run time
while (norm(G-N)>d)
    G=N;
    step=step+1;
    %error(step)=norm(N-exact);
    f=F(G);
    J=Jac(G);
    N=G-J\f;

    Magnitude_N=(N(1)^2+N(2)^2); %calculates what near field should be
    n1; %actual near field value
    error(step)=[Magnitude_N-n1]

    if step > 3
        break;
    end
end
t=cputime-t
end
%error(step+1)=norm(N-[1;1]);
steps(step+1)=step+1;
G-N;
f
N
Magnitude_N=(N(1)^2+N(2)^2); %calculates what near field should be
n1; %actual near field value
error=Magnitude_N-n1
```

Functions:

```
function  f=F(G)
global n0 n1 f0
f1=G(1)^2+G(2)^2-n1;
f2=(1/2)*(sqrt(n0)+G(1))^2 + (1/2)*G(2)^2-f0;
f=[f1;f2];

function J=Jac(G)
global n0 n1 f0
f1px=[2*G(1)];
f1py=[2*G(2)];
f2px=[G(1)+ sqrt(n0)];
f2py=[G(2)];
J=[f1px f1py ; f2px f2py]
```

Appendix D

Alternate Iterative Method

Dr. Indik suggested an alternative approach for reconstructing the phase using the near and far field magnitudes. This method uses an initial guess for the phase obtained from the square root of the near field magnitude. The guess is normalized in magnitude and the Fourier transform is used to obtain the far field phase. The error was calculated by comparing two known phase intensity values to the values determined using the iterative technique. The convergence of this error and the convergence of the residual error were examined. The results are inconclusive thus far and will be investigated further.

```
%first, set up a known q
n=256;
niter=1000;
qexact=randn(n,1)+i*randn(n,1);
x=linspace(-3,3,n)';
%qexact=exp(i*x.^2+i*x);        %constructing a border for the light
qexact=qexact.*exp(-10*x.^2);
qtexact=fft(qexact);
dk=2*pi/6;
k=[0:n/2,1-n/2:-1]'*dk;      %ensuring light that is paraxial
% qtexact=qtexact.*exp(-.0001*k.^2);
qexact=ifft(qtexact);         %using the inverse Fourier Transform
qexact=qexact/(qexact(n/2)/abs(qexact(n/2)));
qtexact=fft(qexact);
near = abs(qexact).^2;
far = abs(qtexact).^2;

%set up some parameters
errorarray=[];
qarray=[];
q1array=[];
residual=[];
residual_1=[];
delta=1e-10;     %error limit
epsilon=1e-5;    %perturbation limit
sqrt_far=sqrt(far);
sqrt_near=sqrt(near);
q=q*(1+epsilon);
% q=(qexact+(epsilon*(randn(n,1)+i*randn(n,1))));
% q=sqrt(near);
count=1;
success=0;
q=q/sign(q(n/2));
qt=fft(q);
residual=[residual,norm(abs(qt).^2-far)];
% now loop iterations
for(count=1:niter)
    error=norm(q-qexact);  %calculating error
    errorarray=[errorarray,error];
```

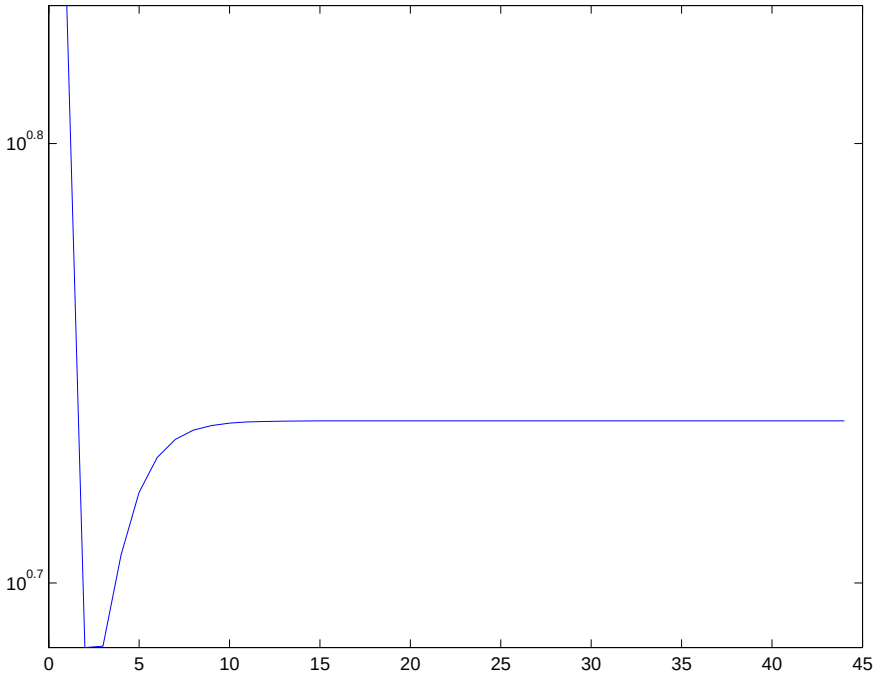
```
    qt1=sign(qt).*sqrt_far;
    q1 = ifft(qt1);
    q1=q1/sign(q1(n/2));

   % q=q/sign(q(n/2));
    q=(q+q1)/2;
    q=q/sign(q(n/2));
    qt=fft(q);
    residual=[residual,norm(abs(qt).^2-far)];  %checking residual ratio
    if norm(residual(end))<delta
        success=1
        break
    end
    % qarray=[qarray,q];
    % q1array=[q1array,q1];
end
check=residual(end-10:end)./residual(end-11:end-1)
semilogy(residual,'x')
title('Plot of the log of residual using a  perturbation with
epsilon=.00001 (n=256)')
```

Some results from the convergence testing:

Similar results were expected for both the error and the residual. This was not the case. The residual always seemed to converge while the error seemed to converge but then got hung up on a constant. One reason for this could be the existence of multiple solutions. The fact that *a solution* is found by the algorithm would force the residual to converge but if the solution found is not the exact solution that was initially created as a known phase then the error will not converge. See following figures.

Plot of the log of Error using a perturbation with epsilon=.00001 (n=256)



Plot of the log of residual using a perturbation with epsilon=.00001 (n=256)