ITERATIVE METHODS FOR SOLVING NON-LINEAR SYSTEMS

Jordan Leiker | GTid: 903453031
Georgia Institute of Technology | MATH 6644

ABSTRACT

In practice, many systems of interest are non-linear and require methods beyond standard linear solvers, such as gradient descent, to efficiently find solutions.

This paper will focus on methods for solving non-linear systems of equations, including the Fixed-Point method and Newton's Method (plus variations such as Chord and Shamanskii Method) as well as discussing how Newton's Method can be used to find multiple roots in a system of non-linear equations.

Index Terms— Iterative Methods, Non-Linear, Newton, Chord, Shamanskii, Fixed-Point

1. INTRODUCTION

The focus of this paper is to compare methods for solving non-linear systems, as well as explore common problems and short-comings of non-linear methods.

This paper is divided into two sections. Section A focuses on solving the Chandrasekhar H-equations using the Fixed-Point, Chord, Newton, and Shamanskii methods. The results from each of these methods are analyzed and compared. Section B focuses entirely on how Netwon's method can be used to find all roots in a non-linear system of 3 equations and 3 variables.

All work described in this paper was written in Python 3.7. To replicate this work, or run the code, the following libraries are required (all of which are part of the Anaconda distribution):

- NumPy – Used for all number manipulations
- SciPy – For LU factoring/solving
- MatPlotLib – Used for plotting
- Pandas – For console print formatting

2. SECTION (A) – CHANDRASEKHAR H-EQUATIONS

2.1 Approach (A)

This section will cover the design decisions made while writing code to generate the non-linear system, iterative methods, and any necessary functions.

2.1.1. Chandrasekhar H-Equations

The discrete version of the Chandrasekhar H-Equations is shown in (1):

$$\boldsymbol{F_i}(\vec{x}) = x_i - \left(1 - \frac{c}{2N}\sum_{j=1}^{N}\frac{\mu_i x_j}{\mu_i + \mu_j}\right)^{-1} = 0 \qquad (1)$$

Where $c \in (0,1)$ and $\mu_{<i,j>} = (i - 1/2)/N$ for $1 \leq i \leq N$and N is the dimension of the unknown vector $\vec{x}$.

The Chandrasekhar expression in (1) implemented as a Python function:

```
def chandrasekharHEq(x, c):
    n = x.shape[0]
    # Work in single dimension
    fx = np.zeros(n)
    x = np.ravel(x)
    m = (np.arange(1,n+1)-0.5)/n
    for i in range(n):
        sum_j = np.sum((m[i] * x)/(m[i] + m))
        fx[i] = x[i] - (1-(c/(2*n))*sum_j)**(-1)
    return np.atleast_2d(fx).T # return "2d",i.e. nx1
```

2.1.2. Fixed-Point Method

The first, and simplest, method used to solve the Chandrasekhar equations from Section 2.1.1 is the Fixed-Point Method. The Fixed-Point algorithm is shown in (2) and is derived from $F(x) = 0$, adding x to both sides, and converting the x's into iterates:

$$\vec{x}^{(n+1)} = \vec{x}^{(n)} - F\left(\vec{x}^{(n)}\right) \qquad (2)$$

The actual implemented version uses the convergence criteria $\|\boldsymbol{F}(x)\| \leq \tau_r r_o + \tau_a$ (where $\tau_r = \tau_a = 1e^{-6}$), or optionally breaking when a maximum number of iterations is reached. The function returns the solution, number of iterations, and a vector containing the error at each iteration. The Python code is:

```
def fixedPointMethod(func_F,x,tau,maxIters):
  # Intialize
  f_x = func_F(x)
  r_0 = np.linalg.norm(f_x)
  err = r_0
  err_vec = [err]
  numIters = 0
```

```
  while (err>(tau[0]*r_0+tau[1]))and(numIters<maxIters):
      x = x - f_x
      f_x = func_F(x)
      err = np.linalg.norm(f_x)
      err_vec.append(err)
      numIters += 1

  return x, numIters, err_vec
```

2.1.3. Difference Jacobian

To efficiently compute the Newton Method (and variants Chord/Shamanskii) special care must be given to the Jacobian calculation since it is the most costly part of Newton. This paper uses a finite-difference Jacobian to efficiently compute the Jacobian matrix as well as "black box" the function being analyzed. In this way any system of equations can re-use this Jacobian calculator without any knowledge of the actual equations.

