

Numerical Analysis

2019 Spring Semester

Youngwan Kim

April 14, 2019

1 ROOT FINDING METHODS

Finding roots for a given function in various situation can be considered as the very essential problem of numerical analysis. We introduce some root finding algorithms and apply it for finding solutions of a single variable equation, and if it is available we will also extend it for solving simultaneous equations.

1.1 Bisection Method

The first method we introduce as a root finding algorithm is the bisection method. For a smooth function $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$ which we need to get its roots, an interval $[a, b] \subset I$ is given where $f(a)f(b) < 0$ which states that there is a zero point $p \in [a, b] : f(p) = 0$. For such initial value given, we then :

- 1) Get a, b and N, ϵ .
- 2) Let $i = 1$ and iterate until $i = N$
 - (1) Get $p = (a + b)/2$.

- (2) if $f(p) = 0$ or $f(p) < \epsilon$: break loop, output p
 - (3) if $f(p)f(a) > 0$: let $a = p$
 - (4) if $f(p)f(a) < 0$: let $b = p$
 - (5) $i = i + 1$
- 3) END

The ϵ used here is the tolerance, the amount of error that we could tolerate. So unless the method fails to find a point which $f(p) < \epsilon$ even with N iterations, the algorithm must stop at 2). The following MATLAB code is an example depicting the application of bisection method where $f(x) = \cos(x) - x$ and the initial interval is given as $[0, \pi/2]$.

Listing 1: bisec.m

```

1 function bisec=bisec(n)
2
3 f=inline('cos(x)-x');
4 a=0; b=pi/2;
5 for k=1:n
6     c=(a+b)/2;
7     if(f(c)>0) a=c; % we know that f(a)=f(0)>0
8     else b=c;
9     end
10 end
11
12 fval=f(c)

```

The code gets the iterating numbers as the only input and outputs the final value of $f(p)$, which is actually the error $|f(p) - 0|$, without the sign. Also for the given function as we know the sign of $f(0)$, we slightly modified the comparing condition. The following output of $n = 10, 50, 100$ shows that this method converges well if the initial value is given right.

```

octave:1> format long
octave:2> bisec(10)
fval = 0.00207533648652292
octave:3> bisec(50)
fval = 3.33066907387547e-16
octave:4> bisec(100)
fval = 0

```

1.2 Newton Method

1.2.1 Single Value Equation : No Multiplicity

The previous bisection method was rather simple as an algorithm as it just used evaluation and comparison. The Newton method we introduce at this section uses the fact that **any function can be approximated linearly near any point**, which is the idea of differentiation, to find roots numerically. We can't assert the convergence of this method, compared to the bisection method which always converges to the root when right initial values were given, but when it converges it shows up to be a very efficient method, which means that it fastly converges to its root. Also this method requires differentiation of the given function, which is quite a computational burden, but we'll talk about it later. The algorithm goes like this :

- 1) Choose the initial point x_0 .
- 2) Let $i = 1$ and iterate until $i = N$
 - (1) $x = x_0 - f(x_0)/f'(x_0)$
 - (2) if $|x - x_0| < \epsilon$ break loop ;
 - (3) let $x_0 = x$
- 3) END

The term $x_0 - f(x_0)/f'(x_0)$ is the gist of this method, which is derived from the linear approximation of $f(x)$ near x_0 . For points near x_0 ,

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(\xi)}{2!}(x - x_0)^2 \\ &\simeq f(x_0) + f'(x_0)(x - x_0) \end{aligned}$$

where we ignored $\mathcal{O}((x - x_0)^2)$ terms and solving this for x where $f(x) = 0$ gives us

$$x \simeq x_0 - \frac{f(x_0)}{f'(x_0)}$$

So we might question that " *If this is method that great why don't we just find 'good' initial values that would eventually converge?* " for the non guranteed convergence that differs by the initial value of this method, but actually that question itself is also a

very *chaotic* problem. For instance we can extend this method for analytic functions $f : \mathbb{C} \rightarrow \mathbb{C}$, and the convergence of each points result in a fractal, where we call such fractal a **Newton fractal**. Nothing is strange about extending the method to complex functions, as **linear approximation is the main idea of the method**. This idea will be used again and again for many other cases, with multiple variables and so on.

