Applications of the Gauss-Newton Method

As will be shown in the following section, there are a plethora of applications for an iterative process for solving a non-linear least-squares approximation problem. It can be used as a method of locating a single point or, as it is most often used, as a way of determining how well a theoretical model fits a set of experimental data points. By solving the system of non-linear equations we obtain the best estimates of the unknown variables in a theoretical model. We are then able to plot this function along with the data points and see how well these data points fit the theoretical equation.

The first problem to be solved involved the finding of a single point. Say, for example, that there are a set of *m* beacons located at positions (p_k,q_k). Each of these beacons is at a given distance d_k from their origin. Given these positions and distances, we expect that the distance to the origin of each point is given by the following equation:

$$D(u,v)=\sqrt{((u-p_k)^2+(v-q_k)^2)}$$

Where *u* and *v* are the *x* and *y* coordinates of the origin. As mentioned before, we are not attempting to find the *exact* location of the origin, by the very nature of a least-squares problem this solution does not exist or else we wouldn't need to bother with the program at all, we are merely looking for the values *u* and *v* that minimize the sum of the squares of the residuals that is, the values that minimize the function:

$$S=\sum_{k=1}^{m} r_k^2$$

Where *r*, in this particular scenario, is given by the equation:

$$r_k=d_k-\sqrt{((u-p_k)^2+(v-q_k)^2)}$$

To test to see if the Gauss-Newton method will actually find the proper solution to this problem, we begin with a system to which we know the solution, in practice this would not be done but as a way to test the success of the method and the code this is appropriate. A vector *z* of 100 random points in the range zero to two pi was generated in MATLAB; our values for *p* and *q* were then given by *p=cos(z)* and *q=sin(z)*. This generated a circular set of points whose distances to the origin, known to be located at (0,0), was 1. Because the Gauss-Newton method requires the calculation of the Jacobian matrix of *r*, we first analytically determined the partial derivates of *r* with respect to both *u* and *v*:

$$\frac{(\partial r)}{(\partial u)}=(u-p_k)*\sqrt{((u-p_k)^2+(v-q_k)^2)}$$

$$\frac{(\partial r)}{(\partial v)}=(v-q_k)*\sqrt{((u-p_k)^2+(v-q_k)^2)}$$

In the program shown below, the Jacobian is calculated at each iteration using the newest approximations of *p* and *q* in the function df at the bottom of the code. Once the partial derivates of the Jacboian were determined, the Gauss-Newton method could be applied. Since we know that the actual origin is at (0,0) the initial guesses for *u* and *v* were chosen to be: *u=0.1* and *v=0.1* (in the program the values for *u* and *v* are stored in the column vector *a*).

