MATH2070: LAB 4: Newton's method

Introduction	Exercise 1
Stopping Tests	Exercise 2
Failure	Exercise 3
Introduction to Newton's Method	Exercise 4
Writing Matlab code for functions	Exercise 5
Newton's method: Matlab code	Exercise 6
Choice of initial guess	Exercise 7
No root	Exercise 8
Complex functions	Exercise 9
Non-quadratic convergence	Exercise 10
Unpredictable convergence	Exercise 11
Newton's method revisited	Exercise 12
	Exercise 13
	Exercise 14
Extra Credit: Square Root	Extra Credit

1 Introduction

The bisection method is very reliable, but it can be relatively slow, and it does not generalize easily to more than one dimension. The secant and Muller's methods are faster, but still do not generalize easily to multiple dimensions. In this lab we will look at Newton's method for finding roots of functions. Newton's method naturally generalizes to multiple dimensions and can be much faster than bisection. On the negative side, it requires a formula for the derivative as well as the function, and it can easily fail. Nonetheless, it is a workhorse method in numerical analysis.

We will be taking *three* sessions to complete this lab. If you print this lab, you may find the pdf version more appropriate.

2 Stopping Tests

Root finding routines check after each step to see whether the current result is good enough. The tests that are made are called "termination conditions" or "stopping tests". Common tests include:

Residual size $|f(x)| < \epsilon$

Increment size $|x_{\text{new}} - x_{\text{old}}| < \epsilon$

Number of iterations: `itCount > ITMAX`

The size of the residual looks like a good choice because the residual is zero at the solution; however, it turns out to be a poor choice because the residual can be small even if the iterate is far from the true solution. You saw this when you found the root of the function $f_5(x) = (x-1)^5$ in the previous lab. The size of the increment is a reasonable choice because Newton's method (usually) converges quadratically, and when it does the increment is an excellent approximation of the true error. The third stopping criterion, when the number of iterations exceeds a maximum, is a safety measure to assure that the iteration will always terminate in finite time.

It should be emphasized that the stopping criteria are **estimated errors**. In this lab, in class, and in later labs, you will see many other expressions for estimating errors. You should not confuse the estimated

error with the **true error,** $|x_{\text{approx}} - x_{\text{exact}}|$. The true error is not included among the stopping tests because you would need to know the exact solution to use it.

3 Failure

You know that the bisection method is very reliable and rarely fails but always takes a (sometimes large) fixed number of steps. Newton's method works more rapidly a good deal of the time, but does fail. And Newton's method works in more than one dimension. One objective of this lab will be to see how Newton can fail and get a flavor of what might be done about it.

Although we will seem to spend a lot of time looking at failures, you should still expect that Newton's method will converge most of the time. It is just that when it converges there is nothing special to say, but when it fails to converge there is always the interesting questions of, "Why?" and "How can I remedy it?"

4 Introduction to Newton's Method

You will find the definition of Newton's method in Quarteroni, Sacco, and Saleri on pp. 255f, with convergence discussed on pp. 263f and systems of equations on pp. 286ff, and in the lectures, along with convergence theorems and the like. The idea of Newton's method is that, starting from a guessed point x_0, find the equation of the *straight* line that passes through the point $(x_0, f(x_0))$ and has slope $f'(x_0)$. The next iterate, x_1, is simply the root of this linear equation, *i.e.*, the location that the straight line intersects the x-axis.

A "quick and dirty" derivation of the formula can be taken from the definition of the derivative of a function. Assuming you are at the point $x^{(k)}$, (I am using superscripts in parentheses to denote iteration to reduce confusion between iteration number and vector components.)

$$\frac{f(x^{(k)} + \Delta x) - f(x^{(k)})}{\Delta x} \approx f'(x^{(k)}).$$

If f were linear, this approximate equality would be true equality and the next iterate, $x^{(k+1)}$ would be the exact solution, satisfying $f(x^{(k+1)}) = 0$. In other words

$$f(x^{(k+1)}) = f(x^{(k)} + \Delta x) = 0.$$

This yields

$$\frac{-f(x^{(k)})}{\Delta x} \approx f'(x^{(k)}),$$

or

$$\Delta x = x^{(k+1)} - x^{(k)} = -\frac{f(x^{(k)})}{f'(x^{(k)})}. \tag{1}$$

As you know, Newton's method also will work for vectors, so long as the derivative is properly handled. Assume that $\mathbf{x}$ and $\mathbf{f}$ are n-vectors. Then the Jacobian matrix is the matrix J whose elements are

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

(Here, $i = 1$ for the first row of J, $i = 2$ for the second row of J, *etc.*, so that the Matlab matrix subscripts correspond to the usual ones.) Thus, Newton's method for vector functions can be written

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = -J(\mathbf{x}^{(k)})^{-1}\mathbf{f}(\mathbf{x}^{(k)}). \tag{2}$$

Note: The Matlab backslash operator will be used instead of the inverse operator because it is about three times faster.