The following MATLAB code implements the Newton method to $f(x) = x^3 - 2ix - 5$ which is a complex function, $f : \mathbb{C} \rightarrow \mathbb{C}$. We used $x_0 = i$ as the initial value, and the single input to see how fast it converges.

Listing 2: newton1d.m

```

1 function newton1d=newton1d(n)
2
3 f=inline('x.^3-2*i*x-5');
4 df=inline('3*x^2-2*i');
5
6 x=i;
7
8 for k=1:n
9 x1=x-f(x)/df(x);
10 x=x1;
11 end
12
13 fval=f(x)

```

And the below is the outputs for quite low number of iterations, starting from absolutely no iteration $n = 1$ to nearly 10 iterations. Note that the needed steps to approximately get to the point where $f(x) \sim 10^{-16}$ decreased remarkably compared to the case of bisection method which nearly needed 50 iterations.

```

octave:1> format long
octave:2> newton1d(1)
fval = 0.700955848884844 + 2.471552116522530i
octave:3> newton1d(5)
fval = -5.14255305006373e-12 - 9.24993415196695e-12i
octave:4> newton1d(10)
fval = 0.000000000000000e+00 + 2.22044604925031e-16i

```

Now let us see how fast this method converges. We haven't explicitly shown it but the bisection method converged linearly, and now we will show that the Newton method converges quadratically. As we assumed no multiplicity of the root we will converge to, we can assume that $f(r) = 0$ and $f'(r) \neq 0$. Then for $f(r)$, we can expand it near the

initial point x_0 as

$$0 = f(r) \simeq f(x_0) + f'(x_0)(r - x_0) + \frac{f''(\xi)}{2!}(r - x_0)^2 \quad \text{where} \quad \exists \xi \in (r, x_0)$$

and simplifying some terms, we derive :

$$x_0 - \frac{f(x_0)}{f'(x_0)} = r + \frac{f''(\xi)}{2f'(x_0)}(r - x_0)^2$$

note that the LHS is the x_1 term of the Newton method so we can again simplify :

$$|x_1 - r| = \left| \frac{f''(\xi)}{2f'(x_0)} \right| (r - x_0)^2 \leq M(r - x_0)^2$$

which implies that the error $e_i = |r - x_i|$ will keep decrease, but in a quadratic sense as $e_1 \leq M e_0^2$. So for $\forall n \in \mathbb{N}$, we can say that $|x_{n+1} - r| \leq |x_n - r|^2$. This shows us that the Newton method converges quadratically, but as we stated before it has high dependence on the initial value which implies that this method isn't very stable.

As we keep mention it let us see how the slight differences in initial values can result in a very unexpected variance to the convergence, and how out of grasp this kind of problem is for us. The following MATLAB code `newton_fractal.m` plots a Newton fractal for $f(x) = x^3 - 2x - 5$.

To elaborate how this `newton_fractal.m` works, it first starts by generating a $n \times n$ matrix, which can be thought as a lattice included in \mathbb{C} , the region where we want to plot the Newton fractal. In this case we assigned $n = 250$, and the region to be $I \subset \mathbb{C}$ where $I = \{z \in \mathbb{C} : \Re(z) \in (-1.5, 2.5), \Im(z) \in (-1.5, 1.5)\}$ which is specified in line 9. Then for the main routine, each elements of the matrix will go through a Newton method routine and save a value differed by the value of the most adjacent root among the three known roots with the tolerance given $\epsilon \sim \mathcal{O}(10^{-6})$. Then after all the routines, we will obtain a $n \times n$ matrix filled with the values of 0, 1, 2 and 3. Note that not all values get assigned with 1, 2, 3. We then use internal functions to plot the given matrix, which will be an approximate version - *low resolution version, I would say* - of the real Newton fractal structure.