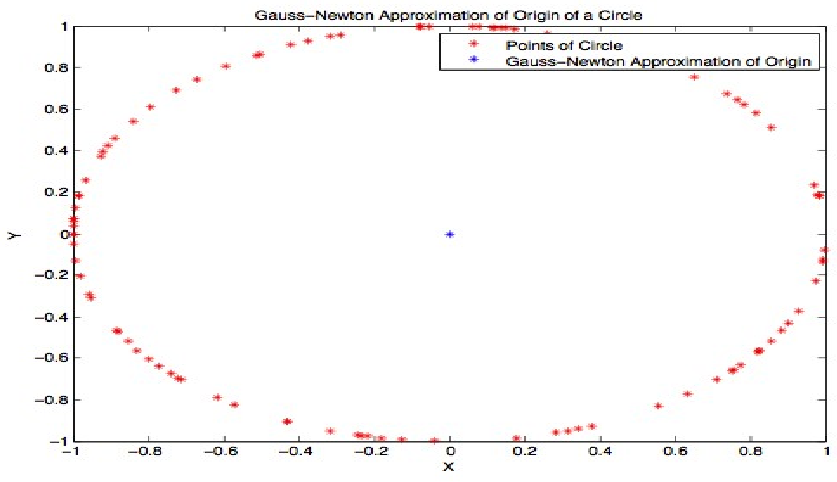
```
function [unknowns,steps,S] = GaussNewton()

%GaussNewton- uses the Gauss-Newton method to perform a non-linear least
%squares approximation for the origin of a circle of points
%  Takes as input a row vector of x and y values, and a column vector of
%  initial guesses. partial derivatives for the jacobian must entered below
%  in the df function
format long
tol = 1e-8;                   %set a value for the accuracy
maxstep = 30;                 %set maximum number of steps to run for
z = 2*pi*rand(100,1);              %create 100 random points
p = cos(z);                   %create a circle with those points
q = sin(z);
d = ones(100,1);              %set distance to origin as 1 for all points
a = [0.1;0.1];                %set initial guess for origin
m=length(p);                  %determine number of functions
n=length(a);                  %determine number of unkowns
aold = a;
for k=1:maxstep               %iterate through process
    S = 0;
    for i=1:m
        for j=1:n
            J(i,j) = df(p(i),q(i),a(1,1),a(2,1),j); %calculate Jacobian
            JT(j,i) = J(i,j);                        %and its trnaspose
        end
    end

    Jz = -JT*J;                                      %multiply Jacobian and
                                                     %negative transpose
    for i=1:m
        r(i,1) = d(i) - sqrt((a(1,1)-p(i))^2+(a(2,1)-q(i))^2);%calculate r
        S = S + r(i,1)^2;            %calculate sum of squares of residuals
    end
    S
    g = Jz\JT;                           %mulitply Jz inverse by J transpose
    a = aold-g*r;                        %calculate new approximation
    unknowns = a;
    err(k) = a(1,1)-aold(1,1);               %calculate error
    if (abs(err(k)) <= tol);                     %if less than tolerance break
        break
    end
    aold = a;                            %set aold to a
end
steps = k;
hold all
plot(p,q,'r*')                           %plot the data points
plot(a(1,1),a(2,1),'b*')                 %plot the approximation of the
origin(expect it to be 0,0)
title('Gauss-Newton Approximation of Origin of Circular Data Points') %set axis
lables, title and legend
xlabel('X')
ylabel('Y')
legend('Data Points','Gauss-Newton Approximation of Origin')
hold off
errratio(3:k) = err(2:k-1)./err(3:k);        %determine rate of convergence
end

function value = df(p,q,a1,a2,index)    %calculate partials
switch index
    case 1
        value = (2*a1 - 2*p)*0.5*((a1-p)^2+(a2-q)^2)^(-0.5);
    case 2
        value = (2*a2 - 2*q)*0.5*((a1-p)^2+(a2-q)^2)^(-0.5);
end
end
```

The program generates two things. First, a plot of the beacons (p_k,q_k) and the final approximation of the origin at point (u,v) and second, the values of u and v stored in the column vector *unknowns* as well as the number of steps the program ran before reaching the desired precision (*steps*) and the sum of the squares of the residuals at each iteration (stored in S).



The final values of *u* and *v* were returned as: *u=1.0e-16 *-0.318476095681976* and *v=1.0e-16 *0.722054651399752*, while the total number of steps run was 3. It should be noted that although both the exact values of *u* and *v* and the location of the points on the circle will not be the same each time the program is run, due to the fact that random points are generated, the program will still return values that are extremely close to the origin. By examining the plot shown and the final values of *u* and *v* it is clear to see that the Gauss-Newton method and its implementation in the above code are successful in solving the non-linear system of equations. Furthermore, because we are trying to minimize the sum of the squares of the residuals, we can look at these values at each iteration of the program to determine if we are getting a more accurate value for our unknowns. Shown below is a table with the number of steps and the value for *S* at each of these steps:

Iteration Number	S
1	1.013113985774467
2	3.137428721248484e-05
3	8.336998350484597e-15

This table shows that the Gauss-Newton method has drastically decreased the value *S* that we are attempting to minimize to a degree of precision that is near machine precision. It should be noted that not all values of *S* will reach this degree of accuracy. Because we are dealing with an example where the value of the unknowns exists, we can obtain a value for *S* that is extremely close to zero. Usually, as mentioned before, the problems dealt with will not have an obtainable solution, merely an approximation of the closes value to the solution.

