MATH2070: LAB 3: Roots of Equations

Introduction	Exercise 1
A Sample Problem	Exercise 2
The Bisection Idea	Exercise 3
Programming Bisection	Exercise 4
Variable Function Names	Exercise 5
	Exercise 6
Convergence criteria	Exercise 7
The secant method	Exercise 8
Regula Falsi	Exercise 9
Muller's method (Extra)	Extra Credit

1 Introduction

The *root finding* problem is the task of finding one or more values for a variable x so that an equation

$$f(x) = 0$$

is satisfied. Denote the desired solution as x_0. As a computational problem, we are interested in effective and efficient ways of finding a value x that is "close enough" to x_0. There are two common ways to measuring "close enough."

1. The "residual error," $|f(x)|$, is small, or
2. The "approximation error" or "true error," $|x - x_0|$ is small.

Of course, the true error is not known, so it must be approximated in some way. In this lab we will look at some nonlinear equations and some simple methods of finding an approximate solution. We will use an estimate of the true error when it is readily available and will use the residual error the rest of the time.

You will see in this lab that error estimates can be misleading and there are many ways to adjust and improve them. We will not examine more sophisticated error estimation in this lab.

This lab will take two sessions. If you print this lab, you may find the `pdf` version more appropriate.

2 Remarks on programming style

You can find Matlab code on the internet and in books (Quarteroni, Sacco and Saleri's book is an example). This code appears in a variety of styles. The coding style used in examples in these labs has an underlying consistency and is, in my opinion, one of the easiest to read, understand and debug. Some of the style rules followed in these labs includes:

- One statement per line, with minor exceptions.
- Significant variables are named with long names starting with nouns and followed by modifiers, and with beginnings of modifiers capitalized.
- Loop counters and other insignificant variables are short, often a single letter such as `k`, `m` or `n`. `i` and `j` are avoided as variable names.
- The interior blocks of loops and if-tests are indented.

- Function files always have comments following the `function` statement so that they are available using the `help` command. Function usage is indicated by repeating the signature line among the comments.

These rules of style improve clarity of the code so that people and not just computers can make sense of it.

Clarity is important for debugging because if code is harder to read then it is harder to debug. Debugging is one of the most difficult and least rewarding activities you will engage in, so anything that simplifies debugging pays off. Clarity is also important when others read your code. Since I need to read your code in order to give you a grade, please format your code similarly to that in the labs.

3 A Sample Problem

Suppose we want to know if there is a solution to

$$\cos x = x \ ,$$

whether the solution is unique, and what its value is. This is not an algebraic equation, and there is little hope of forming an explicit expression for the solution.

Since we can't solve this equation exactly, it's worth knowing whether there is anything we can say about it. In fact, if there is a solution x_0 to the equation, we can *probably* find a computer representable number `x` which approximately satisfies the equation (has low residual error) and which is close to x_0 (has a low approximation error). Each of the following two exercises uses a plot to illustrate the existance of a solution. **Note:** Matlab has the capability of plotting some functions by name. We won't be using this capability in this course because it is too specialized. Instead, we will construct plots the way you probably learned when you first learned about plotting: by constructing a pair of vectors of x-values (abscissæ) and corresponding y-values (ordinates) and then plotting the points with lines connecting them.

Exercise 1: The following steps show how to use Matlab to plot the functions $y = \cos x$ and $y = x$ together in the same graph from $-\pi$ to π.

(a) Define a Matlab (vector) variable `xplot` to take on 100 evenly-spaced values between `-pi` and `pi`. You can use `linspace` to do this.

(b) Define the (vector) variable `y1plot` by `y1plot=cos(xplot)`. This is for the curve $y = \cos x$.

(c) Define the (vector) variable `y2plot` by `y2plot=xplot`. This is for the line $y = x$.

