Scientific Computing HW4

Runze Fang

November 2, 2020

1 Newton-Raphson Method in One Dimension

1.1 The roots

I use *ezplot* to plot this function on the interval [1.4,1.7]. The figure is shown below.

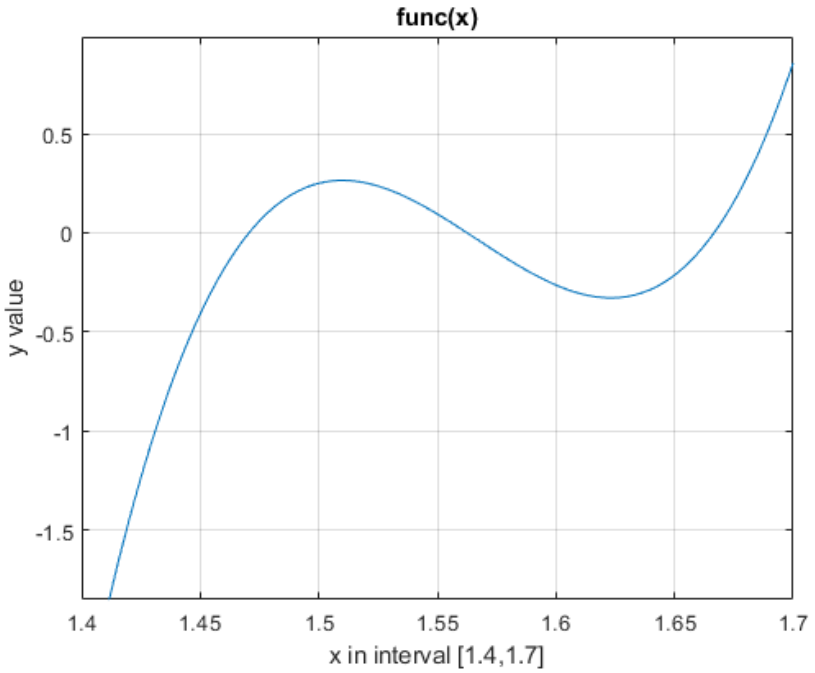

Figure 1: plot of polynomial

Then I compute the zeros using $fzero$ method. Since $fzero$ need the initial guess to compute the zeros near the guess. By observing the function figure, I found that there are 3 zeros which around 1.45,1.55 and 1.65 respectively. Thus, I got my initial guess set [1.45, 1.55, 1.65]. I pass them to $fzero$ and get 3 zeros as shown below.

```
1th zeros of this poly using fzeros is 1.47059
2th zeros of this poly using fzeros is 1.5625
3th zeros of this poly using fzeros is 1.66667
```

Figure 2: zeros computed by fzeros

1.2 Newton's Methos

I implement Newton's method by computing first derivative and second-order derivative by hand then implementing them in computing with Newton method. I generate 10 guesses using *linspace* as the starting point and see whether it willl converge. The convergence value is the roots of $f(x)$. The result is shown below.

```
initial guess is 1.4
after 7 rounds of computation, the newton method converges at 1.47059
initial guess is 1.43333
after 6 rounds of computation, the newton method converges at 1.47059
initial guess is 1.46667
after 4 rounds of computation, the newton method converges at 1.47059
initial guess is 1.5
after 7 rounds of computation, the newton method converges at 1.47059
initial guess is 1.53333
after 5 rounds of computation, the newton method converges at 1.5625
initial guess is 1.56667
after 3 rounds of computation, the newton method converges at 1.5625
initial guess is 1.6
after 5 rounds of computation, the newton method converges at 1.5625
initial guess is 1.63333
after 8 rounds of computation, the newton method converges at 1.66667
initial guess is 1.66667
after 1 rounds of computation, the newton method converges at 1.66667
initial guess is 1.7
after 6 rounds of computation, the newton method converges at 1.66667
```