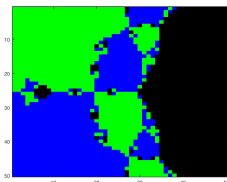
Listing 3: newton_fractal.m

```

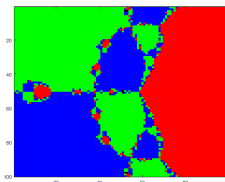
1 f=inline('x.^3-2*x-5');
2 df=inline('3*x^2-2');
3
4 xt(1)= 2.094551481542327;
5 xt(2)=-1.047275740771163 + 1.135939889088928i;
6 xt(3)=-1.047275740771163 - 1.135939889088928i;
7 % three complex roots for f(x)
8
9 n=250; rx=linspace(-1.5,2.5,n); ix=linspace(-1.5,1.5,n);
10
11 z=zeros(n,n);
12
13 for m=1:n
14     for j=1:n
15         x=rx(m)+ix(j)*i;
16         for k=1:25
17             x=x-f(x)/df(x);
18         end
19
20         if(abs(x-xt(1))<10^(-6)) z(m,j)=1;
21         elseif(abs(x-xt(2))<10^(-6)) z(m,j)=2;
22         elseif(abs(x-xt(3))<10^(-6)) z(m,j)=3;
23         end
24     end
25 end
26
27 z=rot90(z);
28 figure; imagesc(z)
29 colormap([0 0 0; 1 0 0; 0 1 0; 0 0 1])
30 % each colors : black , R , G , B

```

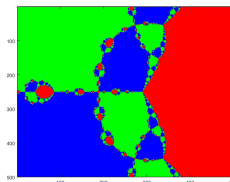
This code certainly has some heavy calculational complexity, approximately $\mathcal{O}(n^2)$ for the input value n , the wanted resolution of the Newton fractal plot.



(a) $n = 50$



(b) $n = 100$



(c) $n = 500$

1.2.2 Two Variable Equations

We can extend this method to two variable problems. Let $f_1(x, y)$ and $f_2(x, y)$ be two variable functions. We want to obtain (x, y) such that $f_1(x, y) = 0$ and $f_2(x, y) = 0$ at the same time. Using the linear approximating idea of Newton method, we can design an algorithm as following :

- 1) Get initial point $p_0 = (x_0, y_0)$
- 2) Let $i = 1$ and iterate until given N :
 - (1) The following equation

$$p = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} \partial_x f_1 & \partial_x f_2 \\ \partial_y f_1 & \partial_y f_2 \end{pmatrix}_{p_0}^{-1} \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}_{p_0}$$

- (2) let $p_0 = p$
 - (3) if $f_1(p) < \epsilon$ and $f_2(p) < \epsilon$ break loop;
- 3) END

The equation is again derived by using linear approximations of f_1 and f_2 . We'll let those approximated functions near p_0 as

$$\begin{cases} l_1 &= \partial_x f_1(p_0)(x - x_0) + \partial_y f_1(p_0)(y - y_0) + f_1(p_0) \\ l_2 &= \partial_x f_2(p_0)(x - x_0) + \partial_y f_2(p_0)(y - y_0) + f_2(p_0) \end{cases}$$

This approximation is available as $f_i(p_0) = l_i(p_0)$ and $\nabla f_i(p_0) = \nabla l_i(p_0)$ for all $i = 1, 2$. Thus we can reduce the problem of getting $(x, y) : f_1(x, y) = f_2(x, y) = 0$ to a rather less complicated computation of finding $(x, y) : l_1(x, y) = l_2(x, y) = 0$ between two linear functions. The following MATLAB code is an example of applying the above method letting $f_1 = 2x^3 + y^3 + xy - 6$ and $f_2 = x^3 - y^3 + xy - 4$.

