CMPE-380: Applied Programming

Laboratory Exercise 07

Root Finding Methods

Table of Contents

Root Finding

Pre-Lab – 10 pts

In this lab session, we will review Root Finding. Please review and understand the following root finding methods and sample code . This pre-lab has an extensive list of questions to be answered at the end.

During this lab we will cover two main approaches:

Bracketing methods

- Bisection

Open methods

- Newton
- Secant (quasi-Newton)

Root Finding Methods

Why We Need Root Finding?

Example: Consider an electric motor that operates in the 0 to 50V range. We found, by **data fitting**, that the RPM of the motor depends on the applied voltage

$$f(v) = 52.2\,v + 0.75\,v^2 - 0.02\,v^3$$

What **voltage** must be applied to the motor to run it at **1909** RPMs?

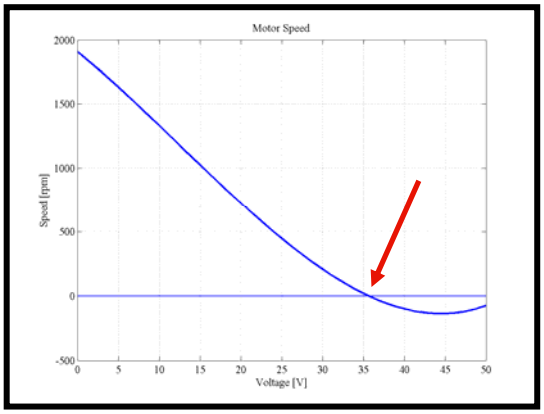
Re-arrange the equation in the "canonical form": f(v)=g(v)-h(v) = 0:

Where: $h(v) = -0.02v^3 + 0.75v^2 + 52.2v = 0$ & g(v) = **1909** (our desired rpm)

So: $f(v) = 0.02\,v^3 - 0.75\,v^2 - 52.2\,v + 1909 = 0$

We could analytically solve this simple equation, but we will use numerical methods instead.

Graphically we want to solve for the zero-crossing point



Bisection Method

An astute observer will notice the value of the function f(v) is positive near the left side of the solution (root) and negative on the right side of the solution (root). We need to develop a numerical method where we guess two values of v and, using our knowledge of the signs, halve the guess and try again until the value of the root is close to zero.

The Bisection Method

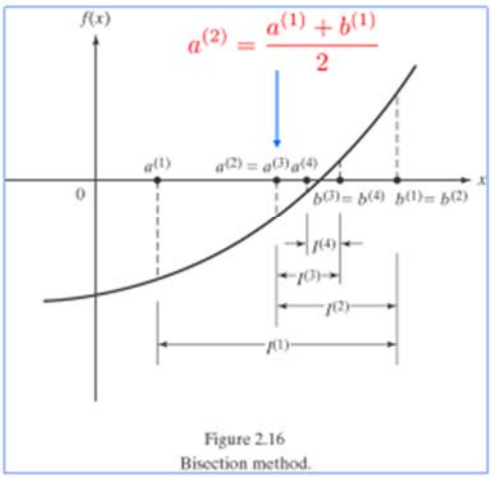


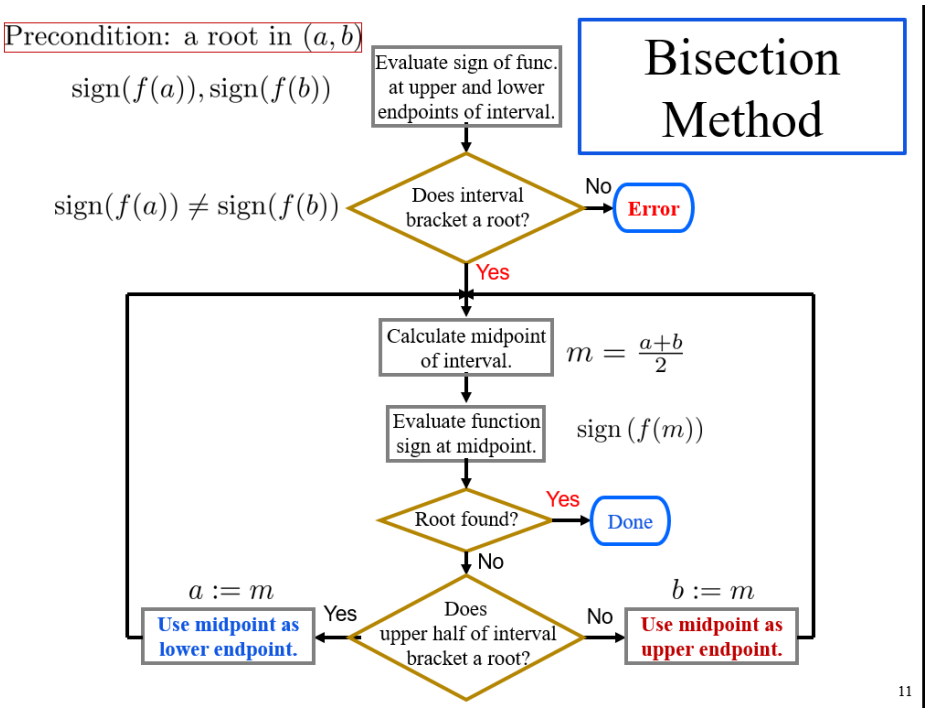
Figure 2.16
Bisection method.

