# Homework #5-Root-finding Methods

***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 **ClassError.h** file from HW 4. Demonstrate the performance advantage of Qn integer calculations over floating point.

## 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 infinite number of iterations.

## Program Organization

- You should use **eqn2solve.c** to define the equations to be solved.
- You should be able to compile the module 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 **hw5.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:
***name of the root finding method***, (**bisection**, **newton** or **secant**) followed by the ***desired tolerance***, followed (depending on the method) by either two real numbers (specifying ***the bracket limits*** [*a*,*b*] or the ***initial guesses*** $x_0$ and $x_1$) or one real number (with the guess $x_0$ and then followed by an ***optional verbose flag*** if partial results are to be printed.

```
hw5  -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
-bracket_b     -bb    -u
```

***Example:*** **./hw5 -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)

**./hw5 -newton -t .000001 -ba -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 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 $1\times10^{-6}$.

- 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 **-O**) and select the one that gives the "best" result. Record timing of your algorithms (per iteration) and save in your results file.

## Qn performance

A testing template program called **intTest.c** is included in this lab. You will rewrite your floating point bisection solver code using Qn format integer code and investigate the performance differences. Your implementation will use the "long" integer data type (64 bit on this machine). You will need to decide on the value of "n" to use, be sure to include your justification in the analysis.txt file.

Rootfinding.h contains the #define QN, modify this as you see fit. QN represents the fractional number of bits in the Qn format. The include file also includes the following Qn mathematical macros, use them in your solution.

FLOAT_TO_FIX(x)
FIX_TO_FLOAT(X)
Qn_DIVIDE(A,B)
MUL_FIX(A,B)

These macros utilize the value of "QN" in the include file.

The function you will be "finding the solution to" is **y = 0.5X-0.3**. The bounding guess and tolerance are: -25.0, 25.0 and 0.000001, respectively. The goal is to solve the equation using both your floating point bisection algorithm and your

integer Qn format bisection algorithms, comparing the performance and accuracy. Recode your floating point bisection algorithm in the ibisection() function of intTest.c.

Modify the main() function in intTest.c to call both bisection algorithms and compare the performance and errors. Use your initial Qn guess and then try other larger and smaller values of Qn, provide a summary in your analysis.txt file. Why does changing the Qn value change the error?

### Makefile

Write make file with the following targets: all, bisection, secant, newton, integer, tests, opts, clean, help

"all" -should make **hw5** and **intTest**

"bisection, secant, newton, integer" – 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

help, clean - should do the normal things

### Results and Analysis

Write in the file **analysis.txt** a short 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.

In the second section include your analysis from the Qn investigation.

Submit a tarball with all C program(s) together with your analysis and Makefiles.

**GradingCriteria**

1. (60 points) Correct program(s)/algorithms.
   (a) (10 points) Makefile works properly
   (b) (10 points) Float Bisection method
   (c) (10 points) Secant method
   (d) (10 points) Newton's method
   (e) (10 points) Integer Bisection method
   (f) (10 points) Command line options work properly

2. (40 points) Analysis
   (a) (15 points) Correct roots found for all programs/algorithms:
      i. (5 points) Bisection method root.
      ii. (5 points) Secant/Newton's method root from initial guess of $x = -1.5$
      iii. (5 points) Secant/Newton's method root from initial guess of $x = +1.5$

   (b) (15 points) Explanation of number of iterations for each program/algorithm:
      i. (5 points) Bisection method iterations.
      ii. (2.5 points) Newton's method iterations from initial guess of $x = -1.5$ .
      iii. (2.5 points) Newton's method iterations from initial guess of $x = +1.5$ .
      iii. (2.5 points) Secant method iterations from initial guess of $x = -1.5$
      iv. (2.5 points) Secant method iterations from initial guess of $x = +1.5$ .

   (c) (5 points) Timing information provided.
   (d) (5 points) Impact of changing Qn values on the integer Bisection method.

Note: There is **NO** requirement to automatically generate the derivative function for Newton, you can use offline tools to generate the derivative.