(d) Plot both `y1plot` and `y2plot` by plotting the first,

```
plot(xplot,y1plot)
```

then `hold on`, plot the second, and `hold off`. You saw this done before in Lab 2. **Note**: If you read the help files for the "plot" function, you will find that an alternative *single* command is `plot(xplot,y1plot,xplot,y2plot)`.

(e) The two lines intersect, clearly showing that a solution to the equation $\cos x = x$ exists. Read an approximate value of x at the intersection point from the plot.

(f) Include the Matlab commands you used in your summary file. Send me the plot in the form of a `jpg` file. To create such a file, use the command

```
print -djpeg exer1.jpg
```

or use the "File" menu on the plot window, "save" and choose JPEG image.

Exercise 2:

(a) Write an m-file named `cosmx.m` ("COS Minus X") that defines the function

$$f(x) = \cos x - x$$

Recall that a function m-file is one that starts with the word `function`. In this case it should start off as

```
function y = cosmx ( x )
% y = cosmx(x) computes the difference y=cos(x)-x

% your name and the date

y = ???
```

(This is very simple–the object of this exercise is to get going.) Include a copy of this file with your summary.

(b) Now use your `cosmx` function to compute `cosmx(0.5)`. Your result should be about 0.37758 and should not result in extraneous printed or plotted information.

(c) Now use your `cosmx` function in the same plot sequence

```
plot(???
hold on
plot(???
hold off
```

as above to plot `cosmx` and a line representing the horizontal axis on the interval $-\pi \leq x \leq \pi$. Send me the plot file as `exer2.jpg`.

Hint: The horizontal axis is the line $y = 0$. To plot this as a function, you need a vector of at least two x-values between $-\pi$ and π and a corresponding vector of y-values all of which are zero. Think of a way to generate an appropriate set of x-values and y-values when all the y-values are zero. You could use the Matlab function `zeros`.

(d) Does the value of x at the point where the curve crosses the x-axis agree with the previous exercise?

4 The Bisection Idea

The idea of the bisection method is very simple. We assume that we are given two values $x = a$ and $x = b$ ($a < b$), and that the function $f(x)$ is positive at one of these values (say at a) and negative at the other. (When this occurs, we say that $[a, b]$ is a "change-of-sign interval" for the function.) Assuming f is continuous, we know that there must be at least one root in the interval.

Intuitively speaking, if you divide a change-of-sign interval in half, one or the other half must end up being a change-of-sign interval, and it is only half as long. Keep dividing in half until the change-of-sign interval is so small that any point in it is within the specified tolerance.

More precisely, consider the point $x = (a + b)/2$. If $f(x) = 0$ we are done. (This is pretty unlikely.) Otherwise, depending on the sign of $f(x)$, we know that the root lies in $[a, x]$ or $[x, b]$. In any case, our change-of-sign interval is now half as large as before. Repeat this process with the new change of sign interval until the interval is sufficiently small and declare victory.

We are *guaranteed* to converge. We can even compute the maximum number of steps this could take, because if the original change-of-sign interval has length ℓ then after one bisection the current change-of-sign interval is of length $\ell/2$, *etc.* We know in advance how well we will approximate the root x_0. These are very powerful facts, which make bisection a *robust* algorithm—that is, it is very hard to defeat it.

Exercise 3:

(a) If we know the start points a and b and the interval size tolerance ϵ, we can predict beforehand the number of steps required to reach the specified accuracy. The bisection method will always find the root in that number or fewer steps. What is the formula for that number?

(b) Give an example of a continuous function that has only one root in the interior of the interval [-2, 1], but for which bisection could not be used.

5 Programming Bisection

Before we look at a sample bisection program, let's discuss some programming issues. If you haven't done much programming before, this is a good time to try to understand the logic behind how we choose variables to set up, what names to give them, and how to control the logic.

First of all, we should write the bisection algorithm as a Matlab `function`. There are two reasons for writing it as a function instead of a script m-file (without the function header) or by typing everything at the command line. The first reason is a practical one: we will be executing the algorithm several times, with differing end points, and functions allow us to use temporary variables without worrying about whether or not they have already been used before. The second reason is pedantic: you should learn how to do this now because it is an important tool in your toolbox.

