These are the lecture notes for CSC349A Numerical Analysis taught by Rich Little in the Spring of 2018. They roughly correspond to the material covered in each lecture in the classroom but the actual classroom presentation might deviate significantly from them depending on the flow of the course delivery. They are provide as a reference to the instructor as well as supporting material for students who miss the lectures. They are simply notes to support the lecture so the text is not detailed and they are not thoroughly checked. Use at your own risk.

1 Overview

We moved into a general overview of root finding. If we have an equation with one variable: $f(x) = 0$ then the value $\hat{x}$ for which this equation holds is called the *root* of the equation or a *zero* of the function $f(x)$.

There are two main families which we will consider. Polynomials such as $f(x) = x^3 + x^2 + x + 1$ which are a special case will be treated separately. For polynomials we will find both real and complex roots. More general, non-polynomial algebraic functions and the so called transcedental functions, such as $f(x) = e^{-x} - x = 0$ will be what we will cover first and we will focus on finding one or more real roots. Although in some cases it is possible to derive the *root* analytically, in many cases this is impossible in which case our only option is to use numerical methods.

In practical engineering problems, frequently there are cases were we can not rearrange an equation so that the unknown quantity is on one side of the equation and all the known quantities are on the other side. For example consider the equation describing the free fall of the parachutist that we used to motivate numerical methods:

$$v(t) = \frac{gm}{c}(1 - e^{ct/m}) \tag{1}$$

Suppose that instead of being given m, c, t and computing v, you want to know the value of c (drag coefficient) for a parchutist of mass m to attain a certain velocity after falling for time t. It is impossible to rearrange this equation so that c is on one side and all the known quantities are on the other. However it is trivial to express it as follows:

$$\frac{gm}{c}(1 - e^{ct/m}) - v = 0 \tag{2}$$

where g,m,t,v are known. Now the task is to find the value of c that makes this equation equal to 0 i.e solve $f(c) = 0$. In this case there is no analytic solution and we have to use numerical root finding methods such as the ones we will cover in this course.

All numerical methods are iterative i.e given one or more initial approximations to a root x_t they compute a sequence of approximations.

$$lim_{i\to\infty} x_i = x_t \tag{3}$$

In the 1830s it was proved that there are no finite algorithms involving $+, -, *, /$ for computing the roots of polynomials of degree n if $n \geq 5$.

2 Bisection

The **Bisection** method can be used to compute a zero of any function $f(x)$ that is continuous on an interval $[x_l, x_u]$ for which $f(x_l) \times f(x_u) < 0$.

Consider x_l and x_u as *two initial approximation* to a zero, say x_t, of $f(x)$. The new approximation is the midpoint of the interval $[x_l, x_u]$ which is $x_r = \frac{x_l + x_u}{2}$.

If $f(x_r) = 0$, the x_r is the desired zero of $f(x)$. Otherwise, a new interval $[x_l, x_u]$ that is half the length of the previous interval is determined as follows.

If $f(x_l) \times f(x_r) < 0$ then $[x_l, x_r]$ contains a zero, so set $x_u \leftarrow x_r$. Otherwise $f(x_u) \times f(x_r) < 0$ (necessarily) and $[x_r, x_u]$ contains a zero, so set $x_l \leftarrow x_r$.

The above procedure is repeated, continually halving the interval $[x_l, x_u]$, until $[x_l, x_u]$ is sufficiently small, at which time the midpoint $x_r = \frac{x_l + x_u}{2}$ will be arbitrarily close to a zero of $f(x)$.

Convergence Criterion

As this is an iterative algorithm that computes a sequence of approximations:

$$x_0, x_1, x_2, \ldots, x_{i-1}, x_i, \ldots$$

to a zero x_t, recall that we can use the iterative approximation relative error:

$$|\varepsilon_a| = \left|\frac{\text{current approx} - \text{previous approx}}{\text{current approx}}\right| = \left|\frac{x_i - x_{i-1}}{x_i}\right| = \left|1 - \frac{x_{i-1}}{x_i}\right|$$

is a good approximation to the actual relative $|\varepsilon_t|$ in x_i, and can be used to determine the accuracy of x_i.