Now that success of both the method and the implementation have been validated, it is possible to expand the previous example to an origin that is impossible to determine exactly. To do this, an almost identical version of the previous mentioned code was implemented. However, this time instead of generating points in a circle about a known origin, 100 entirely random points were generated within the range zero to one, with 100 randomly generated distances. In this case, it is impossible to determine the exact location to the origin; however, the Gauss-Newton may provide us with a way to approximate the best possible solution. The code to do this is shown below along with the plot generated and the final values returned for *u* and *v*.

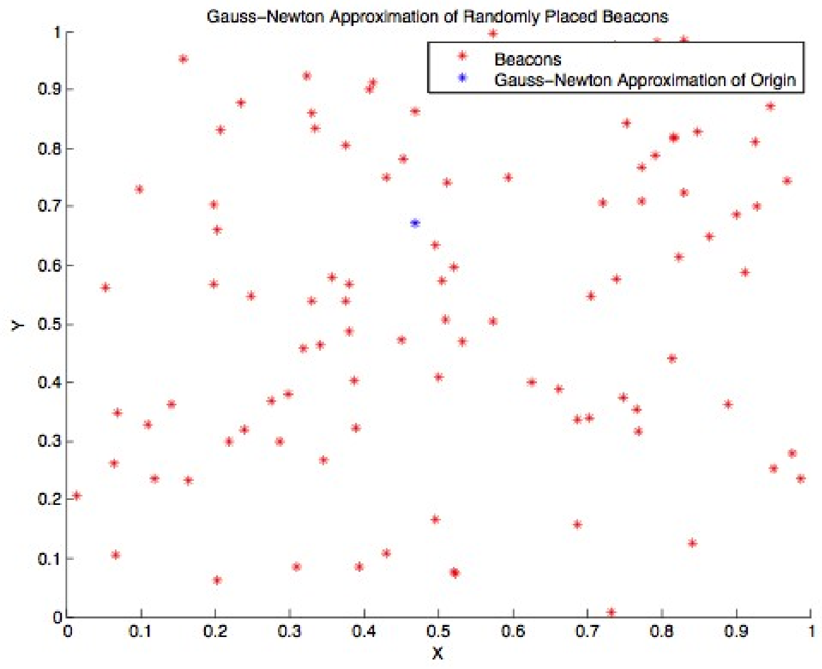
```
function [unknowns,steps,S] = GaussNewtonSquare()

%GaussNewton- uses the Gauss-Newton method to perform a non-linear least
%squares approximation for the origin of a circle of points
%  Takes as input a row vector of x and y values, and a column vector of
%  initial guesses. partial derivatives for the jacobian must entered below
%  in the df function
format long
tol = 1e-8;                     %set a value for the accuracy
maxstep = 50;                   %set maximum number of steps to run for
p = rand(100,1);              %generate random values for p and q between zero and 1
q = rand(100,1);
d = rand(100,1);              %generate random distances to origin
a = [0.5;0.5];                  %set initial guess for origin
m=length(p);                    %determine number of functions
n=length(a);                    %determine number of unkowns
aold = a;
for k=1:maxstep                 %iterate through process
    S = 0;
    for i=1:m
        for j=1:n
            J(i,j) = df(p(i),q(i),a(1,1),a(2,1),j); %calculate Jacobian
            JT(j,i) = J(i,j);                       %and its trnaspose
        end
    end

    Jz = -JT*J;                                     %multiply Jacobian and
                                                    %negative transpose
    for i=1:m
        r(i,1) = d(i) - sqrt((a(1,1)-p(i))^2+(a(2,1)-q(i))^2);%calculate r
        S = S + r(i,1)^2;           %calculate sum of squares of residuals
    end
    S
    g = Jz\JT;                      %mulitply Jz inverse by J transpose
    a = aold-g*r;                   %calculate new approximation
    unknowns = a;
    err(k) = a(1,1)-aold(1,1);          %calculate error
    if (abs(err(k)) <= tol);                %if less than tolerance break
        break
    end
    aold = a;                       %set aold to a
end
steps = k;
hold all
plot(p,q,'r*')                      %plot the data points
plot(a(1,1),a(2,1),'b*')            %plot the approximation of the
origin(expect it to be 0,0)
title('Gauss-Newton Approximation of Randomly Placed Beacons') %set axis lables,
title and legend
xlabel('X')
ylabel('Y')
```

```
legend('Beacons','Gauss-Newton Approximation of Origin')
hold off
erratio(3:k) = err(2:k-1)./err(3:k);       %determine rate of convergence
end

function value = df(p,q,a1,a2,index)    %calculate partials
switch index
    case 1
        value = (2*a1 - 2*p)*0.5*((a1-p)^2+(a2-q)^2)^(-0.5);
    case 2
        value = (2*a2 - 2*q)*0.5*((a1-p)^2+(a2-q)^2)^(-0.5);
end
end
```