A precise description of the bisection algorithm is presented by Quarteroni, Sacco, and Saleri in Section 6.2.1, on pages 250-251. The algorithm can be expressed in the following way. (Note: the product $f(a) \cdot f(b)$ is negative if and only if $f(a)$ and $f(b)$ are of opposite sign and neither is zero.)
Given a function $f : \mathbb{R} \to \mathbb{R}$ and $a, b \in \mathbb{R}$ so that $f(a) \cdot f(b) < 0$, then sequences $a^{(k)}$ and $b^{(k)}$ for $k = 0, \ldots$ with $f(a^{(k)}) \cdot f(b^{(k)}) < 0$ for each k can be constructed by starting out with $a^{(1)} = a$, $b^{(1)} = b$ then, for each $k > 0$,

1. Set $x^{(k)} = (a^{(k)} + b^{(k)})/2$.
2. If $|x^k - b^{(k)}|$ is small enough, or if $f(x^{(k)}) = 0$ exit the algorithm.
3. If $f(x^{(k)}) \cdot f(a^{(k)}) < 0$, then set $a^{(k+1)} = a^{(k)}$ and $b^{(k+1)} = x^{(k)}$.
4. If $f(x^{(k)}) \cdot f(b^{(k)}) \geq 0$, then set $a^{(k+1)} = x^{(k)}$ and $b^{(k+1)} = b^{(k)}$.
5. Return to step 1.

The bisection algorithm can be described in a manner more appropriate for computer implementation in the following way. Any algorithm intended for a computer program *must* address the issue of when to stop the iteration. In the following algorithm,
Algorithm Bisect(f,a,b,x,ϵ)

1. Set x:=$(a + b)/2$.
2. If $|b - x| \leq \epsilon$, or if $f(x)$ happens to be exactly zero, then accept x as the approximate root, and exit.
3. If `sign(`$f(b)$`) * sign(`$f(x)$`) < 0`, then a:=x; otherwise, b:=x.
4. Return to step 1.

Remarks:

- The latter form of the algorithm is expressed in a form that is easily translated into a loop, with an explicit exit condition.
- The latter form of the algorithm employs a signum function rather than the value of f in order to determine sign. This is a better strategy because of roundoff errors. If $f(a) > 0$ and $f(b) > 0$ are each less than $\sqrt{\texttt{realmin}}$ in Matlab, then their product is zero, not positive.

The language of this description, an example of "pseudocode," is based on a computer language called Algol but is intended merely to be a clear means of specifying algorithms in books and papers. (The term "pseudocode" is used for any combination of a computer coding language with natural language.) One objective of this lab is to show how to rewrite algorithms like it in the Matlab language. As a starting point, let's fix $f(x)$ to be the function `cosmx` that you just wrote. Later you will learn how to add it to the calling sequence. Let's also fix $\epsilon = 10^{-10}$.

Here is a Matlab function that carries out the bisection algorithm for our `cosmx` function. In Matlab, returned values are separated from arguments, so the variables `x` and `itCount` are written on the left of an equal sign.

```
function [x,itCount] = bisect_cosmx( a, b)
% [x,itCount] = bisect_cosmx( a, b) uses bisection to find a
% root of cosmx between a and b to tolerance of 1.0e-10
% a=left end point of interval
% b=right end point of interval
% cosmx(a) and cosmx(b) should be of opposite signs
% x is the approximate root found
% itCount is the number of iterations required.

% your name and the date

EPSILON = 1.0e-10;
fa = cosmx(a);
fb = cosmx(b);

for itCount = 1:(???)  % fill in using the formula from Exercise 3

  x = (b+a)/2;
  fx = cosmx(x);

  % The following statement prints the progress of the algorithm
  disp(strcat(  'a=' , num2str(a), ', fa=' , num2str(fa), ...
              ', x=' , num2str(x), ', fx=' , num2str(fx), ...
              ', b=' , num2str(b), ', fb=' , num2str(fb)))

  if ( fx == 0 )
    return;  % found the solution exactly!
  elseif ( abs ( b - x ) < EPSILON )
    return;  % satisfied the convergence criterion
  end

  if ( sign(fa) * sign(fx) <= 0 )
    b = x;
    fb = fx;
  else
    a = x;
    fa = fx;
  end

end
```

```
error('bisect_cosmx failed with too many iterations!')
```