Note that each approximation x_i is equal to $\frac{x_u+x_l}{2}$, and the previous approximation is either x_l or x_u. Therefore:

$$|x_i - x_{i-1}| = \frac{x_u - x_l}{2}$$

thus

$$|\varepsilon_a| = \frac{|x_i - x_{i-1}|}{|x_i|} = \frac{\frac{x_u-x_l}{2}}{\left|\frac{x_u+x_l}{2}\right|} = \frac{x_u - x_l}{|x_u + x_l|}$$

How many iterations n are required to obtain a desired accuracy?

Suppose you want the *absolute error* $< \varepsilon$, and that the length of the initial interval $[x_l, x_u]$ is Δx^0.

approximation	absolute error
$x_1 = \frac{x_l+x_u}{2}$	$\lvert x_t - x_1\rvert \leq \frac{\Delta x^0}{2}$
x_2	$\lvert x_t - x_2\rvert \leq \frac{\Delta x^0}{4}$
x_3	$\lvert x_t - x_2\rvert \leq \frac{\Delta x^0}{8}$
$\cdots$	$\cdots$
x_n	$\lvert x_t - x_n\rvert \leq \frac{\Delta x^0}{2^n}$

Table 1: Relation of approximation and absolute error

Therefore $\frac{\Delta x^0}{2^n} < \varepsilon$ implies that $2^n > \frac{\Delta x^0}{\varepsilon}$ and $n > log_2\left(\frac{\Delta x^0}{\varepsilon}\right)$ or

$$n \ln 2 > \ln(\Delta x^0) - \ln(\varepsilon) \quad \text{and} \quad n > \frac{\ln(\Delta x^0) - \ln(\varepsilon)}{\ln 2}$$

Example
If initially $x_u - x_l = \Delta x^0 = 1$ and $\varepsilon = 10^{-5}$, then the above formula gives $n > 16.61$. Thus, 17 iterations would *guarantee* that the *absolute error* of the computed approximation to a zero x_t of $f(x)$ is $< 10^{-5}$.

2.1 Example

Use the Bisection method to find the positive root of $f(x) = x^2 - 3$ with $|E_t| < 0.01$.

First note that

$$x^2 - 3 = 0 \Longrightarrow (x + \sqrt{3})(x - \sqrt{3}) \Longrightarrow x = \pm\sqrt{3} = \pm 1.73205...$$

Thus, we will start with $x_l = 1$ and $x_u = 2$, making $\Delta x^0 = 1$ and we know $\varepsilon = 0.01$. This means that we can calculate the number of iterations it will take to find the approximate root under these conditions.

$$n \geq \frac{\ln(\Delta x^0) - \ln(\varepsilon)}{\ln(2)} = \frac{\ln(1) - \ln(0.01)}{\ln(2)} = \frac{4.60517}{0.693147} = 6.64...$$

Therefore it will take $n = 7$ itereations of the Bisection method to find this root. I will summarize the results in the following table:

iteration	x_l	x_u	$f(x_l)$	$f(x_u)$	$x_r = \frac{x_l + x_u}{2}$	$f(x_r)$	$\lvert E_t \rvert$	update
1	1	2	-2	1	1.5	-0.75	0.5	$x_l = x_r$
2	1.5	2	-0.75	1	1.75	0.0625	0.25	$x_u = x_r$
3	1.5	1.75	-0.75	0.0625	1.625	-0.359	0.125	$x_l = x_r$
4	1.625	1.75	-0.359	0.0625	1.6875	-0.1523	0.0625	$x_l = x_r$
5	1.685	1.75	-0.1523	0.0625	1.7188	-0.0457	0.0313	$x_l = x_r$
6	1.7188	1.75	-0.0457	0.0625	1.7344	0.0081	0.0156	$x_u = x_r$
7	1.7188	1.7344	-0.0457	0.0081	1.7266	-0.0189	0.0078	

Table 2: Iterations of the Bisection Method

Therefore $x_t \approx x_r = 1.7266$. (Note: the better choice here is $x_u = 1.7344$ since $f(x_u) = 0.0081$ is closer to 0).