5 Writing Matlab code for functions

Newton's method requires *both* the function value and its derivative, unlike the bisection method that requires only the function value. You have seen how Matlab functions can return several results (the root and the number of iterations, for example). In this lab, we will use this same method to return both the function value and its derivative. For example, the `f1.m` file from the previous lab can be modified in the following way to return both the value (`y`) and its derivative (`yprime`).

```
function [y,yprime]=f1(x)
% [y,yprime]=f1(x) computes y=x^2-9 and its derivative,
% yprime=2*x

% your name and the date

y=x.^2-9;
yprime=2*x;
```

Remark: Conveniently, this modified version could still be used for the bisection method as well because Matlab returns only the first value (`y`) unless the second is explicitly used.
Remark: Although the syntax `[value,derivative]` appears similar to a vector with two components, it is not that at all! You will see in the next lab cases where `value` is a vector and `derivative` is a matrix.

In the following exercise, you will modify each of the four functions from the previous lab to return both values of both the function and its derivative.

Exercise 1:

(a) Modify each of the four function m-files `f0.m`, `f1.m`, `f2.m`, `f3.m` and `f4.m` from Lab 3 to return both the function value and that of its derivative. Find the formulæ by hand and add them to the mfles as is done above for `f1.m`. The functions are:

```
f0:  y=x-1               yprime=???
f1:  y=x^2-9             yprime=???
f2:  y=x^5-x-1           yprime=???
f3:  y=x*exp(-x)         yprime=???
f4:  y=2*cos(3*x)-exp(x) yprime=???
```

(b) Use the `help f0` command to confirm that your comments include at least the formula for `f0`, and similarly for `f1`, `f2`, `f3` and `f4`.

(c) What are the values of the function and derivative at x=-1 for each of the five functions?

Remark: There are several other programming strategies for computing derivatives.

- You could compute the derivative value in a new m-file with a new name such as `df0`, `df1`, *etc.*
- You could compute the derivative using some sort of divided difference formula. This method is useful for very complicated functions.
- You could use Matlab's symbolic capabilities to symbolically differentiate the functions. This method seems attractive, but it would greatly increase the time spent in function evaluation.
- There are automated ways to discover the formula for a derivative from the m-file or symbolic formula defining the function, and these can be used to generate the m-file for the derivative automatically.

6 Newton's method: Matlab code

In the next exercise, you will get down to the task of writing Newton's method as a function m-file. In this m-file, you will see how to use a *variable* number of arguments in a function to simplify later calls. The size of the error and the maximum number of iterations will be optional arguments. When these arguments are omitted, they take on their default values.

The m-file you are about to write is shorter than `bisect.m` and the instructions take you through it step by step.

Exercise 2:

(a) Open a new, empty m-file named `newton.m`. You can use either the menus or the command `edit newton.m`.

(b) Start off the function with its signature line and follow that with comments. The comments should repeat the signature as a comment, contain a line or two explaining what the function does, explain what each of the parameters mean, and include your name and the date.

```
function [x,numIts]=newton(func,x,maxIts)
% [x,numIts]=newton(func,x,maxIts)
% func is a function handle with signature [y,yprime]=func(x)
% on input, x is the initial guess
% on output, x is the final solution
% EPSILON is ???
% maxIts is ???
% maxIts is an optional argument
% the default value of maxIts is 100
% numIts is ???
% Newton's method is used to find x so that func(x)=0

% your name and the date
```

(You may have to complete this exercise before coming back to explain all of the parameters.)

(c) From a programming standpoint, the iteration should be limited to a fixed (large) number of steps. The reason for this is that a loop that never terminates appears to you as if Matlab remains "busy" forever.