Note: I have violated my own rule about having functions print things, but the `disp` statement merely prints progress of the function and will be discarded once you are confident the code is correct.
Remarks about the code: Compare the code and the algorithm. The code is similar to the algorithm, but there are some significant differences. For example, the algorithm generates sequences of endpoints $\{a_i\}$ and $\{b_i\}$ and midpoints $\{x_i\}$. The code keeps track of only the most recent endpoint and midpoint values. This is a typical strategy in writing code—use variables to represent only the most recent values of a sequence when the full sequence does not need to be retained.

A second difference is in the looping. There is an error exit in the case that the convergence criterion is never satisfied. If the function ever exits with this error, it is because either your estimate of the maximum number of iterations is too small or because there is a bug in your code.

Note the symbol `==` to represent equality in a test. Use of a single `=` sign causes a syntax error in Matlab. In languages such as C no syntax error is produced but the statement is wrong nonetheless and can cause serious errors that are very hard to find.

Exercise 4: Create the file `bisect_cosmx.m` by typing it in yourself, or by using copy-and-paste to copy it from the web browser window.

(a) Replace the "`???`" with your result from Exercise 3. This value should be an integer, so be sure to increase the value from Exercise 3 to the next larger integer. (If you are not confident of your result, use 10000. If the code is correctly written, it will work correctly for any value larger than your result from Exercise 3.)
Hint: Recall that $\log_2(x) = \log(x)/\log(2)$.

(b) Why is the name `EPSILON` all capitalized?

(c) The variable `EPSILON` and the reserved Matlab variable `eps` have similar-sounding names. As `EPSILON` is used in this function, is it related to `eps`? If so, how?

(d) In your own words, what does the `sign(x)` function do? What if `x` is 0? The `sign` function can avoid representation difficulties when `fa` and `fm` are near the largest or smallest numbers that can be represented on the computer.

(e) The `disp` command is used only to monitor progress. Note the use of `...` as the continuation character.

(f) What is the result if the Matlab `error` function is called?

(g) Based on your understanding of the code, what would the final value of `itCount` be if `a` and `b` are already closer together than the tolerance when the function starts?

(h) Try the command:

```
[z,iterations] = bisect_cosmx ( 0, 3 )
```

Is the value `z` close to a root of the equation `cos(z)=z`?
Warning: You should not see the error message! If you do, your value for the maximum number of iterations is too small.

(i) How many iterations were required? Is this value smaller than the value from your formula in Exercise 3?

(j) Type the command `help bisect_cosmx` at the Matlab command line. You should see your comments reproduced as a help message. This is one reason for putting comments at the top.

(k) For what input numbers will this program produce an incorrect answer (*i.e.*, return a value that is not close to a root)? Add a check before the loop so that the function will call the Matlab `error` function instead of returning an incorrect answer.

(l) In your opinion, why did I use `abs` in the convergence test?

6 Variable Function Names

Now we have written an m-file called `bisect_cosmx.m` that can find roots of the function `cosmx` on any interval, but we really want to use it on an arbitrary function. To do this, we *could* create an m-file for the other function, but we would have to call that m-file `cosmx.m` even if there were no cosines in it, because that is what the `bisect_cosmx` function expects. Suppose that we have three files, called `f1.m` through `f3.m`, each of which evaluates a function that we want to use bisection on. We *could* rename `f1.m` to `cosmx.m`, and change its name inside the file as well, and repeat this for the other files. But who wants to do that?