Listing 4: newton2d.m

```

1 function newton2d=newton2d(n)
2
3 f=inline(' [2*x^3+y^3+x*y-6;_x^3-y^3+x*y-4] ');
4 df=inline(' [6*x^2+y_3*y^2+x;_3*x^2+y_-3*y^2+x] ');
5 x=[1 2]';
6

```

```

7  for k=1:n
8      x=x-inv(df(x(1),x(2)))*f(x(1),x(2));
9  end
10
11 fval=f(x(1),x(2))'

```

Again this gets a single input of amount of iteration, and outputs the final value after such number of iterations. The following is a set of outputs with 10, 25, and 50 iterations.

```

octave:1> format long
octave:2> newton2d(10)
fval = 1.62917432964171e-03    5.71500960422355e-06
octave:3> newton2d(25)
fval = 8.88178419700125e-16    8.88178419700125e-16
octave:4> newton2d(50)
fval = 1.77635683940025e-15    0.00000000000000e+00

```

1.2.3 Single Variable Equations : Roots with Multiplicity

Not every case the Newton method acts fast, even if its convergence is guaranteed. If a certain root has multiplicity, i.e $f(r) = f'(r) = \dots = f^{(m)}(r) = 0$ the convergence is slower than when it has no multiplicity. So we tweak our Newton method for roots with multiplicity : when we know the exact multiplicity of a certain root, or else when we actually have no info about the multiplicity but only know that it is not a single root.

First let us start with the case where the multiplicity is unknown. We denote the multiplicity $m \geq 2$ of $r : f(r) = 0$. Then, for some $g(x)$ such that $g(r) \neq 0$,

$$\begin{aligned}
 f(x) &= (x - r)^m g(x) \\
 \implies \frac{f(x)}{f'(x)} &= \frac{(x - r)^m g(x)}{m(x - r)^{m-1} g(x) + (x - r)^m g'(x)} \\
 &= (x - r) \left(\frac{g(x)}{m g(x) + (x - r) g'(x)} \right)
 \end{aligned}$$

then this f/f' has a simple root in $x = r$, which makes us available to apply Newton method. Then if we let $\{x_n\}$ be the Newton sequence of f/f' ,

$$\begin{aligned}
x_{n+1} &= x_n - \left(\frac{f(x_n)}{f'(x_n)} \right) / \left(\frac{f(x_n)}{f'(x_n)} \right)' \\
&= x_n - \left(\frac{f}{f'} \right) / \left(\frac{f'f' - ff''}{f'^2} \right) \\
&= x_n - \left(\frac{ff'}{f'^2 - ff''} \right)
\end{aligned}$$

would be the modified version for roots with unknown multiplicity. The Newton sequence has a second derivative, which isn't a very delightful situation as it brings us some calculational disadvantage.

Now we will consider the cases where the multiplicity is known, again we will let the multiplicity m , i.e, $f(r) = f'(r) = \dots = f^{(m)}(r) = 0$. This condition implies that $f^{(m-1)}(x)$ has $x = r$ as a single root. Then we will use the unmodified Newton method to $f^{(m-1)}(x)$. In this process, we use a useful approximation presented by Ralston & Rabinowitz (1978),

$$\frac{f^{(m-1)}(x)}{f^{(m)}(x)} \simeq m \cdot \frac{f(x)}{f'(x)}$$

As we claimed that we will apply Newton method to $f^{(m-1)}(x)$, let us consider the Newton sequence $\{x_n\}$ for it. Then,

$$\begin{aligned}
x_{n+1} &= x_n - \frac{f^{(m-1)}(x)}{f^{(m)}(x)} \\
&\simeq x_n - m \frac{f(x)}{f'(x)} \quad [\text{Ralston (1978)}]
\end{aligned}$$

becomes the Newton sequence to iterate, and thanks to the approximation of Ralston & Rabinowitz we derived a method with no high computational disadvantage. We will show that this approximation works, by showing it has quadratic convergence.