In the following code, the special Matlab variable `nargin` gives a count of *the number of input arguments* that were used in the function call. This allows the argument `maxIts` to be omitted when its default value is satisfactory.

```
if nargin < 3
  maxIts=100;      % default value if maxIts is omitted
end
```

(d) Define the convergence criterion

```
EPSILON = 5.0e-5;
```

(e) Some people like to use `while` loops so that the stopping criterion appears at the beginning of the loop. My opinion is that if you are going to count the iterations you may as well use a `for` statement. Start off the loop with

```
for numIts=1:maxIts
```

(f) Evaluate the function and its derivative at the current point `x`, much as was done in `bisect.m`. Remember that it is customary to indent the statements within a loop. You may choose the variable names as you please. Recall that the syntax for using two values from a single function call is

```
[value,derivative]=func(x);
```

(g) Define a Matlab variable `increment` as the negative ratio of the function value divided by the value of its derivative. This is the right side of Equation (1).

(h) Complete Equation (1) with the statement

```
x = x + increment;
```

(i) Finish the loop and the m-file with the following statements

```
  errorEstimate = abs(increment);

  disp(strcat(num2str(numIts), ' x=', num2str(x), ' error estimate=', ...
    num2str(errorEstimate)));

  if errorEstimate<EPSILON
    return;
  end
end
% if get here, the Newton iteration has failed!
error('newton: maximum number of iterations exceeded.')
```

The `return` statement causes the function to return to its caller before all `maxIts` iterations are complete. If the error estimate is not satisfied within `maxIts` iterations, then the Matlab `error` function will cause the calculation to terminate with a red error message. It is a good idea to include the name of the function as part of the error message so you can find where the error occurred.

The `disp` statement prints out some useful numbers during the iteration. For the first few exercises in this lab, leave this printing in the file, but when the function is being used as part of a calculation, it should not print extraneous information.

(j) Complete the comments at the beginning of the function by replacing the "`???`" symbols, and use the `help newton` command to confirm your comments are correct.

(k) Recall the Newton iteration should take only a single iteration when the function is linear. Test your `newton` function on the linear function `f0` that you wrote in the previous exercise. Start from an initial guess of `x=10`. The default value of `maxIts` (100) is satsifactory, so you can omit it and use the command

```
[approxRoot numIts]=newton(@f0,10)
```

The correct solution, of course, is `x=1` and it should take a single iteration. It actually takes a second iteration in order to recognize that it has converged, so `numIts` should be 2. If either the solution or the number of iterations is wrong, you have a bug: fix your code. **Hint:** The mistake might be in the derivative in `f0.m` or in the m-file `newton.m`.

(l) Test your `newton` function on the quadratic function `f1` that you wrote in the previous exercise. Start from an initial guess of `x=0.1`. The default value of `maxIts` (100) is satsifactory, so you can omit it. The correct solution, of course, is `x=3`. How many iterations are needed?

(m) As you know, the theoretical convergence rate for Newton is quadratic. This means that if you compute the quantity

$$r_2^{(k)} = \frac{|\Delta x^{(k)}|}{|\Delta x^{(k-1)}|^2}$$

(the superscripts indicate iteration number) then $r_2^{(k)}$ should approach a nonzero constant as $k \to \infty$. You have only taken a few iterations, but if you look at the sequence of values of

`errorEstimate` values does it appear the the ratio $r_2^{(k)}$ is approaching a nonzero limit for this case of Newton applied to `f1`? (If not, you have a bug: fix your code. The mistake could be in `f1.m` or in `newton.m`.) Based on the iterations you have, what do you judge is the value of the constant?

(n) What is the true error in your approximate solution, $|x - 3|$? Is it roughly the same size as `EPSILON`?

(o) Try again, starting at `x=-0.1`, you should get `x=-3`.

(p) Fill in the following table.

```
Name Formula           guess  approxRoot No. Steps

 f0  x-1                0.1    _________  _________
 f1  x^2-9              0.1    _________  _________
 f2  x^5-x-1           10      _________  _________
 f3  x*exp(-x)          0.1    _________  _________
 f4  2*cos(3*x)-exp(x)  0.1    _________  _________
 f4  2*cos(3*x)-exp(x)  1.5    _________  _________
```

Remark: The theoretical convergence rate of Newton's method is quadratic. In your work above, you used this fact to help ensure that your code is correct. This rate depends on an accurate computation of the derivative. Even a seemingly minor error in the derivative can yield a method that does not converge very quickly or that diverges. In the above table, none of the cases should have taken as many as 25 iterations. If they did, check your derivative formula. (As a general rule, if you apply Newton's method and it takes hundreds or thousands of iterations but it does converge, check your derivative calculation very carefully. In the overwhelming majority of cases you will find a bug there.)