The final values of *u* and *v* are: *u=0.468790145959325* and *v=0.673144815981593*. Because these points were plotted randomly using the MATLAB pseudo-random number generator there is bound to be a non-uniform distribution of points throughout the area. Because of this, the solution given by the Gauss-Newton method will tend to be closest to the area with the higher concentration of points. This fact illustrates one of the very useful attributes of non-linear least-squares approximations. That is that they account for and, to some degree, avoid statistical outliers, meaning that outliers in the data will not have a large affect on the data like they would if simple interpolation is used. As mentioned before, not all values of *S* will be as small as in the first example. Because the program ran for 44 steps, it would be superfluous to show the values of *S* at each iteration, below are a table of the first, and last two iterations.

Iteration Number	S
1	12.932504392657616
43	11.822767836017297
44	11.822767836017240

The desired tolerance has been met, but the value of *S* has not been decreased by very much from the initial approximation. This is because the data points have been distributed randomly and there is no perfect solution that will fit the theoretical model given. However, given the constraints and the theoretical function, the Gauss-Newton method has given an approximation that is as close as is desired by the specified tolerance.

In the above two examples we are using the Gauss-Newton method to find two variables that correspond to a single point given from a theoretical equation. Another common application of non-linear least-squares techniques is in data fitting. In this scenario, a system of equations is arranged using experimental data points, and smooth curve given by a theoretical equation is attempted to be fitted to the data. The unknowns of the function are treated in the exact same manner as in the previous examples and are found using the same technique. As an example of this application of the Gauss-Newton method, we will examine the basic exponential formula for population growth.

The equation for the theoretical population growth model is given below:

$$P(t)=P_0 * e^{(r*t)}$$

where P_0 is the initial population and *r* is the rate of growth. Here, we will be using the Gauss-Newton method to attempt to fit data collected by the census bureau on the population of New York City from 1810 until 1930 (shown below) to this theoretical model.

Population of NYC 1810 – 1930

Year	Population
1810	119,734
1820	152,056
1830	242,278
1840	391,114
1850	696,115
1860	1,174,779
1870	1,478,103
1880	1,911,698
1890	2,507,414
1900	3,437,202
1910	4,766,883
1920	5,620,048
1930	6,930,446

We expect the experimental data to closely follow the theoretical equation for population growth given above. In order to do this, using the Gauss-Newton method, we must first calculate the partial derivatives for the Jacobian as was done in the previous example. Here, *r* is given by the equation:

$$r_k = q_k - a_1 * e^{(a_2 * p_k)}$$

thus our equations for the partial derivatives for the Jacobian are given by:

$$\frac{(\partial r)}{(\partial a_1)} = e^{(a_2 * p_k)}$$

$$\frac{(\partial r)}{(\partial a_2)} = p_k * a_1 * e^{(a_2 * p_k)}$$

where p_k is the year, q_k is the measured population at that year, and a_1 and a_2 are P_0 and *r* respectively. Once the partial derivatives for the Jacobian have been calculated, we can proceed with the Gauss-Newton method; implemented in the code below.

```
function [unknowns,steps,S] = GaussNewtonPopulation()

%GaussNewtonPopulation - Uses the Gauss-Newton Method to find the initial
%population(p0) and growth rate(r) of the Malthusian Model for the population
%of New York City from 1810 until 1930 (data obtained from the Census bureau

format long
tol = 1e-8;                   %set a value for the accuracy
```