It is more convenient to introduce a dummy variable for the function name, just as we used the dummy variables `a` and `b` for numerical values. In Matlab, function names can be passed as "function handles."

In the bisection function given below, the name of the function is specified in the function call using the `@` character as:

```
[x, itCount] = bisect ( @cosmx , a, b )
```

Warning: If you forget the `@` in front of the function name, Matlab will produce errors that seem to make no sense at all (even less than usual). Keep in mind that function names in calling sequences *must* have the `@` character in front of them. Check for this first when you get errors in a function.
Remark: Quarteroni, Sacco, and Saleri have an m-file named `bisect.m` on p. 252. Their function includes a variable named `fun` that contains a string representing the function whose root is to be found. They find function values using `eval`. In contrast, `bisect.m` presented above uses a variable named `func` to contain a function handle to evaluate the function. Using function handles is more flexible, allowing much more complicated functions, and is similar to the way that other programming languages (Fortran, C, C++, Java, *etc.*) work.

Exercise 5: Copy the file `bisect_cosmx.m` to a new file named `bisect.m`, or use the "Save as" menu choice to make a copy with the new name `bisect.m`.

(a) In the declaration line for `bisect`, include a dummy function name. We might as well call it `func`, so the line becomes

```
function [x, itCount] = bisect ( func, a, b )
```

(b) Change the comments describing the purpose of the function, and explain the variable `func`.

(c) Where we evaluated the function by writing `cosmx(x)`, you now write `func(x)`. For instance, the line

```
fa = cosmx(a);
```

must be rewritten as:

```
fa = func( a );
```

Make this change for `fa`, and a similar change for `fb` and `fx`.

(d) When we use `bisect`, we must include the function name preceeded by `@`

```
[z, iterations] = bisect ( @cosmx, 0, 3 )
```

Execute this command and make sure that you get the same result as before. (This is called "regression testing.")

Warning: If you forget the `@` before `cosmx` and use the command

```
[z, iterations] = bisect ( cosmx, 0, 3 )   % error!
```

you will get the following mysterious error message:

```
>> [z, iterations] = bisect ( cosmx, 0, 3 )
Not enough input arguments.

Error in cosmx (line 6)
y = cos(x)-x;
```

(e) It is always a good idea to check that your answers are correct. Consider the simple problem $f_0(x) = 1 - x$. Write a simple function m-file named `f0.m` (based on `cosmx.m` but defining the function f_0) and show that you get the correct solution (`x=1` to within `EPSILON`) starting from the change-of-sign interval `[0,3]`.

Remark for advanced users: Matlab allows a simple function to be defined without a name or an m-file. You can then use it anywhere a function handle could be used. For example, for the function $f_0(x) = 1 - x$, you could write

```
[z, iterations] = bisect ( @(x) 1-x, 0, 3 );
```

or even as

```
f0=@(x) 1-x;
[z, iterations] = bisect ( f0, 0, 3 );
```

Be very careful when using this last form because there is no `@` appearing in the `bisect` call but it is a function handle nonetheless.

Exercise 6: The following table contains formulas and search intervals for four functions. Write four simple function m-files, `f1.m`, `f2.m`, `f3`.m and `f4.m`, each one similar to `cosmx.m` but with formulas from the table. Then use `bisect` to find a root in the given interval, and fill in the table. Since you should be confident that your code is correct, turn the `disp` statement into a comment so that you do not get distracting printed lines. I have formatted the table as ordinary text so that you can easily use copy-and-paste to include it in your summary file.

Name	Formula	Interval	approxRoot	No. Steps
f1	x^2-9	[0,5]	_____	_____
f2	x^5-x-1	[1,2]	_____	_____
f3	x*exp(-x)	[-1,2]	_____	_____
f4	2*cos(3*x)-exp(x)	[0,6]	_____	_____
f5	(x-1)^5	[0,3]	_____	_____

7 Convergence criteria

You found above that the bisection algorithm admits an excellent stopping criterion, and the number of iterations can be predicted before beginning the algorithm. This feature is remarkable and highly unusual. The much more common case is that it takes as much or more ingenuity to design a good stopping criterion as it does to design the algorithm itself. Furthermore, stopping criteria are likely to be problem-dependent.