7 Non-quadratic convergence

The proofs of convergence of Newton's method show that quadratic convergence depends on the ratio $\frac{f''}{2f'}$ being finite at the solution. In most cases, this means that $f' \neq 0$, but it can also mean that both f'' and f' are zero with their ratio remaining finite. When f''/f' is not finite at the desired solution, the convergence rate deteriorates to linear. You will see this illustrated in the following exercises.

Exercise 3:

(a) Write a function m-file for the function `f6=(x-4)^2`, returning both the function and its derivative.

(b) You recall using `newton` to find a root of `f1=x^2-9` starting from `x=0.1`. How many iterations did it take?

(c) Now try finding the root (`x=4`) of `f6` using Newton, again starting at `x=0.1`. How many iterations does it take? Since the exact answer is `x=4`, the true error is `abs(x-4)`. Is the true error larger than `EPSILON` or smaller?

(d) Now try finding the root (`x=4`) of `f7=(x-4)^20` using Newton, again starting at `x=0.1`. For this case, increase `maxIts` to some large number such as 1000 (recall `maxIts` is the optional third argument to `newton`). How many iterations does it take? Is the true error larger than `EPSILON` or smaller? In this case, you see that convergence rate can deteriorate substantially.

(e) Look at the final two iterations and compute the values of the ratios

$$r_1 = \frac{|\Delta x^{(k+1)}|}{|\Delta x^{(k)}|}, \text{ and} \tag{3}$$

$$r_2 = \frac{|\Delta x^{(k+1)}|}{|\Delta x^{(k)}|^2}. \tag{4}$$

In this case, you should notice that r_1 is *not* nearly zero, but instead is a number not far from 1.0, and r_2 has become large. This is characteristic of a linear convergence rate, and not a quadratic convergence rate.

In practice, if you don't know the root, how can you know whether or not the derivative is zero at the root? You can't, really, but you can tell if the convergence rate is deteriorating. If the convergence is quadratic, r_2 will remain bounded and r_1 will approach zero. If the convergence deteriorates to linear, r_2 will become unbounded and r_1 will remain larger than zero.

Exercise 4:

(a) We are now going to add some code to `newton.m`. Because `newton.m` is working code, it is a good idea to keep it around so if we make errors in our changes then we can start over. Make a copy of `newton.m` called `newton0.m`, and change the name of the function inside the file from newton to newton0. If you need to, you can always return to a working version of Newon's method in `newton0.m`. (You should never have two different files containing functions with the same name because Matlab may get "confused" about which of the two to use.)

(b) Just before the beginning of the loop in `newton.m`, insert the following statement

```
increment=1;  % this is an arbitrary value
```

Just before setting the value of `increment` inside the loop, save `increment` in `oldIncrement`:

```
oldIncrement=increment;
```

(c) After the statement `x=x+increment;`, compute `r1` and `r2` according to (3) and (4) respectively.

(d) Comment out the existing `disp` statement and add another `disp` statement to display the values of `r1` and `r2`.

(e) Use your revised `newton` to fill in the following table, using the final values at convergence, starting from `x=1.0`. Enter your estimate of the limiting values of `r1` and `r2` and use the term "unbounded" when you think that the ratio has no bound. As in the previous exercise, use `maxIts=1000` for these tests.

Function	numIts	r1	r2
f1=x^2-9	______	______	______
f6=(x-4)^2	______	______	______
f7=(x-4)^20	______	______	______

This exercise shows that quadratic convergence sometimes fails, usually resulting in a linear convergence rate, and you can estimate the rate. This is not always a bad situation–it *is* converging after all–but now the stopping criterion is not so good. In practice, it is usually as important to know that you are reliably close to the answer as it is to get the answer in the first place.

In the cases of $(x-4)^2$ and $(x-4)^{20}$, you saw that r_1 turned out to be approaching a constant, not merely bounded above and below. If r_1 were exactly a constant, then

$$x^{(\infty)} = x^{(n)} + \sum_{k=n}^{\infty} \Delta x^{(k)}$$

because it converges and the sum collapses. I have denoted the exact root by $x^{(\infty)}$. Because r_1 is constant, for $k > n$, $\Delta x^{(k)} = r_1^{k-n} \Delta x^{(n)}$. Hence,

$$\begin{aligned} x^{(\infty)} &= x^{(n)} + \Delta x^{(n)} \sum_{k=n}^{\infty} r_1^{k-n} \\ &= x^{(n)} + \frac{\Delta x^{(n)}}{1 - r_1} \end{aligned}$$

This equation indicates that a good error estimate would be

$$|x^{(\infty)} - x^{(n)}| \approx \frac{|\Delta x^{(n)}|}{1 - r_1}$$

Exercise 5:

(a) Comment out the `disp` statement displaying r_1 and r_2 in `newton.m` since it will become a distraction when large numbers of iterations are needed.