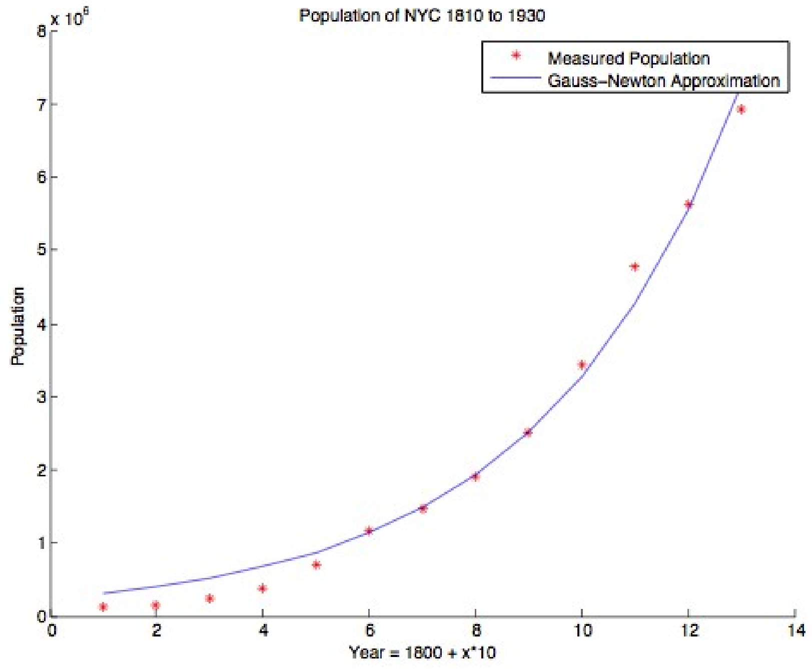
```
maxstep = 30;               %set maximum number of steps to run for
%for convenience p is set as 1-13
p = [1,2,3,4,5,6,7,8,9,10,11,12,13];
%set q as population of NYC from 1810 to 1930
q=[119734,152056,242278,391114,696115,1174779,1478103,1911698,2507414,3437202,47668
83,5620048,6930446];
a = [110000;0.5];           %set initial guess for P0 and r
m=length(p);                %determine number of functions
n=length(a);                %determine number of unkowns
aold = a;
for k=1:maxstep             %iterate through process
    S = 0;
    for i=1:m
        for j=1:n
            J(i,j) = df(p(i),q(i),a(1,1),a(2,1),j); %calculate Jacobian
            JT(j,i) = J(i,j);                       %and its trnaspose
        end
    end

    Jz = -JT*J;                                 %multiply Jacobian and
                                                %negative transpose
    for i=1:m
        r(i,1) = q(i) - a(1,1)*exp(a(2,1)*p(i));%calculate r
        S = S + r(i,1)^2;           %calculate the sum of the squares of the
residuals
    end
    S
    g = Jz\JT;                              %mulitply Jz inverse by J transpose
    a = aold-g*r;                           %calculate new approximation
    unknowns = a;                            %set w equal to most recent approximations
of the unkowns
    abs(a(1,1)-aold(1,1));                  %calculate error
    if (abs(a(1,1)-aold(1,1)) <= tol);      %if less than tolerance break
        break
    end
    aold = a;                               %set aold to a
end
steps = k;
f = unknowns(1,1)*exp(unknowns(2,1)*p);                 %determine the malthusian
approximation using P0 and r determined by Gauss-Newton method
hold all
plot(p,q,'r*')                              %plot the measured population
plot(p,f)                                   %plot the approximation
title('Population of NYC 1810 to 1930') %set axis lables, title and legend
xlabel('Year = 1800 + x*10')
ylabel('Population')
legend('Measured Population','Gauss-Newton Approximation')
end

function value = df(p,q,a1,a2,index)    %calculate partial derivatives
switch index
    case 1
        value = exp(a2*p);
    case 2
        value = p*a1*exp(a2*p);
end
end
```

The code above is nearly identical to the one used in the previous example. All that has been changed is the values used for *p* and *q,* now representing the year (which for simplicity has been made

1-13) and the population at that year, the partial derivatives in the function df, and the equation for *r*. Our initial guesses, of course, must change as well and they have now been chosen to be 110,000 for the initial population and 0.5 for the growth factor. With these initial guesses the following plot was generated:



The final answers for the unknown variables , determined to within the proper precision and returned in the column vector *unknowns* where the first entry is P_0 and the second is *r*, are: $P_0 = 233044.9540808864$ and $r = 0.2643895092$. Below is a table of values at each iteration for the sum of the squares of the residuals (*S*).