Figure 3: implement of Newton's Method

Now I try to verify the quadratic order of convergence. I choose 1.65 as my starting point and iterate Newton method for 4 times. I set k=4 because the Newton method converges so rapid, and when $k \geq 5$, the error is dominated by truncation errors instead of roundoff errors. I calculate the raletive error between $\lim_{k\to\infty} \frac{|e^{k+1}|}{|e^k|^2}$ and $C = \left|\frac{f''(\alpha)}{2f'(\alpha)}\right|^{-1}$ when k is from 1 to 4.

```
verify with the initial guess 1.65, the relative error between
limit of |e_k+1|/|e_k|^2 and C when k is 1 to 4



when k = 1, the relative error is 0.613979
when k = 2, the relative error is 0.130183
when k = 3, the relative error is 0.0124692
when k = 4, the relative error is 1.47804e-05
```

Figure 4: verification of quadratic convergence

As shown in Figure4, with the growth of k, the relative error is getting smaller. This indicates the limit is getting close to C as the grwoth of k, which conforms the quadratic convergence.

1.3 Robustness

I generate 100 guesses in the interval [1.4,1.7] and run Newton Method for 100 times. I plot the value to which it converges. To judging whether it converges, I use increment test method as the termination option. I set $\epsilon = 1e - 10$ and when $|x^{k+1} - x^k| < \epsilon$, I terminate the computation.

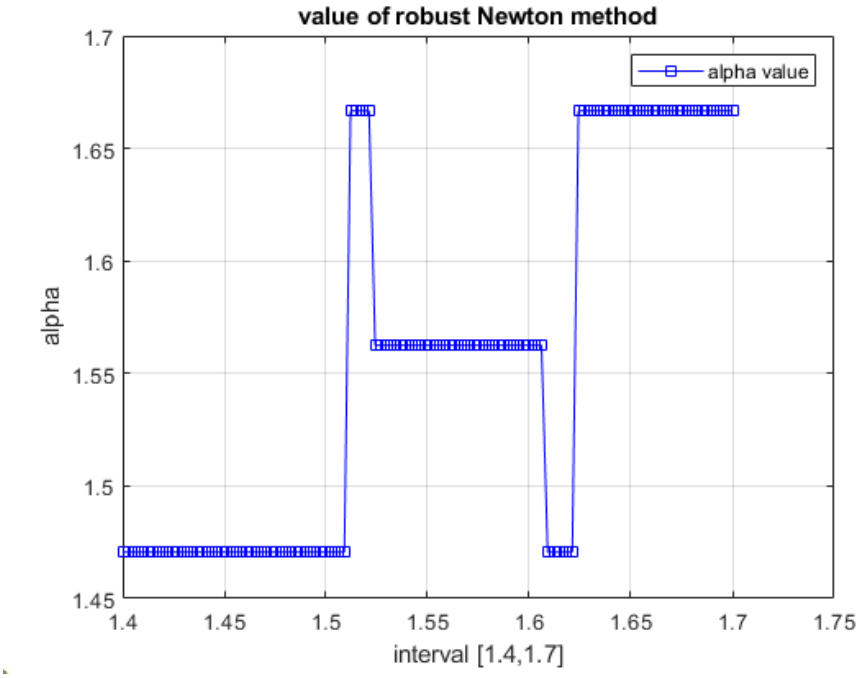

Figure 5: plot of 100 guesses of Newton's method

As shown in the graph, there are three basins that the method will converge to. The two shorter interval indicates that the initial guess in thest two interval is not sufficiently close to one of the roots and cannot get the root value. The basin of the middle root is [1.52, 1.61], which has a width of 0.09. Then I compute the width of basin of each root by the formula $|x^0 - \alpha| \leq \left|\frac{f''(\alpha)}{2f'(\alpha)}\right|^{-1}$.

```
basin of 1.47059 is 0.0625782
basin of 1.5625 is 0.78125
basin of 1.66667 is 0.0680272
```

Figure 6: width of basin

I found that the estimate is particularly bad for the middle root 25/16. To

figure out why, I plot the graph of the function's first derivative and second-order derivative.