First we can expand $f(x_n)$ around the root $x = r$, for $\exists \xi, \eta \in (r, x_n)$:

$$\begin{aligned}
f(x_n) &= \sum_{n=0}^{m-1} \frac{f^{(n)}(r)}{n!} (x_n - r)^n + \frac{f^{(m)}(\xi)}{m!} (x_n - r)^m \\
&= \frac{f^{(m)}(\xi)}{m!} (x_n - r)^m \\
f'(x_n) &= \sum_{n=1}^{m-1} \frac{f^{(n)}(r)}{(n-1)!} (x_n - r)^{n-1} + \frac{f^{(m)}(\eta)}{(m-1)!} (x_n - r)^{m-1} \\
&= \frac{f^{(m)}(\eta)}{(m-1)!} (x_n - r)^{m-1}
\end{aligned}$$

where we used the multiplicity condition to reduce terms. This implies that

$$\exists \eta, \xi \in (r, x_n) \quad : \quad m \frac{f(x_n)}{f'(x_n)} = \frac{f^{(m)}(\xi)}{f^{(m)}(\eta)} (x_n - r)$$

Now we use the approximation of Ralson (1978) :

$$\begin{aligned}
x_{n+1} &= x_n - m \frac{f(x_n)}{f'(x_n)} = x_n - \frac{f^{(m)}(\xi)}{f^{(m)}(\eta)} (x_n - r) \\
x_{n+1} - r &= (x_n - r) - \frac{f^{(m)}(\xi)}{f^{(m)}(\eta)} (x_n - r) \\
&= (x_n - r) \left[1 - \frac{f^{(m)}(\xi)}{f^{(m)}(\eta)} \right] \\
&= (x_n - r) \left[\frac{f^{(m)}(\eta) - f^{(m)}(\xi)}{f^{(m)}(\eta)} \right] \\
\exists \nu \in (r, x_n) \quad : \quad &\leq (x_n - r)(\eta - \xi) \left[\frac{f^{(m+1)}(\nu)}{f^{(m)}(\eta)} \right]
\end{aligned}$$

And as $\exists \xi, \eta \in (r, x_n) \implies |\xi - \eta| \leq |x_n - r|$, and there exists some $M > 0$ such that

$$|x_{n+1} - r| \leq M|x_n - r||\eta - \xi| \implies |x_{n+1} - r| \leq M|x_n - r|^2$$

which implies that the error for such sequence quadratically converges to 0.

Now we will use $f(x) = \sin^2(x)$ as an example which has multiplicity of 2 for its root $x = 0$, and will apply all the three kinds of Newton method we introduced : the *vanilla* Newton, the unknown multiplicity modification, and the Raston Newton method. We will compare the amount of time for each method to break the iteration with identical tolerance with $\epsilon \sim \mathcal{O}(10^{-15})$. We used the internal time function `tic` and `toc` of MATLAB to measure how much time each methods used. Also for the initial value, we used the `rand` function, which outputs a random number in $(0, 1)$.

Listing 5: 'newton_modifications.m'

```

1 f=inline('sin(x)^2'); df=inline('sin(2*x)');
2 ddf=inline('2*cos(2*x)'); % for unknown multiplicity
3
4 x=rand;
5
6 tic
7 for i=1:20 % Non-modified Newton
8     x = x - f(x)/df(x);
9     if(f(x) < 10^(-15)) break; end
10 end
11 toc
12
13 tic
14 for i=1:20 % Unknown multiplicity modification
15     x = x - f(x)*df(x)/(df(x)*df(x)-f(x)*ddf(x));
16     if(f(x) < 10^(-15)) break; end
17 end
18 toc
19
20 tic
21 for i=1:20 % Known multiplicity, Raston 1978 modification
22     x=x-2*f(x)/df(x); % known multiplicity = 2
23     if(abs(df(x)) < 10^(-15)) break; end
24 end

```

As we expected, the Raston-Newton method shows up to be the fastest method.

```
octave:1> format long
octave:2> newton_modifications
Elapsed time is 0.001894 seconds.
Elapsed time is 0.000106096 seconds.
Elapsed time is 5.00679e-05 seconds.
```

1.3 Secant Method

The secant method is somehow a negotiation between the bisection and Newton method. It also uses linear approximation but not via the derivative concept. If the Newton method was using a single point to obtain the linear approximation of a certain function which can be thought of obtaining the instantaneous *slope*, the secant method uses the fact that we can also approximate such instant derivative by substituting it with the slope of the secant line between necessarily adherent points. Thus it eases derivational burden for obtaining a linear approximation while getting a somehow reasonable accuracy, and at the same time gains stability better than the Newton method.