One stopping criterion that is usually available is to stop when the residual of the function becomes small ($|f(x)| \leq \epsilon$). This criterion is better than nothing, and often serves as a good starting point, but it can have drawbacks.

Exercise 7:

(a) Copy your `bisect.m` file to one named `bisect0.m`, or use "Save as" from the "File" menu. Be sure to change the name of the function inside the file. Change the line

```
elseif ( abs ( b - x ) < EPSILON )
```

to read

```
elseif ( abs ( fx ) < EPSILON )
```

Since there is no longer a theoretical expression for the maximum number of interations, you should also increase the maximum number of iterations to 10000.

(b) Consider the function `f0=x-1` on the interval `[0,3]`. Does your new function `bisect0` find the correct answer? (It should.)

(c) The amount the residual differs from zero is called the *residual error* and the difference between the solution you found and the true solution (found by hand or some other way) is called the *true error*. What are the residual and true errors when you used `bisect0` to find the root of the function `f0` above? How many iterations did it take?

(d) What are the residual and true errors when using `bisect0` to find the root of the function `f5=(x-1)^5`? How many iterations did it take?

(e) To summarize your comparison, fill in the following table

Name	Formula	Interval	residual error	true error	number of steps
f0	x-1	[0,3]	_____	_____	_____
f5	(x-1)^5	[0,3]	_____	_____	_____

(f) The `bisect0` function has the value `EPSILON = 1.0e-10;` in it. Does the table show that the residual error is always smaller than `EPSILON`? What about the true error?

8 The secant method

The secant method is described by Quarteroni, Sacco, and Saleri in Section 6.2.2. Instead of dividing the interval in half, as is done in the bisection method, it regards the function as approximately linear, passing through the two points $x = a$ and $x = b$ and then finds the root of this linear function. The interval $[a, b]$ need no longer be a change-of-sign interval, and the next value of x need no longer lie inside the interval $[a, b]$. In consequence, residual convergence must be used in the algorithm instead of true convergence.

It is easy to see that if a straight line passes through the two points $(a, f(a))$ and $(b, f(b))$, then it crosses the x-axis ($y = 0$) when $x = b - \frac{b-a}{f(b)-f(a)} f(b)$.

The secant method can be described in the following manner.

Algorithm Secant(f,a,b,x,ϵ)

1. Set
$$x = b - \frac{b-a}{f(b)-f(a)} f(b)$$
2. If $|f(x)| \leq \epsilon$, then accept x as the approximate root, and exit.
3. Replace a with b and b with x.
4. Return to step 1.

The secant method is sometimes much faster than bisection, but since it does not maintain an interval inside which the solution must lie, the secant method can fail to converge at all. Unlike bisection, the secant method can be generalized to two or more dimensions, and the generalization is usually called Broyden's method.

Exercise 8:

(a) Write an m-file named `secant.m`, based partly on `bisect.m`, and beginning with the lines

```
function [x,itCount] = secant(func, a, b)
% [x,itCount] = secant(func, a, b)
% more comments explaining what the function does and
% the use of each variable

% your name and the date
```

(b) Because residual error is used for convergence, and because the number of iterations is unknown, choose the values

```
EPSILON = 1.0e-10;
ITMAX = 1000;
```

(c) Comment any "`disp`" statements so they do not provide a distraction.

(d) Complete `secant.m` according to the secant algorithm given above.

(e) Test `secant.m` with the linear function `f0(x)=x-1`. You should observe convergence to the exact root in a single iteration. If you do not, go back and correct your code. Explain why it should only take a single iteration.