- ***Halve the size*** of the bracketing interval enclosing the root (e.g., a binary search)
- Choose the new smaller bracket that includes (brackets) the root.
- Repeat ***until bracket size*** is ***small enough***
- Root inclusion criteria: value of function has ***opposite signs at bracket endpoints***

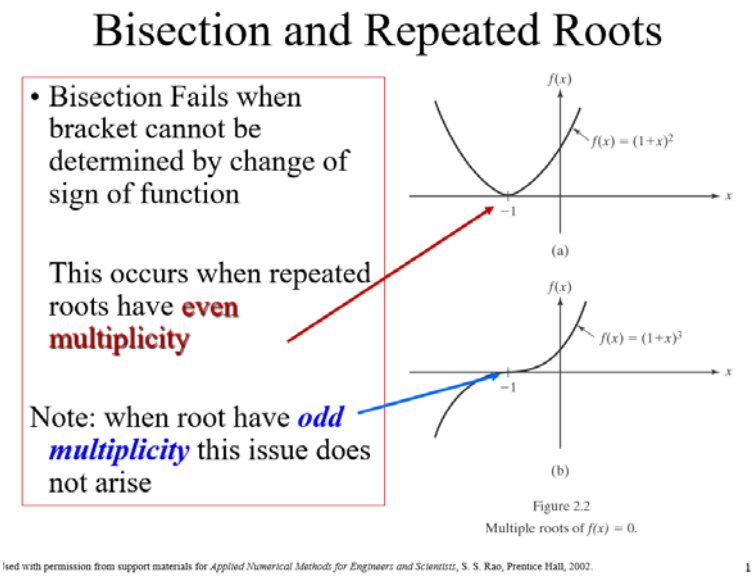
Used with permission from support materials for *Applied Numerical Methods for Engineers and Scientists*, S. C. Rao, Prentice Hall, 2002.

10

Consider the following flow chart:



Bisection will ONLY work if the two guesses A & B bracket the root as shown in the first decision block. Bisection can fail if the function has an even number of roots.



In all numerical solutions the value of the root found is almost NEVER zero (0.0) but rather a very small number near zero. In any root finding algorithm a non-zero tolerance must be specified and used.

Open Methods

Unlike bisection, open methods only really require one "initial guess" and it doesn't need to be close to the root. Open methods start with an "initial guess" and iteratively refine the guess to get closer to the "true" root. Open methods may diverge (the algorithm may proceed down the "wrong path") and **never find a root**. Open methods always require an "escape hatch" to prevent infinite loops. Like other root finding methods the value of the root is never exactly zero, so a tolerance is required.