3 Some more MATLAB programming

Functions as arguments

Numerical methods frequently are expressed as iterations that require to evaluate a function (and sometimes its derivatives) for multiple points. Examples include Euler's method which requires the evaluation of the slope in the incremental update, the bisection method that requires the evaluation of the function at the boundaries and midpoint of the interval under consideration in each iteration, and the Newton-Raphson that requires evaluation of a function at the current root estimate as well as its first derivative in

order to calculate the improved estimate. From a programming perspective in all these cases we would like to abstract the numerical algorithm (Euler's method, Bisection, Newton/Raphson) from the specific function being evaluated. In the same way that we can generalize a function by adding a parameter or argument we would like to generalize our methods to arbitrary functions.

To make this more concrete let's start with some simple examples. Consider the following function in MATLAB (in a file mypow.m):

```
function [y] = mypow(x)
       y = x^2;
end
```

We can call this function in the command window as follows:

```
>> mypow(2)
ans = 4
```

We can generalize this function by adding an extra argument that is the power that x is raised to as follows:

```
function [y] = mypow(x,n)
       y = x^n;
end
```

Now that we have a function of two arguments we can use it in different ways as follows:

```
>> mypow(2,2)
ans = 4
>> mypow(2,3)
ans = 8
>> mypow(2,4)
ans = 16
```

In some ways by introducing the second argument we have generalized the function from one that only computes the square of its input to many different functions of one argument parametrized by the second argument n. Similarly we would like to generalize a function for a numerical method such

as Euler's method from one that solves a particular specific problem to one that is more general and parametrized by an argument that somehow capture the specific problem under consideration. This extra argument needs to be a function rather than a number. MATLAB requires the special command *feval* to achieve this (this is because functions are not first class citizens in MATLAB. In a functional language such as Haskell they are and can be used as any other type (i.e passed as arguments, placed in data structures, etc).). The command *feval* takes two or more arguments. The first argument is the name of the function to be applied, and the remaining arguments are the arguments that are passed to it. The following examples should clarify how it operates:

```
>> feval('cos', 2*pi)       % evaluate the cosine function at 2pi
ans =

     1
>> feval('mypow', 2, 3)    % evaluate the mypow function with
                           % arguments 2 and 3
ans =

     8
```

Notice that the syntax *feval(mypow, 2,3)* will not work as *feval* expect the first argument to be the string name of the function.

3.1 Example

The volume of liquid in a spherical tank is given by

$$V = \frac{\pi h^2(3R - h)}{3}$$

where h is the depth of the liquid and R is the radius. If $R = 3$, to what depth must the tank be filled so that it contains $V = 30m^2$ of water? Use the Bisection Method to solve this problem within 1000 iterations in MATLAB.

```
function [ fh ] = height( h )
    fh = (pi*h^2*(9 - h))/3 - 30;
end
```

```
>> root = Bisect(1,3,10^(-3),1000,'height')
 iteration approximation
      1         2.00000000
      2         2.50000000
      3         2.25000000
      4         2.12500000
      5         2.06250000
      6         2.03125000
      7         2.01562500
      8         2.02343750
      9         2.02734375
     10         2.02539063

root =

    2.0254
```

Note: the exact answer is 2.02690...

I strongly encourage you to experiment with *feval*, passing functions as arguments to other function and global variables, on your own, trying the examples provided as well as your own scenarios directly in MATLAB to understand the concept. Just reading about it and attending lectures is not enough.

Advantages of the Bisection Method:

- If $f(x)$ is continuous and if appropriate initial values x_l and x_u can be found, then the method is **guaranteed to converge**.

Disadvantages

- converges slowly (requires more iterations than other methods)
- it may be difficult to find appropriate initial values
- it cannot be used to compute a zero x_t of **even multiplicity** of a function $f(x)$; that is, if

$$f(x) = (x - x_t)^m g(x)$$

where m is a positive even integer and $g(x_t) \neq 0$