The `diffjac` code, as provided by T. Kelley [1], for Jacobian calculations was suggested for use but unfortunately this code was only available in MATLAB. To perform the analysis described in this paper, `diffjac` and supporting function `dirder` were ported from MATLAB to Python. The code is *nearly* identical between the two languages with the exception of some small differences, mainly in syntax and necessary libraries.

These two functions implementing the finite difference Jacobian, which is used in the Newton-based methods described in Section 2.1.4. are shown. First is `dirder`, which is a function used to calculate a finite difference directional derivative; and second is `diffjac`, the finite difference Jacobian that calls `dirder` while iterating over every column of the Jacobian.

```
def dirder(x, w, f, f0):
    # Hardwired difference increment.
    epsnew = 1.e-7
    n = x.shape[0]

    # scale the step
    if np.linalg.norm(w) == 0:
        z = np.zeros((n, 1))
        return z

    epsnew = epsnew / np.linalg.norm(w)
    if np.linalg.norm(x) > 0:
        epsnew = epsnew * np.linalg.norm(x)

    # del and f1 could share the same space if storage
    # is more important than clarity
    delx = x + epsnew * w
    f1 = f(delx)
    z = (f1 - f0) / epsnew
    return z
```

```
def diffjac(x, f, f0):
    n = x.shape[0];
    jac = np.zeros((n, n))

    for j in range(n):
        zz = np.zeros((n, 1))
        zz[j] = 1
        jac[:, j] = np.ravel(dirder(x, zz, f, f0))

    return jac
```

2.1.4. Newton-Based Methods

The Newton based methods, i.e. Newtons Method, Chord Method, and Shamanskii Method, were all implemented in the same Python function, `nonlinearMethods`, as they are based on the same equation (3), where $F'(\vec{x})$ is the Jacobian for the function $F(\vec{x})$.

Notice that the Fixed-Point method in (2) and Newtons Method in (3) are incredibly similar with the exception of (3) including the inverse Jacobian matrix. Recall that the Fixed-Point method is derived from $F(x) = 0$, but instead of adding x to each side of the equation and iterating, Newton's method takes the Taylor Expansion of $F(\vec{x})$ - in this case to the first derivative - and then solves for x. This is where the Jacobian (i.e. the derivative in 1-dimension) is introduced, and what allows Newton's method to converge quadratically.

$$\vec{x}^{(n+1)} = \vec{x}^{(n)} - \left(F'\left(\vec{x}^{(n)}\right)\right)^{-1} F\left(\vec{x}^{(n)}\right) \tag{3}$$

The distinction between Newton-based methods is the number of times the Jacobian is recalculated while the algorithm iterates. For the methods, the Jacobian is calculated the following number of times:

Table 1. Jacobian Calculations per Method

Method	**Jacobian Updates**	**Function "m" value**
Newton's Method	Every iteration	1
Shamanskii Method	Every m iterations	$1 < m < \infty$
Chord Method	Only the first iteration	∞
Fixed-Point Method	Never	N/A

The `nonlinearMethods` function was written to select between the three methods, Newton-Chord--Shamanskii, by simply selecting the value of m passed as argument (Table 1 for details on m).

In Newton's Method, depending on the structure of the Jacobian, any iterative algorithm or matrix inverse solver could be used to solve the Jacobian system $F'(x)\delta = F(x)$. For the analysis done in this paper, the decision was made to limit the `nonlinearMethods` function to only use only LU factorization + LU solving. The reason for this is because while the Chord and Shamanskii method do not repeatedly calculate the inverse Jacobian, they would benefit greatly from the increased speed of solving an LU factorized Jacobian system (recall that solving an LU system is just forward-reverse substitution). Thus, the decision was made to force all three methods to only use LU factorization + LU solving instead of leaving the Jacobian calculation as user customizable, as is the case with the "basic Newton's method" used in Section B.

Again, as seen in the Fixed-Point method, the convergence criteria $\|\boldsymbol{F}(x)\| \leq \tau_r r_o + \tau_a$ (where $\tau_r = \tau_a = 1e^{-6}$) was used.

```
def nonlinearMethods(func_F, x, tau, maxIters, m=1):
 # Intialize
 f_x = func_F(x)
 r_0 = np.linalg.norm(f_x)
 err = r_0
 err_vec = [err]
 numIters = 0

 while (err>(tau[0]*r_0+tau[1])) and (numIters<maxIters):
     Jacobian = diffjac(x,func_F,f_x)
     (LU, piv) = sp.lu_factor(Jacobian)
     for j in range(m):
         s = sp.lu_solve((LU,piv), -f_x)
         x = x + s
         f_x = func_F(x)
         err = np.linalg.norm(f_x)
         err_vec.append(err)
         numIters += 1

         if np.linalg.norm(f_x) <= (tau[0]*r_0 + tau[1]):
                break

 return x, numIters, err_vec
```

2.2 Results (A)

For all methods the ones vector, $x_0 = [1\ 1\ \dots\ 1]^T$, was used as the starting point; the Chandrasekhar H-Equations used $N = 200$ and $c = 0.9$; and the Shamanski method used $m = 2$.