Iteration Number	S
1	6.623778985888237e+15
2	5.769851917836982e+14
3	3.380200120826823e+13
4	2.985624164800214e+12
5	8.467364836905938e+11
6	6.542511966853999e+11

7	6.540381256878523e+11
8	6.540364080058896e+11
9	6.540363892243904e+11
10	6.540363890124553e+11
11	6.540363890100629e+11
12	6.540363890100376e+11
13	6.540363890100355e+11
14	6.540363890100377e+11
15	6.540363890100369e+11
16	6.540363890100361e+11
17	6.540363890100372e+11
18	6.540363890100359e+11

The above table shows a final value of S that is quite large despite the facts that the plot of the function using the final determined values of P_0 and r appears relatively accurate and the desired tolerance has been met. This is because we are dealing with experimental data with values that are very large. Therefore differences that appear smaller in the graph are actually fairly big. This will, of course, result in residuals that are large and sums of squares of residuals that are even larger still. What we can look at, however, is the fact that the Gauss-Newton algorithm has the reduced the value of S by several orders of magnitude from our initial guesses for the values of the unknowns. This tells that, although the value of S is still large, the fit to this data is as good as it can be within the constraints of the theoretical model and the desired level of precision.

As a final example of the application of the Gauss-Newton method, we attempted to find the best fit for a set of data with a sinusoidal function. This example illustrates how the Gauss-Newton method can applied to functions with more that just two variables, and that it can be applied to an equation of any form. In it, will attempt to model temperature data with a sinusoidal function. Below is a chart of the average high temperatures per month of the city of Monroe Louisiana (courtesy of *weather.com*).

Month	Temperature (Fahrenheit)
January	56
February	60
March	69
April	77
May	84
June	90
July	94
August	94
September	88
October	79
November	68
December	58

As can be seen by the chart, this data appears to take the form of a sinusoidal function. Therefore, we will try to fit it with the general form of this function, shown below:

$$y(t)=A*\sin(w*t+phi)+C$$

A is the amplitude, *w* is the frequency, *t* is time, *phi* is the phase and *C* is a constant. Given this equation there are four unknowns that must be found. In the code, these four variables are represented by the column vector *a* and are returned in the column vector *w*. As always, the code requires that we first solve, analytically, for the partial derivatives required for the Jacobian. In this example, the function to be minimized, *r*, is given by:

$$r_k=q_k-(a_1*\sin(a_2*p_k+a_3)+a_4)$$

where r_k is the residual at the particular temperature value, q_k is the measured temperature and p_k is the month, represented in the code as January = 1, February = 2, etc. As mentioned before, the values a_l through a_k are the four unknowns in the equation for the sinusoid given above. Since the Gauss-Newton method calls the Jacobian of *r*, the four partial derivatives with respect to the four unknowns must be calculated and entered into the function *df* in the MATLAB code. The four partial derivatives are shown below:

$$\frac{(\partial r)}{(\partial a_1)} = \sin(a_2 * p_k + a_3)$$

$$\frac{(\partial r)}{(\partial a_2)} = p_k * a_1 * \cos(a_2 * p_k + a_3)$$

$$\frac{(\partial r)}{(\partial a_3)} = a_1 * \cos(a_2 + a_3)$$

$$\frac{(\partial r)}{(\partial a_4)} = 1$$

Using these four partial derivatives, the Jacobian of *r* and its transpose can be calculated, thereby allowing us apply the Gauss-Newton method. The MATLAB code used to perform this is included below:

```
function [unknowns,steps,S] = GaussNewtonTemps()

%GaussNewton- uses the Gauss-Newton method to perform a non-linear least
%squares approximation for he average temperatures (by month) of Monroe
%Louisiana
%  Takes as input a row vector of x and y values, and a column vector of
%  initial guesses. partial derivatives for the jacobian must entered below
%  in the df function
format long
tol = 1e-5;                     %set a value for the accuracy
a = [15;0.4;10;1];           %initial approximations for unknowns
maxstep = 30;                   %set maximum number of steps to run for
q = [56,60,69,77,84,90,94,94,88,79,68,58]; %average highs January through february
p = [1,2,3,4,5,6,7,8,9,10,11,12];%months jan=1,february = 2,..., december = 12
m=length(p);                    %determine number of functions
n=length(a);                    %determine number of unkowns
aold = a;                       %set aold equal to initial guess
for k=1:maxstep                 %iterate through process
    S = 0;
    for i=1:m
        for j=1:n
            J(i,j) = df(p(i),a(1,1),a(2,1),a(3,1),a(4,1),j); %calculate Jacobian
            JT(j,i) = J(i,j);                            %and its trnaspose
        end
    end
    Jz = -JT*J;                                       %multiply Jacobian and
                                                     %negative transpose
    for i=1:m
        r(i,1) = q(i) - a(1,1)*sin(a(2,1)*p(i) + a(3,1)) - a(4,1);%calculate r
       S = residual + r(i,1)^2      %calculate sum of squares of residuals
    end
    g = Jz\JT;                               %mulitply Jz inverse by J transpose
    a = aold-g*r;                            %calculate new approximation
    unknowns = a;
    S
    err(k) = a(1,1)-aold(1,1);               %calculate error
    if (abs(err(k)) <= tol);                 %if less than tolerance break
        break
    end
    aold = a;                                %set aold to a
end
steps = k;                                   %set steps ran for equal to k
for i=1:m
    f(i) = a(1,1)*sin(a(2,1)*p(i) + a(3,1)) + a(4,1);%calculate the value of the
function
end
hold all
plot(p,q,'r*')                       %plot the data points
plot(p,f)                            %plot the approximation of the function
title('Gauss-Newton Approximation of Average High Temperatures Monroe LA') %set
axis lables, title and legend
xlabel('Month (1=January, 12=December)')
ylabel('Average High Temperatures')
legend('Data Points','Gauss-Newton Approximation')
hold off
end

function value = df(p,a1,a2,a3,a4,index)  %calculate partials
switch index
    case 1
        value = sin(a2*p + a3);
    case 2
        value = p*a1*cos(a2*p + a3);
    case 3
        value = a1*cos(a2*p + a3);
    case 4
        value = 1;
end
end
```

As in the previous examples, for each iteration, the Jacobian using the newest approximations of the unknowns is calculated, along with its transpose, *r* (the residual) is calculated and the Gauss-Newton algorithm is implemented. The program returns the values of the unknowns (returned in *unknowns*), the number of steps that were run (*steps*), and the sum of the squares of the residuals (returned in *residual*). After 11 steps, using initial guesses $a_1 = 15$, $a_2 = 0.4$, $a_3 = 10$, and $a_4 = 1$, the program reached the desired precision and returned that the unknown variables were:

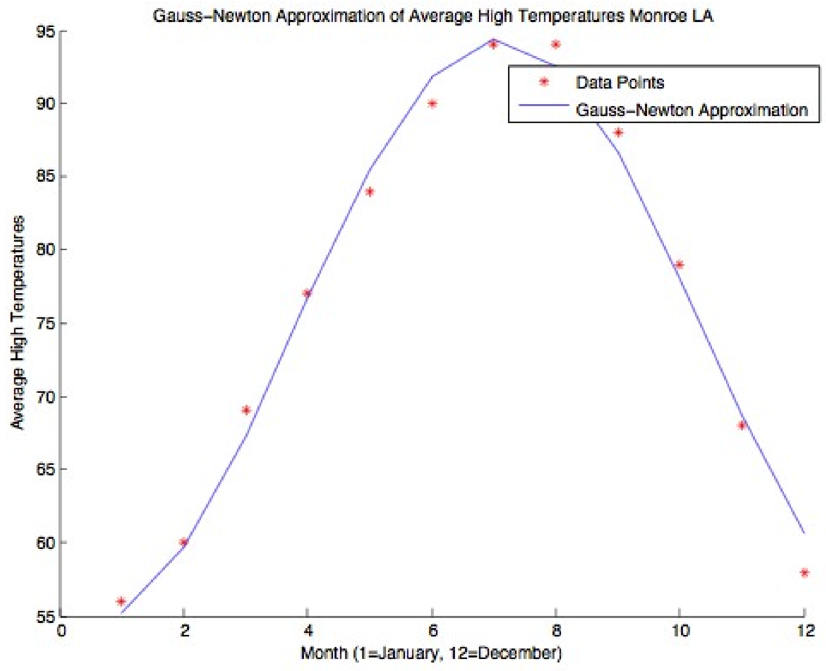
$$a_1 = A = 19.898334641005835$$

$$a_2 = w = 0.475806006054592$$

$$a_3 = phi = 10.771250102297055$$

$$a_4 = C = 74.500317750078267$$

Now that the four unknowns in the function have been determined, it possible to plot the theoretical model along the measured experimental data points to see if the formulas is actually a good fit for the data. The graphs below shows this plot of the theoretical model and the collected data points:



In the plot shown above, we can see that the sinusoidal data is actually a good approximation of the collected data. Furthermore, a good way to examine if the fit is truly good is to look at the sum of the squares of the residuals (*S*) at each iteration of the program. Running this program displays the value of this sum at each iteration, a table of these values is shown below, starting with the initial approximations and ending with the value calculated using the final approximation of the unknowns:

Iteration Number	S
1	7.064899174626484e+04
2	1.523781598418015e+03
3	1.419187165814137e+03

4	1.792616558543499e+02
5	22.317696985314178
6	21.705865558143735
7	21.691210636797358
8	21.690924130056597
9	21.690917403577657
10	21.690917244593553
11	21.690917240837535

As can be seen in the table, the sum of the squares of the residuals has been dramatically reduced and, in addition, by examining the last two entries in the table we see that the desired tolerance has in fact been met. The drastic reduction in the value of *S*, along with its low final value, indicate that the theoretical model used is a fairly accurate approximation of the function describing the average temperature.

Rate of Convergence

A good way to examine the rate of convergence of the Gauss-Newton method is to look at the decrease in the value of *S* at each iteration of the function. By calculating the ratio of the differences at adjacent steps we can get an approximation the rate of decrease, and therefore the rate of convergence, of the Gauss-Newton method. Below is a table of these differences and ratio for the final example discussed in the applications section.

Iteration	S	Differences	Ratio
1	7.064899174626484e+04		
2	1.523781598418015e+03	-6.912521014784682e+04	
3	1.419187165814137e+03	-1.045944326038780e+02	6.608880456346957e+02
4	1.792616558543499e+02	-1.239925509959787e+03	0.084355416324381
5	22.317696985314178	-1.569439588690357e+02	7.900434772353754
6	21.705865558143735	-0.611831427170443	1.070930066150341e+04
7	21.691210636797358	-0.014654921346377	41.749212616668217

8	21.690924130056597	-2.865067407604727e-04	51.150354464535631
9	21.690917403577657	-6.726478940066727e-06	42.593865722804239
10	21.690917244593553	-1.589841041038653e-07	42.309128814993208
11	21.690917240837535	-3.756017719069860e-09	42.327836553240786

As can be seen by the above table, the Gauss-Newton method offers a very fast rate of convergence. However, convergence is not always garunteed. If the initial guesses provided to the algorithm are far away from the exact answer, then the algorithm will not converge at all, not even to a local minima or maxima.

Conclusion

Non-linear least-squares approximations are an incredibly useful tool for analyzing sets of data and determining whether or not a theoretical model is a good fit for data obtained experimentally. Although it involves the solving of a system of non-linear equations, it provides us with a powerful, rapidly converging tool that provides answers to a system of equations that does not have an absolute solution. We have shown, in the first example, that given a system that does have a known solution, the Gauss-Newton method will provide the correct answer. When given a system without a solution, like the three other examples, the Gauss-Newton method attempts to find an approximation to the solution that is as close as possible. There are drawbacks to the Gauss-Newton method, such as the fact that it will not converge if the initial guesses are not in a suitable range, however it is nonetheless a powerful computational tool that provides answers where analytic methods fail.