Konkuk University Dept. Of Physics

Numerical Analysis

2019 Spring Semester

Youngwan Kim

April 13, 2019

1 Root Finding Methods

For some kind of real problems, it can be reduced into a problem of finding zeros of a certain function, and such problem can be considered as the very basic 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 $f : I \subset \mathbb{R} \to \mathbb{R}$ 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 $N, \epsilon$.
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

```
function bisec=bisec(n)

f=inline('cos(x)-x');
a=0; b=pi/2;
for k=1:n
  c=(a+b)/2;
   if(f(c)>0) a=c;  % we know that f(a)=f(0)>0
   else b=c;
   end
end

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 inital 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 comparision. 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 $f : \mathbb{C} \to \mathbb{C}$, 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, $f : \mathbb{C} \to \mathbb{C}$. We used $x_0 = i$ as the initial value, and the single input to see how fast it converges.

Listing 2: newton1d.m

```
function newton1d=newton1d(n)

f=inline('x.^3-2*i*x-5');
df=inline('3*x^2-2*i');

x=i;

for k=1:n
x1=x-f(x)/df(x);
x=x1;
end

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 =  0.00000000000000e+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 $f'(r) = 0$. 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 difference in initial values can result in a very unexpected variance to the convergence, and how out of grasp this is for us.

Listing 3: newton_fractal.m

```
f=inline('x.^3-2*x-5');
df=inline('3*x^2-2');

xt(1)= 2.094551481542327;
xt(2)=-1.047275740771163 + 1.135939889088928i;
xt(3)=-1.047275740771163 - 1.135939889088928i;

n=250; rx=linspace(-1.5,2.5,n); ix=linspace(-1.5,1.5,n);

z=zeros(n,n);

for m=1:n
 for j=1:n
```

```
  x=rx(m)+ix(j)*i;
   for k=1:25
    x=x-f(x)/df(x);
   end

  if(abs(x-xt(1))<10^(-6)) z(m,j)=1;
  elseif(abs(x-xt(2))<10^(-6)) z(m,j)=2;
  elseif(abs(x-xt(3))<10^(-6)) z(m,j)=3;
  end
 end
end

z=rot90(z);
figure; imagesc(z)
colormap([0 0 0; 1 0 0; 0 1 0; 0 0 1])
% each colors : black , R , G , B
```

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 variabled 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 $(x, y) : l_1(x, y) = l_2(x, y) = 0$. The following MATLAB code is an example where $f_1 = 2x^3 + y^3 + xy - 6$ and $f_2 = x^3 - y^3 + xy - 4$.

Listing 4: newton2d.m

```
function newton2d=newton2d(n)

f=inline('[2*x^3+y^3+x*y-6; x^3-y^3+x*y-4]');
df=inline('[6*x^2+y 3*y^2+x; 3*x^2+y -3*y^2+x]');
x=[1 2]';

for k=1:n
 x=x-inv(df(x(1),x(2)))*f(x(1),x(2));
end

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
```

1.3 Secant Method

2 The Floating Number System

2.1 IEEE 754 Standard

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.

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 :

$$\underline{s}\ \underline{e_1e_2\cdots e_{11}}\ \underline{b_1b_2\cdots b_{52}} = (-1)^s\cdot(1.b_1b_2\ldots b_{52})_{(2)}\cdot 2^{(e_1\cdots e_{11})_{(2)}-1023}$$

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 5: ridicule.m

```
a=0;

for i=1:10
 a=a+0.1
end

if (a==1) display('ok')
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 6: print_memory.m

```
function [total64bits,sign,exponent,mantissa]=print_memory(x)

%     [total64bits,sign,exponent,mantissa]=read_memory(x)
%     for double precision number x,
%     gives IEEE binary representation
%     as string of 64 characters (each is 0 or 1)

s = sprintf('%bx',x);
s = reshape(dec2bin(hex2dec(s'),4)',1,4*16);

total64bits=s;
sign=s(1:1);
exponent=s(2:12);
mantissa=s(13:end);
```

Now using the above code let us speculate what happened in the perspective of the machine.

```
octave:1> print_memory(a)
ans = 0011111111101111111111111111111111111111111111111111111111111111
octave:2> print_memory(1)
ans = 0011111111110000000000000000000000000000000000000000000000000000
octave:3> print_memory(0.1)
ans = 0011111110111001100110011001100110011001100110011001100110011010
```

We can check that $a = 0.1 + \cdots + 0.1$ is not percieved as 1 for binary machines, but rather a slightly smaller value than 1.

Also another thing we need to consider is that we need to assign ∞ and 0 for certain representations, or in other words we need to set certain boundaries for *very small* and *very big* numbers. We define the smallest number representation as `realmin` and the bigest number representation as `realmax`. To elaborate, we let $s = 0$, $e = 1$, and $b = 0$ for `realmin` ,

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

and for representations between 0 and `realmin` we call it **underflow** and let all of it represent $0 \in \mathbb{R}$. For `realmax` we let $s = 0$, $e = 2046$, and $b = 1 - 2^{-52}$,

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

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)
ans = 0000000000010000000000000000000000000000000000000000000000000000
octave:2> print_memory(0)
ans = 0000000000000000000000000000000000000000000000000000000000000000
octave:3> print_memory(realmax)
ans = 0111111111101111111111111111111111111111111111111111111111111111
octave:3> print_memory(inf)
ans = 0111111111110000000000000000000000000000000000000000000000000000
```

2.1.1 Division Algorithm

yeet

3 Polynomials

3.1 Polynomial Interpolation

3.2 Integration Methods