The results for each method are shown in Table 2. The results show the number of iterations each method took to converge and the total computational time. Additionally, to prove that a root was correctly found, the solution vector x_{sol} was plugged back into the Chandrasekhar H-Equation and evaluated, then the norm on that value was taken and printed.

Table 2. Non-Linear Method Results

Method	Iterations	Run-Time	$\|\boldsymbol{F}(x_{sol})\|$
Newton's Method	3	2.142660 s	0.000002
Shamanskii Method	4	1.389598 s	0.000003
Chord Method	9	0.766461 s	0.000002
Fixed-Point	19	0.060836 s	0.000003

Figure 1 compares the run-time (blue) to number of iterations (red) for each of the four methods being analyzed.

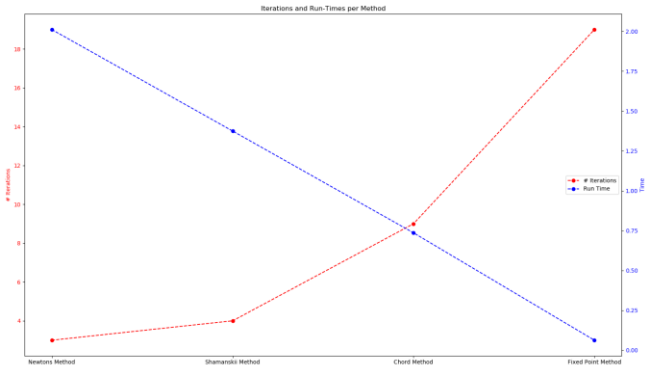

Figure 1. Run-Time and Number of Iterations for each Method

Also plotted from these calculations are the error-per iteration for each method. Figure 2 plots the error per iteration directly, while Figure 3 plots the log10 of the error.

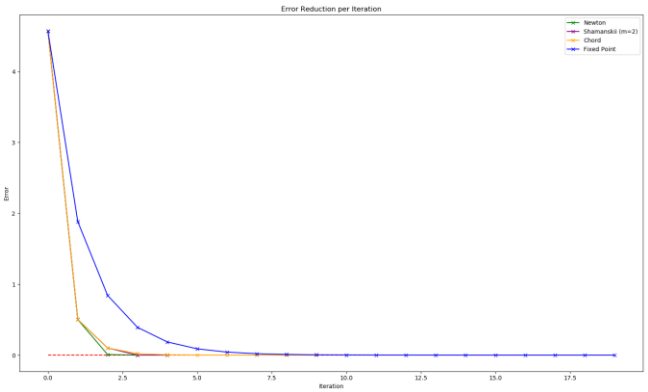

Figure 2. Error per Iteration for Each Method

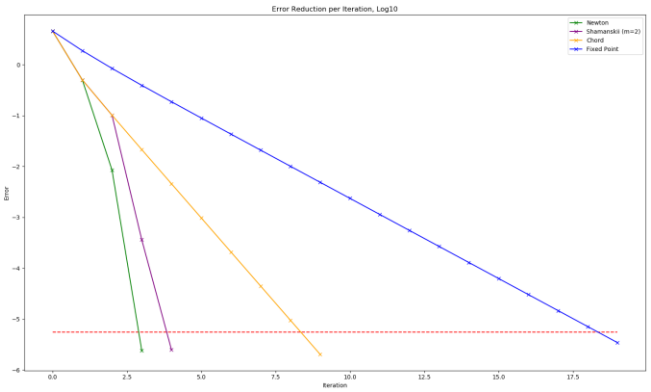

Figure 3. Log-10 of Error per Iteration for Each Method

2.3 Analysis (A)

2.3.1. Run-Time and Iterations per Method

The first thing to note is that all 4 methods found a correct root, since (in Table 2) the norm of the evaluated result, x_{sol}, was almost zero (recall that we just solved $F(x) = 0$).