Bracketing vs. Open Methods

Bracketing	Open
• *Refine the interval* in which root is contained	• *Refine the value* of the initial guess of the root
• *Guaranteed to converge* (as long as root in bracket)	• *May diverge* (converge only when "close" to the root)
• *Slow* Converge (linear)	• *Fast* convergence (superlinear, quadratic)

Newton's Method

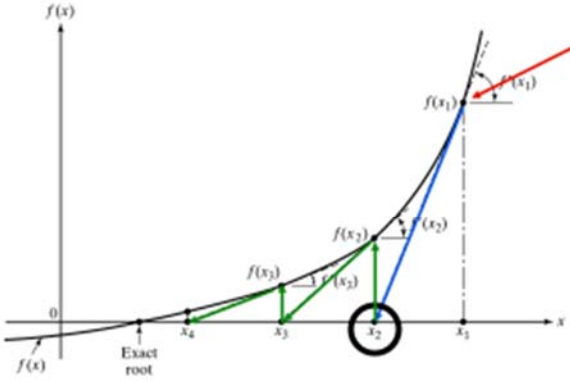
The Newton method uses information about the ***slope of the function*** (*e.g.*, its derivative) ***at the current point*** to refine the current root estimate. Convergence is guaranteed only when the guess is ***"close" to the solution***; otherwise it may diverge. In general, we don't know how close is close enough. The Newton method uses the local slope of the function and the intersection of that slope with the X axis to create an improved guess. Newton follows the slope down to the root.

The Newton update equation is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Note: f'(x) can't be zero or very small.

Newton's Method in Action



- Start with an initial guess (x_1) of the root and
- Use tangent at current point to approximate *f(x)* locally
- Use intersection of tangent with y=0 to obtain next point.
- Repeat until accuracy is satisfied.

Figure 2.17
Newton's method.

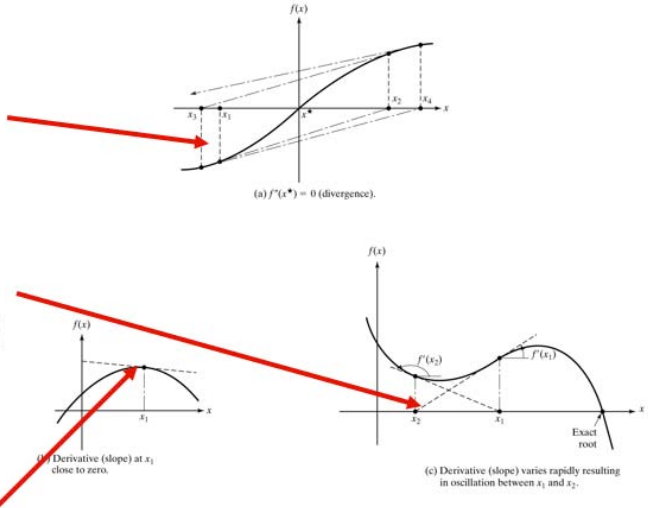
Used with permission from support materials for *Applied Numerical Methods for Engineers and Scientists*, S. S. Rao, Prentice Hall, 2002.

7

As with bisection, the Newton method has error cases:

Newton's Algorithm Failures

- Slope of function away from root is a bad predictor (*diverges*)
- In theory we could get trapped in a *non-convergent cycle* (oscillates)
- $f'(x_i)$ very small (close to zero)



(a) $f''(x^*) = 0$ (divergence).

(b) Derivative (slope) at x_1 close to zero.

(c) Derivative (slope) varies rapidly resulting in oscillation between x_1 and x_2.

Figure 2.18
Non-convergence of Newton's method.

These could be avoided with a better initial guess !

with permission from support materials for *Applied Numerical Methods for Engineers and Scientists*, S. S. Rao, Prentice Hall, 2002.

In addition to the function to be solved, the Newton method requires a closed form expression for the ***derivative.*** Our next method does not require the derivative.

Secant Method

The Secant method is similar the Newton method, but it uses an approximation to the derivative to traverse the slope. Secant achieves the slope approximate by providing a 2nd "guess", near the first guess that is used to calculate the slope. Unlike bisection, the 2nd guess does not have to bracket the root, it is just used to calculate the slope. Typically, the 2nd guess is + 0.001* of the first guess, so the 2nd guess is not strictly required from the use and could be "assumed" by the application.

The Secant update method is: $$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Note: the denominator can't be zero or very small or we will get a floating-point overflow error.