(b) Conveniently, `r1` either goes to zero or remains bounded. If the sequence converges, `r1` should remain below 1, or at least its average should remain below 1. Replace the if-test for stopping in `newton` to

```
if errorEstimate < EPSILON*(1-r1)
  return;
end
```

Note: This code is mathematically equivalent to `errorEstimate=abs(increment)/(1-r1)`, but I have multiplied through by $(1 - r_1)$ to avoid the problems that occur when $r_1 \geq 1$. Convergence will never be indicated when $r_1 \geq 1$ because `errorEstimate` is non-negative.

(c) Use `newton` to fill in the following table, where you can compute the absolute value of the true error because you can easily guess the exact solution. (Continue using `maxIts=1000` for this exercise.) You have already done these cases using the original convergence criterion in Exercise 3, so you can get those values from that exercise.

Function	start	ex. 3 newton numIts	ex. 3 newton true err	newton numIts	newton true error
f1=x^2-9	0.1	------	------	-------	-------
f6=(x-4)^2	0.1	------	------	-------	-------
f7=(x-4)^20	0.1	------	------	-------	-------

You should see that the modified convergence does not harm quadratic convergence (about the same number of iterations required) and greatly improves the estimated error and stopping criterion in the linear case. A reliable error estimate is almost as important as the correct solution–if you don't know how accurate a solution is, what can you do with it?

Warning: This modification of the stopping criterion is very nice when r_1 settles down to a constant value quickly. In real problems, a great deal of care must be taken because r_1 can cycle among several values, some larger than 1, or it can take a long time to settle down.
Remark: In the rest of this lab, you should continue using `newton.m` with the convergence criterion involving $(1 - r_1)$.

8 Choice of initial guess

The theorems about Newton's method generally start off with the assumption that the initial guess is "close enough" to the solution. Since you don't know the solution when you start, how do you know when it is "close enough?" In one-dimensional problems, the answer is basically that if you stay away from places where the derivative is zero, then any initial guess is OK. More to the point, if you know that the solution lies in some interval and $f'(x) \neq 0$ on that interval, then the Newton iteration will converge to the solution, starting from any point in the interval. When there are zeros of the derivative nearby, Newton's method can display highly erratic behavior and may or may not converge. In the last part of the previous exercise, you saw a case where there are several roots, with zeros of the derivative between them, and moving the initial guess to the *right* moved the chosen root to the *left*.

Exercise 6: In this and the following exercise, you will be interested in the sequence of iterates, not just the final result. Re-enable the `disp` statement displaying the values of the iterates in `newton.m`.

(a) Write a function m-file for the `cosmx` function used in the previous lab ($f(x) = \cos x - x$). Be sure to calculate both the function and its derivative, as we did for `f1`, `f2`, `f3` and `f4`.

(b) Use `newton` to find the root of `cosmx` starting from the initial value `x=0.5`. What is the solution and how many iterations did it take? (If it took more than ten iterations, go back and be sure your formula for the derivative is correct.)

(c) Again, use `newton` to find the root of `cosmx`, but start from the initial value `x=12`. Note that $3\pi < 12 < 4\pi$, so there are several zeros of the derivative between the initial guess and the root. You should observe that it takes the maximum number of iterations and seems not to converge.

(d) Try the same initial value `x=12`, but also use `maxIts=5000`. (To do this, include 5000 in the call: `newton('cosmx',12,5000)`) (This is going to cause a large number of lines of printed information, but you are going to look at some of those lines.) Does it locate a solution in fewer than 5000 iterations? How many iterations does it take? Does it get the same root as before?

(e) Look at the sequence of values of `x` that Newton's method chose when starting from `x=12`. There is no real pattern to the numbers, and it is pure chance that finally put the iteration near the root. Once it is "near enough," of course, it finds the root quickly as you can see from the estimated errors. Is the final estimated error smaller than the square of the immediately preceeding estimated error?

You have just observed a common behavior: that the iterations seem to jump about without any real pattern until, seemingly by chance, an iterate lands inside the circle of convergence and they converge rapidly. This has been described as "wandering around looking for a good initial guess." It is even more striking in multidimensional problems where the chance of eventually landing inside the ball of convergence can be very small.