The next notable result is the run-time and iteration count for each method (Figure 1). The run-time and iteration count behavior can be explained via the number of Jacobians being calculated. Recall from Table 1 (comparing the number of Jacobians calculated per iteration) that Newton's Method calculates a Jacobian every iteration – this is computationally expensive but results in rapid convergence because the Jacobian introduces an entire other dimension of information about the function; this is why the Newton's Method converges in 3 iterations but takes the most time by far. With $m = 2$, the Shamanski Method calculates the Jacobian every-other iteration, which is why it also converges in few iterations (albeit more than Newtons) but quite a bit faster than Newtons. The Chord method only calculates the Jacobian on the first iteration, which results in yet more iterations required to converge but still faster run-time. Finally, the Fixed-Point method never computes a Jacobian and runs quickly, but it lacks the extra dimension of information and as a result has by far the most number of iterations required to converge.

Of note in Figure 1 is that as we move from right to left (i.e. increasing the number of Jacobians to calculate) we seem to linearly increase the amount of calculation time, but decrease the number of iterations required at a higher-order than linearity. This indicates, that while Fixed-Point performed the fastest on the Chandrasekhar H-Equations with the parameters and starting value mentioned earlier, that Fixed-Point method could grow quadratically in the number of iterations required while only linearly decreasing the run-time. Thus, Fixed-Point is a viable method for some problems, it's certainly worth incurring the extra cost from calculating Jacobians in the Newton-based methods to increase the rate of convergence.

As for discriminating between the Newton-based methods, the rule of thumb is to use Chord if calculating the Jacobian and LU factorizing is expensive, and use Newton/Shamanskii if the Jacobian is cheap. In the particular case we have studied, it seems to make sense to use a method with fewer Jacobian calculations, such as Chord or Fixed-Point, because they ran significantly faster.

2.3.2. Error per Iteration for each Method

Figure 2 illustrates that all 4 methods reduce error incredibly fast, but unfortunately after just a couple of iterations the error is reduced so much, and the values so small, that it is hard to realize any information from the plot. Thus, the log10 of each error was taken and Figure 3 generated. Note that in both Figure 2 and Figure 3 the stopping condition is denoted by a red dashed line.

Figure 3 clearly illustrates that when no Jacobians are being calculated (i.e. Fixed-Point and Chord after the first iteration) the error reduces linearly, i.e. every iteration reduces the error by the same ratio. When we begin calculating Jacobians, like in Newton's Method, the error converges quadratically (that is, the amount of error reduction increases with every iteration). These results are as expected, because recall:

Fixed Point $$\left\|e^{(n+1)}\right\| \leq \gamma \left\|e^{(n)}\right\| \quad (4)$$

Newtons $$\left\|e^{(n+1)}\right\| \leq K \left\|e^{(n)}\right\|^2 \quad (5)$$

3. SECTION (B) – FINDING MULTIPLE ROOTS USING NEWTONS METHOD

3.1 Approach (B)

The goal for this section of study was to use Newton's Method on the system of equations in (6) to find as many roots as possible. Unfortunately, Newton's Method does not, by default, find more than one root in a system. This section will discuss the strategy used to work around this shortcoming in Newton's Method.

$$\begin{cases} xy = z^2 + 1 \\ xyz + y^2 = x^2 + 2 \\ e^x + z = e^y + 3 \end{cases} \quad (6)$$

3.1.1. Environment Setup

The system of equations described in (6) was rearranged to be of the form $F(x) = 0$, i.e.:

$$\begin{cases} 0 = z^2 + 1 - xz \\ 0 = xyz + y^2 - x^2 - 2 \\ 0 = e^x - e^y + z - 3 \end{cases} \quad (7)$$

This new system, (7), was implemented in Python as the function call, `function`, and because the degree of system is so small I manually derived the Jacobian and explicitly calculate each member of the matrix in the function, `jacobian`. Both of these functions are shown:

```
def function(x):
    n = x.shape[0]
    f = np.zeros((n,1))
    f[0,0] = x[2]**2 + 1 - x[0]*x[1]
    f[1,0] = x[0]*x[1]*x[2] + x[1]**2 - x[0]**2 - 2
    f[2,0] = np.exp(x[0]) - np.exp(x[1]) + x[2] - 3
    return f
```

```
def jacobian(x):
   n = x.shape[0]
   J = np.zeros((n,n))
   J[0, 0] = -x[1]
   J[0, 1] = -x[0]
   J[0, 2] = 2*x[2]
   J[1, 0] = x[1]*x[2]-2*x[0]
   J[1, 1] = x[0]*x[2]+2*x[1]
   J[1, 2] = x[0]*x[1]
   J[2, 0] = np.exp(x[0])
   J[2, 1] = -np.exp(x[1])
   J[2, 2] = 1
   return J
```

The main reasoning for explicitly calculating the Jacobian instead of re-using the difference Jacobian from Section A, which recall could take in "black box functions" and thus could've easily been used here, was because (1) this really is such a low order system and the Jacobian simple enough that calculating by hand was trivial and enabled (2) fairly significant savings in computational efficiency by not using the many stages of finite differencing in the `diffjac` function from Section A.