Answer the following questions in**: prelab.txt**

1) What is the minimum requirement for root finding using bisection?
2) How do you know when you have found a root?
3) Is the root found ever exactly zero?
4) What is the basic bisection algorithm?
5) When you bisect the roots, how do you pick the endpoint root to change?
6) What is the basic idea behind Newton and Secant algorithms?
7) Why is an escape hatch required for Newton and Secant?
8) What special requirement does Newton have that Secant doesn't.
9) What are the Newton and Secant update equations?
10) How does the 2nd guess of the Secant differ from the 2nd bisection guess?
11) Why doesn't Newton require a second guess?
12) Why do we use the fabs() function instead of the abs() function in all of our root finding code?
13) In Newton, why do we check for a zero-slope using: **if (fabs(dx0) < atol)** rather than **if (fabs(dx0) == 0.0)?**

Interactive Exercises – 30 pts

In this exercise we test various root finding algorithms. Redirect the output from your programs, other commands and answers to questions to **exercise.txt** and show the results to your TA when you are finished. You will be documenting your comments in exercise.txt, be sure to clearly indicate what step you are commenting on. You will want to test each stage of the exercise before redirecting the output into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

Newton Example

In this example you will find the solution (root) for the motor problem described in the prelab. Copy the include files from the root directory of this lab.

1) Modify the provided **newton.c** to find the solution voltage which will result in **1909** RPM. The motor function is: **f(v) = 52.2 v + 0.75 v2 – 0.02 v3**. Assume the roots are between **0 and 50**. Use an initial guess of **1.0**
2) What is the derivative of the motor function, why is it needed?

3) Change the guess to **30.0**, recompile and re-run.
4) Does newton find the same root? What can you say about the number of iterations?

5) How many roots does the motor function really have? Why?
6) Find the other roots using guesses of **50.0** & **-50.0**.

Bisection Example

In this section you will implement the Bisection algorithm as outlined in the prelab. A **bisection.c** framework file is provided to implement your solution.

1) Modify the provided **bisection.c** to find the solution voltage for the motor function: **$f(v) = 52.2\ v + 0.75\ v^2 - 0.02\ v^3$** which will result in **1909** RPM.
2) Why are the voltage constants in the code 0.0 and 50.0 and not 0 and 50? Does it matter?
3) What do the variables "a" and "b" represent in verbose output?

4) Implement the bisection algorithm using the comments in the bisection.c framework code and the prelab notes.

5) Find all the other roots using the bisection code. What are the roots?
6) Does bisection find the same roots as newton? Which algorithm seems faster?
7) What can you say about the number of iterations between bisection and newton?

Assignment – 60 pts

Objective

Implement a C module with the most common root-finding algorithms and test them for efficiency and robustness. Implement a C module with getopt_long_only(). Use your **ClassErrors.h** file from your earlier assignment. A results.txt file is provided as an example of the required output.

Implementation

You will implement a C module called ***rootfinding*** consisting of the pair (**rootfinding.h**, **rootfinding.c**). In the module you will implement the Bisection method, Newton's method and the Secant method. You can add to the header file as you see fit, but do not change the function prototypes given. If you add to the .h file, make sure to justify and document the changes in your **analysis.txt** file.

Your implementation (**rootfinding.c**) should be ***efficient and robust***. At minimum, it should provide safety mechanisms to check preconditions and avoid an infinite number of iterations. The Secant method is very similar to the Newton method so the easiest way to create the Secant code is to copy and modify the Newton code.

Program Organization

- You should use **eqn2solve.c** to define the equations to be solved.
- You should be able to compile the modules independently of the driver program, that is, you will generate an object file called **rootfinding.o** and link it to the driver program as required.

Testing and Specifications

a) Write a "driver" program called **hw7.c** to test your module. The driver program must use **getopt_long_only()** to parse command line parameters and should be designed such that it takes the following inputs:

hw7 -bisection | -secant | -newton --tolerance num -guessa num <-guessb num> <-verbose>

-bisection -tolerance num -bracket_a num -bracket_b> <-verbose>
-secant -tolerance num -guessa num [-guessb num] <-verbose>
-newton -tolerance num -guessb num <-verbose>

Note: The following abbreviations should be allowed on the command line:

```
-verbose      -verb, -v
-bisection    -b
-secant       -s
-newton       -n
-tolerance    -tol,  -t
-guessa       -ga,   -g
-bracket_a    -ba    -g
-guessb       -gb    -u          // Might be unused for Secant if you implement that way
-bracket_b    -bb    -u
```

Example: **./hw7 -bisection -tol 1E-6 -ga -2 -gb 3 -verb** at the prompt will run the bisection method with tolerance 1*E−6* in the bracket [−2,3] and print all partial results (verbose on)

./hw7 -newton -t .000001 -ga -1.5 at the prompt will run the newton method with tolerance 1*E−6* starting at −1.5, with no verbose output

- If verbose is set, the program should print the results to **stdout** in ***tabular form*** (so that they can be redirected to a text file if desired.) Each line of the output should be formatted as shown.