(f) Repeat the experiments done with bisection above, using the secant method, and fill in the following table.

Name	Formula	Interval	Secant approxRoot	Secant No. Steps	Bisection approxRoot	Bisection No. Steps
f1	x^2-9	[0,5]	_____	_____	_____	_____
f2	x^5-x-1	[1,2]	_____	_____	_____	_____
f3	x*exp(-x)	[-1,2]	_____	_____	_____	_____
f4	2*cos(3*x)-exp(x)	[0,6]	_____	_____	_____	_____
f5	(x-1)^5	[0,3]	_____	_____	_____	_____
f3	x*exp(-x)	[-1.5,1]	_____	_____	_____	_____

(g) In the above table, you can see that the secant method can be either faster or slower than bisection. You may also observe convergence failures: either convergence to a value that is not near a root or convergence to a value substantially less accurate than expected. Regarding the bisection roots as accurate, are there any examples of convergence to a value that is not near a root in the table? If so, which? Are there any examples of inaccurate roots? If so, which?

9 The Regula Falsi method

The regula falsi algorithm can be described in the following manner, reminiscent of the bisection and secant algorithms given above. Since the interval $[a, b]$ is supposed to begin as a change-of-sign interval, convergence is guaranteed.

Algorithm Regula(f,a,b,x,ϵ)

1. Set
$$x = b - \frac{b-a}{f(b)-f(a)} f(b)$$
2. If $|f(x)| \leq \epsilon$, then accept x as the approximate root, and exit.

3. If `sign(`$f(a)$`) * sign(`$f(x)$`)` ≥ 0, then set $a = b$, to keep the change-of-sign interval.

4. Set $b = x$.

5. Return to step 1.

Remark: It is critical to observe that Step **??** maintains the interval $[a, b]$ as a change-of-sign interval that is a subinterval of the initial change-of-sign interval. Thus, regula falsi, unlike the secant method, *must* converge, although convergence might take a long time.

Exercise 9:

(a) Starting from your `bisect0.m` file, write an m-file named `regula.m` to carry out the regula falsi algorithm.

(b) Test your work by finding the root of `f0(x)=x-1` on the interval `[-1,2]`. You should get the exact answer in a single iteration.

(c) Repeat the experiments from the previous exercise but using `regula.m` and fill in the following table.

Name	Formula	Interval	approxRoot	Regula No. Steps	Secant No. Steps	Bisection No. Steps
f1	x^2-9	[0,5]	_____	_____	_____	_____
f2	x^5-x-1	[1,2]	_____	_____	_____	_____
f3	x*exp(-x)	[-1,2]	_____	_____	_____	_____
f4	2*cos(3*x)-exp(x)	[0,6]	_____	_____	_____	_____
f3	x*exp(-x)	[-1.5,1]	_____	_____	_____	_____

You should observe both faster and slower convergence, compared with bisection and secant. You should not observe lack of convergence or convergence to an incorrect solution.

(d) Function `f5`, `(x-1)^5` turns out to be very difficult for regula falsi! Loosen your convergence criterion to a tolerance of 10^{-6} and increase the maximum allowable number of iterations to 500,000 and fill in the following line of the table. (Be sure that any "`disp`" statements are commented out.)

Name	Formula	Interval	approxRoot	Regula No. Steps
f5	(x-1)^5	[0,3]	_____	_____

You should observe convergence, but it takes a very large number of iterations.

(e) `regula.m` and `bisect.m` both keep the current iterate, `x` in a change-of-sign interval. Why would it be wrong to use the same convergence criterion in `regula.m` as was used in `bisect.m`?

10 Extra Credit: Muller's method (8 points)

In the secant method, the function is approximated by its secant (the linear function passing through the two endpoints) and then the next iterate is taken to be the root of this linear function. Muller's method goes one better and approximates the function by the *quadratic* function passing through the two endpoints and the current iterate. The next iterate is then taken to be the root of this quadratic that is closest to the current iterate.