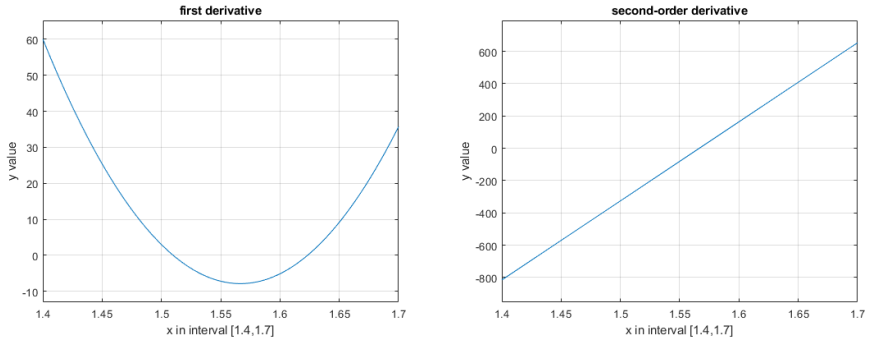

Figure 7: first and second-order derivative

As shown in Figure 7, when $x = 25/16$, the value of second-order derivative is close to 0. This may be the reason of the abnormal esimate: when the second-order derivative is close to 0 and it is the denominator of $\left|\frac{f''(\alpha)}{2f'(\alpha)}\right|^{-1}$, and the whole formula is close to infinity.

2 Non-liear Least-Squares Fitting

2.1 Synthetic Data

I generate $m = 100$ points randomly and uniformly distributed using *rand* and compute the actual function. Then I generated some pertubation by *randn*. I plot the actual function and synthetic function in the same plot.

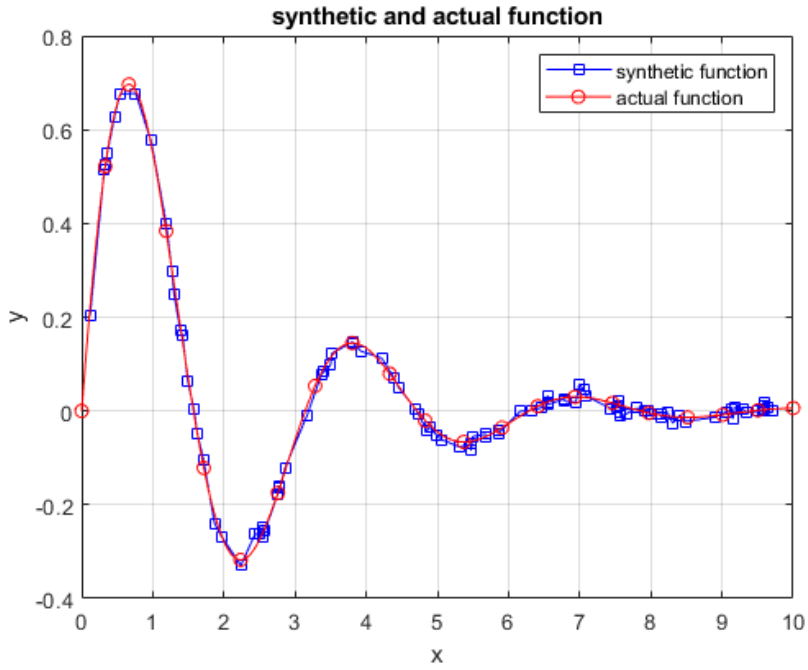

Figure 8: plot of acutal function and synthetic function

The curve of two functions are bacically the same. They have slightly difference in some points and if we zoom up Figure 8 a little bit, we can find some differences. The figure below shows two differences.