Since the Jacobian for this section is now a function call instead of the black-box `diffjac`, a slightly modified Newton's Method function was used that accepts a function pointer to (1) the non-linear system function itself (2) the Jacobian and (because this is purely Newton's Method and not combined with Chord and Shamannski) (3) function pointer to a solving method, e.g. LU, CG, etc. With the addition of (3) any solving method could be used, not just the built in LU factor + solve from the `nonlinearMethods` function in Section (A). This generalized Newton's Method is:

```
def newtonsMethod(fun_F,fun_Jacbn,fun_solv,x,tau,mxItrs):
  # Intialize
  f_x = fun_F(x)
  r_0 = np.linalg.norm(f_x)
  err = r_0
  err_vec = [err]
  numIters = 0

  while (err>=(tau[0]*r_0+tau[1]))and(numIters<mxItrs):
     Jacobian = fun_Jacbn(x)
     s, _ = fun_solv(A=Jacobian, b=-f_x)
     x = x + s
     f_x = fun_F(x)
     numIters += 1
     err = np.linalg.norm(f_x)
     err_vec.append(err)
     # if the error grew, break
     if (err_vec[numIters] - err_vec[numIters-1]) > 0:
         break

  return x, numIters, err_vec
```

The argument `fun_solv` for this version of Newton's Method allows for any system solver to be used. At first, my plan was to use LU factorization + LU solving to evaluate the inverse Jacobian; however, inaccurate "starting guesses" (starting guesses are discussed in Section 3.1.2) resulted in function evaluations that rapidly approached "infinity" or "NaN" and this caused the LU factorizor and solver in the SciPy library to crash. As a result, I settled on using just a matrix inverse to solve the Jacobian inverse system. I deemed the Jacobian inverse as acceptable because the dimension of the problem was so low, i.e. a matrix inverse with $O(N^3)$ with our dimension of $N = 3$ is still only 27 operations – a trivial computation.

3.1.2. Root Finding Strategy

Because Newton's method has no mechanism to find multiple roots, my strategy was to leverage the following theorem:

> *Theorem*
> If the following assumptions hold:
> 1. $F(x) = 0$ has a solution
> 2. $F' : \Omega \to \mathbb{R}^{nxm}$ is Lipschitz Continuous i.e. $\|F'(x) - F'(y)\| \leq \gamma\|x - y\|$
> 3. $F'(x^*)$ is non-singular
>
> Then there exists a $K > 0$ and a small neighborhood $B_{x^*}(\delta)$ such that if $x_0 \in B_{x^*}(\delta)$, then Newton's iterates converge quadratically.
>
> $$\left\|x^{(n+1)} - x^*\right\| \leq K\left\|x^{(n)} - x^*\right\|^2$$

That is, if I manage to pick a starting value x_0 within a radius δ of the true solution x^*, then Newton's method will converge quadratically.

To accomplish the task of picking roots within the ball of radius δ, my plan was to sample a volume so densely that I am guaranteed to pick a starting value within δ of every root in the system. In code, this amounts to defining a meshgrid and simply iterating over every point in the grid. Of course, this means that the limits of my grid must be picked such that all roots of the system lie within the outer limits, and a step size of the meshgrid must be fine enough so that roots cannot land between meshgrid steps. It is important to be aware that this methodology could lead to a grid so dense that it is computationally significant.

The meshgrid code:

```
stepSize = 0.5
vol = 10
xx = np.arange(-vol,vol,stepSize)
yy = np.arange(-vol,vol,stepSize)
zz = np.arange(-vol,vol,stepSize)
xxx, yyy, zzz = np.meshgrid(xx,yy,zz)
```

Deciding on reasonable limits and step size for the meshgrid is no trivial task, so to simplify I am making the assumption that my step size will sample the space so finely that I will not miss any roots. I am also making the assumption that I do not have any roots significantly beyond the volume defined by the meshgrid limits. To further justify this, beyond these limits I found that Newton's method tended to overflow during math operations quite quickly when starting guesses were sufficiently far from roots. Note that picking step size and mesh grid limits was a manual process of trial and error until I was satisfied that I was not missing any roots.