Let a function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ with two given points as an initial value, let it $(x_0, f(x_0))$ and $(x_1, f(x_1))$. Then we approximate f to a linear function $l(x) : \deg(l) = 1$ which satisfies $f(x_0) = l(x_0)$ and $f(x_1) = l(x_1)$ at the same time. Obtaining such $l(x)$ is a quite elementary problem,

$$l(x) = \frac{f(x_0) - f(x_1)}{x_0 - x_1}(x - x_0) + f(x_0)$$

and we can easily show that it suffices the conditions we stated above. Using the root of this $l(x)$ we implement such algorithm :

- 1) Get initial points x_0, x_1
- 2) Let $i = 0$ and iterate until N
 - (1) Get $x = x_0 - f(x_0) \frac{(x_0 - x_1)}{f(x_0) - f(x_1)}$
 - (2) **temp** = x_1
 - (3) $x_1 = x$
 - (4) $x_0 = \mathbf{temp}$
 - (5) if $f(x) = 0$ or $f(x) < \epsilon$ break loop;

3) END

where we used a temporary variable `temp` to circulate the variables for the sequence to iterate through. The following MATLAB code implemented the above algorithm where $f(x) = x^3 - 2x - 5$ and the initial values are given as $x_0 = 0$ and $x_1 = i$.

Listing 6: secant.m

```
1 f=inline('x^3-2*x-5');
2
3 x0=0; x1=i;
4
5 for k=1:20
6     x = x1 - f(x1)*(x0-x1)/(f(x0)-f(x1));
7     temp=x1;
8     x1=x;
9     x0=temp;
10    if(abs(f(x))<10^(-15)) break; end
11
12    iter = k
13
14 end
```

Here we used $\epsilon \sim 10^{-15}$ as the tolerance for this algorithm. The output shows us how much iterations lead us to errors with ϵ tolerance.

```
octave:1> secant
iter = 14
x = -1.04727574077116 + 1.13593988908893i
Elapsed time is 0.000838041 seconds.
```

We can check that not even iterating until $k = 20$, the loop broke at $k = 14$.

Now we will calculate how this method converges towards the root. To recall, the bisection method had linear convergence, and the Newton method had quadratic convergence. To tell the results first, the secant method has $(1 + \sqrt{5})/2$ convergence. Let us check how.

Claim 1. $|x_2 - r| \leq M|x_1 - r||x_0 - r|$

Proof. The secant method states that :

$$x_2 = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)}$$

which implies that

$$\begin{aligned}
x_2 - r &= x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)} - r \\
&= \frac{x_0 f(x_1) - x_1 f(x_0)}{f(x_1) - f(x_0)} - r \\
&= \frac{(x_0 - r)f(x_1) - (x_1 - r)f(x_0)}{f(x_1) - f(x_0)} \\
&= \left(\frac{\frac{f(x_1)}{x_1 - r} - \frac{f(x_0)}{x_0 - r}}{f(x_1) - f(x_0)} \right) (x_0 - r)(x_1 - r)
\end{aligned}$$

Thus to prove the **Claim 1** is identical of showing that the coefficient in front of $(x_0 - r)(x_1 - r)$ is bounded. Let $g(t) = f(t)/(t - r)$. Then,

$$\begin{aligned}
x_2 - r &= \frac{g(x_1) - g(x_0)}{f(x_1) - f(x_0)} (x_1 - r)(x_0 - r) \\
&= \frac{g'(\xi)}{f'(\xi)} (x_1 - r)(x_0 - r)
\end{aligned}$$

for some $\xi \in (x_0, x_1)$ due to Cauchy mean value theorem. We again use the Cauchy mean value theorem for $g'(\xi)$. By definition of $g(t)$ we explicitly get

$$g'(\xi) = \frac{f'(\xi)(\xi - r) - f(\xi)}{(\xi - r)^2} = \frac{g_2(\xi)}{g_1(\xi)}$$