In more detail, suppose I want a function with signature similar to our functions above:

```
[result, itCount] = muller( func, a, b)
```

1. Choose a convergence criterion $\epsilon = 10^{-10}$ and a maximum number of iterations `ITMAX = 100`

2. Choose a third point, not quite at the center of the change-of-sign interval `[a,b]`

```
x0 = a;
x2 = b;
x1 = .51*x0 + .49*x2;
```

3. Evaluate `y0`, `y1`, and `y2` as values of `func` at `x0`, `x1`, and `x2`.

4. Determine coefficients `A`, `B`, and `C` of the polynomial passing through the three points `(x0,y0)`, `(x1,y1)` and `(x2,y2)`, $y(x) = A(x - x_2)^2 + B(x - x_2) + C$:

```
A = ( (y0 - y2) * (x1 - x2) - (y1 - y2) * (x0 - x2) ) / ...
      ( (x0 - x2) * ( x1- x2) * (x0 - x1) );
B = ( (y1 - y2) * (x0 - x2)^2 - (y0 - y2) * (x1 - x2)^2 ) / ...
      ( (x0 - x2) * (x1 - x2) * (x0 - x1) );
C = y2;
```

5. If the polynomial has real roots, find them and choose the one closer to `x2`, otherwise choose something reasonable:

```
if A ~= 0

  disc = B*B - 4.0*A*C;
  disc = max( disc, 0.0 );

  q1 = (B + sqrt(disc) );
  q2 = (B - sqrt(disc) );

  if abs(q1) < abs(q2)
    dx = -2.0*C/q2;
  else
    dx = -2.0*C/q1;
  end

elseif B ~= 0
  dx = -C/B;
else
  error(['muller: algorithm broke down at itCount=', num2str(itCount)])
end
```

6. Discard the point `(x0, y0)` and add the new point

```
x0 = x1;
y0 = y1;

x1 = x2;
y1 = y2;

x2 = x1 + dx;
y2 = func(x2);
```

7. Exit the loop and return `result = x2` if the residual (`y2`) is smaller than ϵ.

8. Go back to **??** unless `ITMAX` is exceeded, in which case print an error message.

Exercise 10:

(a) Following the outline above, create a file `muller.m` that carries out Muller's method.

(b) Test `muller.m` on the simple linear function $f_0(x) = x - 1$ starting with the change-of-sign interval `[0,2]`. You should observe convergence in a single iteration. If you do not observe convergence in a single interation, you probably have a bug. Fix it before continuing. Explain why (one sentence, please) it requires only a single iteration.

(c) Test `muller.m` on the function `f1` on the change-of-sign interval `[0, 5]`. Again, you should observe convergence in a single iteration. If you do not, you probably have a bug. Fix it before continuing. Explain why (one sentence, please) it requires only a single iteration.

(d) Repeat the experiments from the previous exercise but using `muller.m` and fill in the following table.

Name	Formula	Interval	approxRoot	Muller No. Steps	Regula No. Steps	Secant No. Steps	Bisection No. Steps
f1	x^2-9	[0,5]	_____	_____	_____	_____	_____
f2	x^5-x-1	[1,2]	_____	_____	_____	_____	_____
f3	x*exp(-x)	[-1,2]	_____	_____	_____	_____	_____
f4	2*cos(3*x)-exp(x)	[0,6]	_____	_____	_____	_____	_____
f5	(x-1)^5	[0,3]	_____	_____	_____	_____	_____
f3	x*exp(-x)	[-.5,.5]	_____	_____			

You should observe that, since the interval at each iteration need not be a change-of-sign interval, it is possible for the method to fail, as with `secant`. It is possible to combine several methods in such a way that the speed of `secant` or `muller` is partially retained but failure is avoided by maintaining a change-of-sign interval at each iteration. One such method is called Brent's method. See, for example, `http://en.wikipedia.org/wiki/Brent%27s_method`

Last change `$Date: 2016/08/25 20:20:28 $`