Finally, I exclude the root $[0\ 0\ 0]^T$ because by directly plugging in to our system of equations we can see that it is not a solution.

Pseudo-code for the entire root-finding procedure is shown (see the included zip file for complete source code):

```
Define dense meshgrid of startvals
for all startvals in meshgrid
    if startval is not zero
        x = newtonsMethod(startval)
        if F(x) = 0    //Check if actual root
            Check if already found root, if not,
             add it to list of roots
```

3.1.3. Original Root Finding Strategy

Originally my method was to use a much sparser meshgrid so as to alleviate the computational burden of performing Newton's Method over the dense set of points described in Section 3.1.2. This first attempt at a root finding algorithm was to define the sample space in a much courser manner and then use Fixed-Point iterations to condition the starting value. Unfortunately, instead of helping, the Fixed-Point method tended to move the starting guess farther away to the affect that Newton's Method was no longer able to find a solution anymore.

For example, when Fixed-Point was forced to start at $x_0 = [1\ 1\ 1]^T$, in just a few iterations it veered off towards infinity and could never self-correct. The exponentials in the third equation of (6) and (7) are responsible for this behavior.

Because the original intent of using Fixed-Point in my root-finding scheme was to turn a "bad" guess into an "okay" guess for Newton to solve, the fact that the iterates rapidly diverge from a solution when not already close to a root was a major problem that quickly invalidated using Fixed-Point in this manner. Thus, an incredibly finely sampled solution space and using Newton's Method directly was settled on for the root finding strategy.

3.2 Results (B)

To try and capture as wide a volume with as dense a sampling as possible and still maintain acceptable compute times, I ran several different tests with different meshgrid limits and step sizes, see Table 3.

Table 3. Root Finding Results

Meshgrid Limits (x,y,z)	Step Size	Total Samples	Number of Roots Found
+/- 5	0.1	1,000,000	2
+/- 10	0.2	1,000,000	2
+/- 20	0.5	512,000	2
+/- 2	0.04	1,000,000	2

These results lead me to believe that there are only 2 real roots in the system. They are:

$$\vec{x}_0 = [-6.00007775\ \ -1.82891628\ \ 3.15810866]^T$$

$$\vec{x}_1 = [1.77767576\ \ 1.42396038\ \ 1.23747321]^T$$

Note that, as part of my "root checking" algorithm I throw away any result from Newton's Method that

does not have an $F(x_{sol}) = 0$ within tolerance, using $\tau_r = \tau_a = 1e^{-4}$. Additionally, I consider a root as "already found" when the root falls within this same tolerance when compared to roots already in my root-list.

3.3 Analysis (B)

Based on the fact that my Newton-based root-finding algorithm did in fact find multiple roots, I can claim that it works. I did note that when I relaxed my tolerances for what constitutes a "root" or "already found" root, my algorithm finds many more potential roots.

It is also important to recognize that an enormous majority of starting values did not converge at all. This exemplifies the need for Newton's Method to be given a reasonable starting value, i.e. this illustrates the theorem presented in Section 3.1.2.

3.4 Future Work (B)

A potential upgrade to the algorithm would be to re-sample the volumes around all found roots more densely than the rest of the meshgrid. Because one root was found, it's very possible that other roots are "nearby" and thus the increased sampling density may help to find additional roots.

8. CONCLUSION

This paper successfully compared four non-linear system solving methods: Fixed-Point Method, Newton's Method, Chord Method, and Shamanskii Method. In all cases the run-time, number of iterations to converge, and error reduction per iteration were compared and explained in terms of the number of Jacobians calculated per iteration.

The second section of this paper successfully demonstrated how Newton's Method could be used within a root finding algorithm to locate multiple roots in a multivariate non-linear system.

Based on the analysis in this paper, the most important factors when evaluating and using non-linear system solving methods are (1) whether the Jacobian can be easily calculated and (2) deriving a good starting value for the solver. The complexity of the Jacobian will dictate whether Newton's or the Shamanskii Method can be used (as is the case when the Jacobian is cheap to calculate) or when the Jacobian is expensive this leads us to the Chord or Fixed-Point Method. In all cases, regardless of method, it is critically important to select a reasonable starting value otherwise convergence is not guaranteed, or even possible.

9. REFERENCES

[1] C.Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Philadelphia, PA, SIAM, 1995.