where we let $g_2(\xi) = f'(\xi)(\xi - r) - f(\xi)$ and $g_1(\xi) = (\xi - r)^2$. For such g_1, g_2 we know that $g_1(r) = g_2(r) = 0$ so we again apply Cauchy, for some $\zeta \in (\xi, r)$

$$\begin{aligned}
g'(\xi) &= \frac{g_2(\xi)}{g_1(\xi)} = \frac{g_2(\xi) - g_2(r)}{g_1(\xi) - g_1(r)} = \frac{g'_2(\zeta)}{g'_1(\zeta)} \\
&= \frac{f''(\zeta)(\zeta - r)}{2(\zeta - r)} = \frac{1}{2}f''(\zeta)
\end{aligned}$$

Thus, the coefficient is bounded as

$$\frac{g'(\xi)}{f'(\xi)} = \frac{f''(\zeta)}{2f'(\xi)} \quad \text{for} \quad \exists \xi \in (x_0, x_1) \text{ and } \exists \zeta \in (\xi, r)$$

□

Following the above method we can conclude that for any $n \in \mathbb{N}$, $|x_{n+2} - r| \leq M|x_{n+1} - r||x_n - r| \implies e_{n+2} \leq Me_{n+1}e_n$ for some $M > 0$. We then claim that

Claim 2. $e_{n+1} \leq Me_n^{\frac{1+\sqrt{5}}{2}}$, where $e_i = |x_i - r|$.

Proof. Let $s = \max\{e_0, e_1\}$, which implies that $e_0 \leq s$ and also $e_1 \leq s$. Then using the relation $e_{n+2} \leq e_{n+1}e_n$ we can derive that $e_2 \leq s^2$. Doing this inductively, we get an inequality

$$e_n \leq s^{\lambda_n} \quad \text{where} \quad \lambda_{n+1} = \lambda_n + \lambda_{n-1} \quad (\lambda_0 = 0, \lambda_1 = 1)$$

with an exponential Fibonacci term. We know that we can express the general terms of λ_n as

$$\lambda_n = c_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

where we will denote $\frac{1+\sqrt{5}}{2}$ as α and the other one as β . Note that $\alpha + \beta = 1$ and $\alpha\beta = -1$. Then,

$$e_{n+1} \leq e_{n-1}e_n = e_{n-1}e_n^{\alpha+\beta} = (e_{n-1}e_n^\beta)e_n^\alpha$$

and we will show that $e_{n-1}e_n^\beta$ is bounded.

$$\begin{aligned} e_{n-1}e_n^\beta &\leq s^{\lambda_{n-1}}s^{\beta\lambda_n} = s^{c_1\alpha^{n-1}+c_2\beta^{n-1}} \cdot s^{\beta(c_1\alpha^n+c_2\beta^n)} \\ &= s^{c_2(\beta^{n-1}+\beta^{n+1})} = s^{c_2\beta^n(\beta^{-1}+\beta)} \end{aligned}$$

As $|\beta| < 1$, the β_n term goes to 0 when $n \rightarrow \infty$, which implies that $e_{n-1}e_n^\beta$ is bounded for some $N > 0$. Thus,

$$e_{n+1} \leq Ne_n^\alpha = Ne_n^{\frac{1+\sqrt{5}}{2}}$$

□

As $\alpha \simeq 1.62$, we can conclude that this method has efficiency somewhere between Newton ($n = 2$) and bisection ($n = 1$). Let's compare the convergence of all of the methods we introduced.

- Bisection : $e_{n+1} \leq Me_n^1$
- Secant : $e_{n+1} \leq Me_n^{\frac{1+\sqrt{5}}{2}} \simeq Me_n^{1.6}$
- Newton : $e_{n+1} \leq Me_n^2$

Also do not forget about how to modify Newton method with functions with roots that have multiplicity, as regarding the multiplicity and modifying the method would enhance computation speed.

2 THE FLOATING NUMBER SYSTEM

Numerical analysis is useful not because we know there exists a convergent series towards some root or value we need to derive, but rather there exists high speed computational machines that makes us available to implement such numerical algorithms, and finally output the value we want with very little error. So it is important to understand how the machines percieve real numbers, and how they process and save them as binary data.