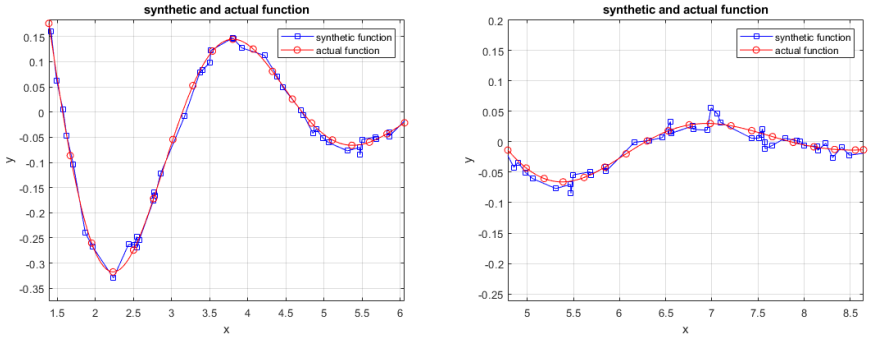

Figure 9: zoom of Figure 8

2.2 Gauss-Newton Methos

I try to linearize the function $f(x_i; c)$ around the current extimate c_k, using Jacobian matrix. I just use back slash to do the approximation transform. The correct value of c is [1,0.5,2,0]'. I implement the Gauss-Newton's algorism and set the initial guess c_0 as [1.1,0.4,2.1,0.1]'. I also set the termination condition as $||\Delta c_k|| \leq 1e - 10$.

```
when the guess c is [1.1,0.4,2.1,0.1]', the estimate c is
    1.0050
    0.5062
    2.0032
   -0.0034
```

Figure 10: guess c0 = [1.1,0.4,2.1,0.1]'

To determine whether the convergence is linear of quadratic, I compute $\lim_{k\to\infty} \frac{|e^{k+1}|}{|e^k|}$ and $\lim_{k\to\infty} \frac{|e^{k+1}|}{|e^k|^2}$ for several different guesses of c_0.

```
when the c0 is [1.1,0.4,1.7,0.1]', the estimate c is
    0.9857
    0.4922
    2.0002
   -0.0013
the linear convergence test is
    0.5219    0.3097    0.3493    0.8354    0.9993    1.0000    1.0000    1.0000         0
the quadratic convergence test is
    1.5065    1.7133    6.2384   42.7111   61.1538   61.2389   61.2394   61.2394         0
```

Figure 11: guess c0 = [1.1,0.4,1.7,0.1]'

```
when the c0 is [1.4,0.2,2.3,0]', the estimate c is
    1.0068
    0.5017
    1.9981
    0.0067
the linear convergence test is
0.7541    0.6999    0.3100    0.1184    0.8637    1.0158    1.0001    1.0000    1.0000    1.0000
the quadratic convergence test is
1.2933    1.5918    1.0073    1.2404   76.4767  104.1412  100.9474  100.9248  100.9242  100.9242
```

Figure 12: guess c0 = [1.4,0.2,2.3,0]'

```
when the c0 is [1.1,0.4,2.1,0.1]', the estimate c is
    0.9959
    0.4976
    2.0030
   -0.0057
the linear convergence test is
    0.3201    0.1838    0.6835    0.9881    1.0001    1.0000    1.0000    1.0000         0
the quadratic convergence test is
    1.6007    2.8705   58.0834  122.8428  125.8331  125.8041  125.8044  125.8044         0
```

Figure 13: guess c0 = [1.1,0.4,2.1,0.1]'

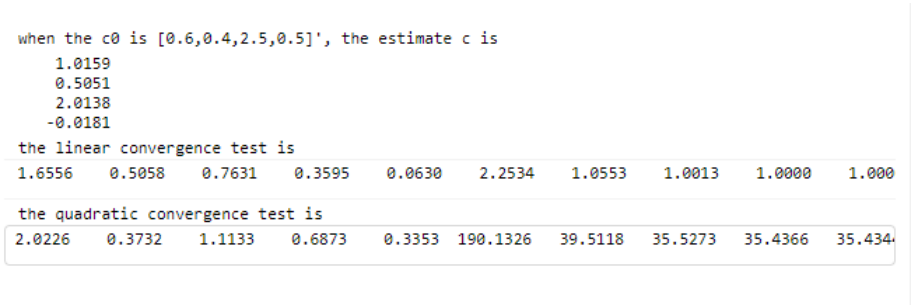

Figure 14: guess c0 = [0.6,0.4,2.5,0.5]'