Bisection should be: **iter a b x err**

```
iter:0  a:1.500000      b:2.000000      x:1.750000      err:0.250000
iter:1  a:1.750000      b:2.000000      x:1.875000      err:0.125000
```

Secant/Newton should be: **iter x0 x1 er**

```
iter:0  x0:-2.000000    x1:5.000000     err:2.288493
iter:1  x0:0.288493     x1:-2.000000    err:6.416755
```

- All errors messages should be printed to **stderr** (not to **stdout**).
- Verbose output is not an error and should be printed to **stdout**.
- Verbose error is defined as the difference between the last two "guesses".
- **The driver program should check for the right number and type of arguments, print friendly messages and should return an appropriate error code (non-zero integer) for each error encountered.**

b) All testing should be automated via a **makefile**

TestCases

Find a real root of the function (note x is a variable)

$$\mathbf{f(x) = 0.76\ x\ \sin(\frac{30}{52}\ x)\ \tan(\frac{10}{47}\ x) + 2.9\ \cos(x+2.5)\ \sin(0.39(1.5+x))}$$

accurate to 1×10^{-6}.

- For *Bisection* use the interval [−2.5,2.5].
- For Newton's method use **two** different initial guesses: −1.5 and 1.5
- For the Secant method use the same initial guesses as in Newton's case and provide a suitable additional second point for initialization (justify your choice in the analysis part.)

Debugging and Timing the Code

Debug your module functions and make sure that you have an efficient implementation. Try different optimization levels, (**gcc** option **-O1 .. -O3**) and select the one that gives the "best" result. Record timing of your algorithms (per iteration), clearly identifying each test case and sample number and save in your analysis file. As always, average 5 iterations of each test case and use the average in your calculations.

Makefile

Write a make file with the following targets:

"all" -should make **hw7**

"bisection, secant, newton," - should run the appropriate test with the identified parameters and output the results to **out.txt**

"tests" - should run all the above test, in a row, and output all to **out.txt**

"opts" - should run the bisection test using all of the following abbreviations:
-b -tol 1E-6 -ga -2 -gb 3 -verb
-b -t 1E-6 -g -2 -u 3 -v
-b -t 1E-6 -ba -2 -bb 3 -v redirected to out.txt

"mem" - Run valgrind for all three modes of the hw7 with good data redirecting output to mem.txt. e.g: bisection -2.5, 2.5 secant -1.5 -1.6 newton -1.5

help, clean- should do the normal things

Note: The makefile should include a "verbose" makefile macro and it should be disabled by default.

Results and Analysis

Write in the file **analysis.txt** an explanation of the results obtained. In the first section, summarize and document briefly your approach to optimization, the performance observed for each of the programs/executions and explain the factors behind the performance observed and the roots found.

Grading Criteria

1. (45 points) Correct program(s)/algorithms.
   (a) (9 points) Makefile works properly
   (b) (6 points) Bisection method
   (c) (6 points) Secant method
   (d) (12 points) Newton's method

   (g) (12 points) No memory leaks

2. (15 points) Analysis
   (a) (5 points) Conclusion on compiler optimization changes for each method and conclusion of the overall wall time efficiency for root finding using the "best" optimization with data to support your position.
   (b) (7 points) Compare and contrast each root finding method for each starting point, does each algorithm find a valid root? Do they find the same root? Why or why not?
   (c) (3 points) All raw timing, performance and other data included.

Note:

1. There is **NO** requirement to automatically generate the derivative function for Newton, you can use offline tools to generate the derivative.
2. Please note that you can use the implementations you created during the lab for the Bisection and Newton methods!
3. **Your code MUST validate command line parameters!! (e.g. idiot checking)**

Student Name: _____

Laboratory Grading Sheet

Lab 07 - Root Finding Methods

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: prelab.txt	**10**		
Interactive Exercises: Newton	**5**		
Interactive Exercises: Bisection	**25**		
Total	**40**		

You must turn this signed sheet in at the end of lab to receive credit!