2.1 IEEE 754 Standard

Most of the machine these days are 64-bit, so we will consider 64-bit real number representation, which means that for a real number we correspond an array of 64 binary numbers. The IEEE 754 standard for machine numbers is given as the following rule :

$$s \ e_1 e_2 \dots e_{11} \ b_1 b_2 \dots b_{52} = (-1)^s \cdot (1.b_1 b_2 \dots b_{52})_{(2)} \cdot 2^{(e_1 \dots e_{11})_{(2)} - 1023}$$

where the first s is the **sign indictor**, the following 11 digits are called the **characteristic** which denotes the exponent term, and the last 52 digits are called the **mantissa** which denotes the fractional term.

This shows how different binary machines percieve real numbers compared to humans. The following MATLAB code is a very good example showing such discrepancy.

Listing 7: ridicule.m

```
1 a=0;
2
3 for i=1:10
4     a=a+0.1
5 end
6
7 if (a==1) display('ok')
8 end
```

It is a rather simple code to understand : add 0.1 ten times, and if it is 1 print a string. Of course $10 \times 0.1 = 1$ for us, but the following result says that it isn't so for machines.

```
octave:1> ridicule
octave:2>
```

This is because the representation of 0.1 for binary machines doesn't actually represent the real number 0.1. The following MATLAB code is implemented to show the 64-bit representation of the input value.

Listing 8: print_memory.m

```
1 function [total64bits,sign,exponent,mantissa]=print_memory(x)
2
3 s = sprintf('%bx',x);
```

5

machine.

```
octave:1> print_memory(a)
```

rather a slightly smaller value than 1.

We also need to consider which representations will correspond to ∞ and 0, or in other words we need to assign certain boundaries for *very small* and *very big* numbers. We define the smallest number representation as `realmin` and the biggest number representation as `realmax`. To elaborate, we let $s = 0$, $e = 1$, and $b = 0$ for `realmin`,

$$\text{realmin} = 2^{-1022} \cdot (1.0) \sim 0.22 \times 10^{-307}$$

represent $0 \in \mathbb{R}$. For `realmax` we let $s = 0$, $e = 2046$, and $b = 1 - 2^{-52}$,

$$\text{realmax} = 2^{1023} \cdot (2 - 2^{-52}) \sim 0.17 \times 10^{309}$$

and for representations between `realmax` and `inf` we call it **overflow** and let all of it represent as a breaking point value for programs. Again using `print_memory.m`, we explicitly see the 64-bit representation for `realmax`, `realmin`, `inf`, and 0.

```
octave:1> print_memory(realmin)
```


the derivation of the function $f(x)$ can be pre-calculated before plugging it into the algorithm, which also eases a step of computational burden of calculating the derivative of f .

Consider the Newton sequence $\{x_n\}$ for such $f(x)$,

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \left(\frac{b - 1/x_n}{1/x_n^2} \right) \\ &= x_n - (bx_n^2 - x_n) = 2x_n - bx_n^2 \end{aligned}$$

we will set the initial value $x_0 = 3/4$ as $1 \leq b < 2$ we can assert convergence in this region - as it holds bisection conditions, a root must be there. Letting such initial value we can derive that the initial error has its maximum around $1/4$, and as the error converges quadratically we can have certainty that the method will let us converge real close to the real value.

Listing 9: division.m

```

1 function division=division(n)
2 b=sqrt(2); % 1 < sqrt(2) < 2
3 x=3/4; err=0;
4
5 for k=1:n
6     x=2*x-b*x*x ; % Newton sequence iteration
7 end
8 err=abs(x-1/b)

```

The above MATLAB code is the implementation of the division algorithm we introduced. It fastly converges to the real value, and the following output shows it :

```

octave:1> format long
octave:2> division(1)
err = 0.00260191002141352
octave:3> division(2)
err = 9.57413496749382e-06
octave:4> division(5)
err = 0

```

3 POLYNOMIALS

3.1 Polynomial Interpolation

3.2 Integration Methods