We can see from the graph above that although the value will converge if we apply the quadratic convergence rules, the convergence value is different every time. If we apply the linear convergence rules, the convergence value will always be 1. Therefore, the convergence is linear.

Next, I set c_0 as [1,1,1,1], and run the algorithm again.

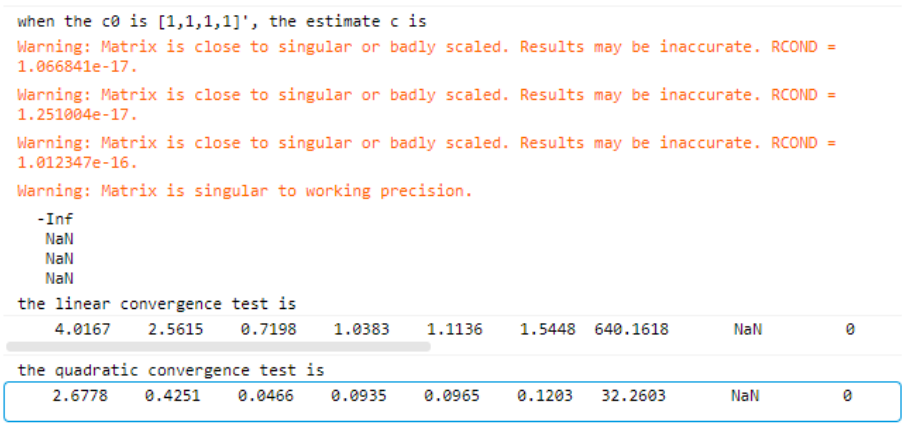

Figure 15: guess c0 = [1,1,1,1]'

As shown in Figure 15, when $c_0 = [1, 1, 1, 1]$, the method will not converge to the correct answer. Then I do two more test.

```
when the c0 is [0,2,0,1]', the estimate c is
Warning: Matrix is singular to working precision.
    0.6214
       NaN
       NaN
       NaN
the linear convergence test is
   NaN     0     0     0     0     0     0     0     0     0     0     0     0     0     0
the quadratic convergence test is
   NaN     0     0     0     0     0     0     0     0     0     0     0     0     0     0
```

Figure 16: guess c0 = [0,2,0,1]'

```
when the c0 is [1.5,1.6,0.3,1.2]', the estimate c is
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
2.454113e-27.
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
2.999159e-37.
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
3.260294e-37.
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
2.810014e-36.
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
1.646974e-33.
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
2.949911e-40.
Warning: Matrix is singular to working precision.
   NaN
   NaN
   NaN
   NaN
the linear convergence test is
   1.0e+95 *
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    4.9489       NaN
the quadratic convergence test is
   1.0e+91 *
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    5.0464       NaN
```

Figure 17: guess c0 = [1.5,1.6,0.3,1.2]'

By observing Figure 10 to Figure 17, we can conclude that for different initial guesses, the method may not converge;it converges most of time if the initial guess is not too far away from correct c and if it converges, it will converge to the correct c.

Next, I test without the pertubation.

```
when the c0 is [1.1,0.4,2.1,0.1]', the estimate c is
    1.0000
    0.5000
    2.0000
   -0.0000
the relative error is 1.83102e-17
   1.8310e-17
steps you need is 7
```

Figure 18: guess c0 = [1.1,0.4,2.1,0.1]'

As shown in Figure 18, we use 7 steps to achieve 17 digits accuracy.

I believe that Guass-Newton Algorithm has some connections with Newton's Method. They share the same methodology of solving the non-linear problem: linearize the non-linear problem and then apply a series of guess value to approximate the correct value. By checking Wiki page of Guass-Newton Algorithm, I read that the Guass-Newton Algorithm is actually a modification of Newton's method.

2.3 Levenberg-Marquardt Algorithm

I delete the pertubation and implement Levenberg-Marquardt Algorithm. I set $\lambda_1 = 10$, and try $c_0 = [1, 1, 1, 1]$, which failed using Guass-Newton Algorithm.

```
when the c0 is [1,1,1,1]' and lambda is 10 , the estimate c is
    0.9961
    0.4931
    2.0013
   -0.0051
steps you need is 16
    16
```

Figure 19: Levenberg-Marquardt Alg, c0 = [1,1,1,1]'

As shown in Figure 19, this time we extimate c converges to the correct c. To figure out how large λ should be, I test λ starting from $1e-8$, and times ten each time until it reaches 10.

```
lambda is 0.0001
the estimate c is
   NaN
   NaN
   NaN
   NaN
step we need is 22
Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate.
RCOND = NaN.
lambda is 0.001
the estimate c is
   NaN
   NaN
   NaN
   NaN
step we need is 10
Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate.
RCOND = NaN.
lambda is 0.01
the estimate c is
   NaN
   NaN
   NaN
   NaN
step we need is 12
Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate.
RCOND = NaN.
lambda is 0.1
the estimate c is
   NaN
   NaN
   NaN
   NaN
step we need is 13
lambda is 1
the estimate c is
    1.0089
    0.5025
    1.9930
    0.0069
step we need is 11
lambda is 10
the estimate c is
    1.0089
    0.5025
    1.9930
    0.0069
step we need is 12
```

Figure 20: increasing lambda test

As shown in Figure 20, λ should be greater than 0.1 in order to converge.

We can use how many iterations it cost to reach the convergence point to measure the speed of convergence for different values of λ_1. The more steps it use, the lower the speed it has. So I test λ_1 in [1,50] and count the number of steps it cost to converge.

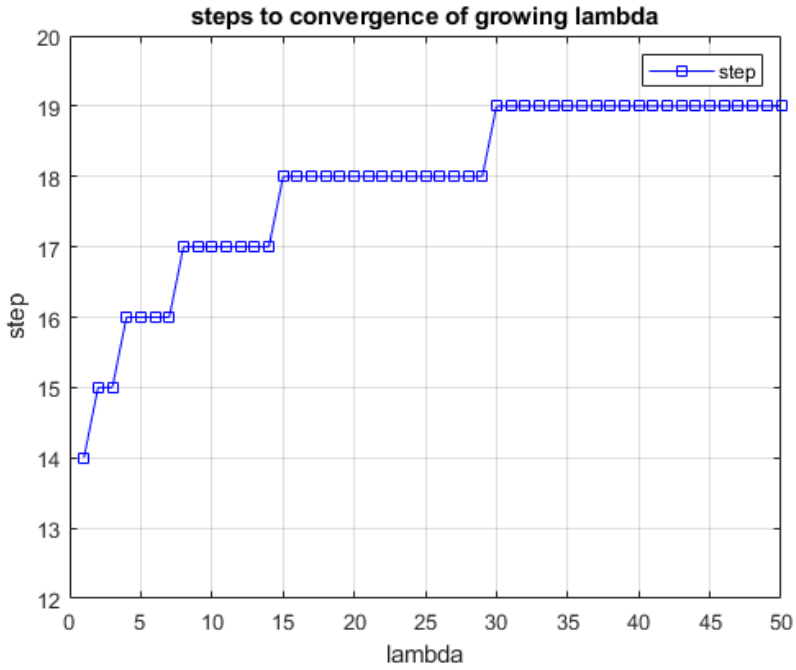

Figure 21: speed of convergence

As shown in Figure 21, the speed of convergence getting slower when λ_1 becoming larger.