Another possible behavior is simple divergence to infinity. The following exercise presents a case of divergence to infinity.

Exercise 7: Attempt to find the root of `f3 = x*exp(-x)`, starting from `x=2`. You should find it diverges in a monotone manner, so it is clear that the iterates are unbounded. What are values of iterates 95 through 100? This behavior can be proved, but the proof is not required for this lab.

9 No root

Sometimes people become so confident of their computational tools that they attempt the impossible. What would happen if you attempted to find a root of a function that had no roots? Basically, the same kind of behavior occurs as when there are zeros of the derivative between the initial guess and the root.

Exercise 8:

(a) Write the usual function m-file for `f8=x^2+9`.

(b) Apply `newton` to the function `f8=x^2+9`, starting from `x=0.1`. Describe what happens.

(c) **Intermediate prints will no longer be needed in this lab.** Comment out the `disp` statements in `newton.m`. Leave the `error` statement intact.

10 Complex functions

But the function $x^2 + 9$ *does* have roots! The roots are complex, but Matlab knows how to do complex arithmetic. (Actually the roots are imaginary, but it is all the same to Matlab.) All Matlab needs is to be reminded to use complex arithmetic.
Warning: If you have used the letter `i` as a variable in your Matlab session, its special value as the square root of -1 has been obscured. To eliminate the values you might have given it and return to its special value, use the command `clear i`. It is always safe to use this command, so I always use it just before I start to use complex numbers. For those people who prefer to use `j` for the imaginary unit, Matlab understands that one, too.
Remark: For complex constants, Matlab will accept the syntax `2+3i` to mean the complex number whose real part is 2 and whos imaginary part is 3. It will also accept `2+3*i` to mean the same thing and it is necessary to use the multiplication symbol when the imaginary part is a variable as in `x+y*i`.

Exercise 9:

(a) Apply `newton` to `f8=x^2+9`, starting from the initial guess `x=1+1i`. (Matlab interprets "`1i`" and "`3i`" as $\sqrt{-1}$ and $3\sqrt{-1}$ respectively.) You should get the result `0+3i` and it should take fewer than 10 iterations. Recall that if you call the `newton` function in the form

```
[approxRoot,numIts]=newton(@f8,1+1i)
```

then the root and number of iterations will be printed even though there are no intermediate messages.

(b) Fill in the following table by using `newton` to find a root of `f8` from various initial guesses. The exact root, as you can easily see, is $\pm 3i$, so you can compute the true error (typically a small number), as you did in Exercise 1.

Initial guess	numIts	true error
1+1i	-------	---------
1-1i	-------	---------
10+5i	-------	---------
10+eps*i	-------	---------

(Recall that `eps` is the smallest number you can add to 1.0 and still change it.)

What happened with that last case? The derivative is $f_5'(x) = 2x$ and is not zero near $x = (10+(\texttt{eps})i) \approx 10$. In fact, the derivative is zero only at the origin. The origin is not near the initial guess nor either root. It turns out that the complex plane is just that: a plane. While Newton's method works in two or more dimensions, it is harder to see when it is going to have problems and when not. We will elaborate a bit in a later section and also return to this issue in the next lab.

11 Unpredictable convergence

The earlier, one-dimensional cases presented in this lab might lead you to think that there is some theory relating the initial guess and the final root found using Newton's method. For example, it is natural to expect that Newton's method will converge to the root nearest the initial guess. *This is not true in general!* In the exercise below, you will see that it is not possible, in general, to predict which of several roots will arise starting from a particular initial guess.

Consider the function $f_9(z) = z^3 + 1$, where $z = x + iy$ is a complex number with real part x and imaginary part y. This function has the following three roots.

$$\omega_1 = -1$$
$$\omega_2 = (1 + i\sqrt{3})/2$$
$$\omega_3 = (1 - i\sqrt{3})/2$$

In the following exercise, you will choose a number of regularly-spaced points in the square given by $|x| \leq 2$ and $|y| \leq 2$ and then use your `newton.m` function to solve $f(z) = 0$ many times, each time one of those points as initial guess. You will then construct an image by coloring each starting point in the square according to which of the four roots was found when starting from the given point. The surprise comes in seeing that nearby initial guesses do not necessarily result in the same root at convergence.

Exercise 10:

(a) Be sure your `newton.m` file does not have any active `disp` statements in it.

(b) Create a function m-file named `f9.m` that computes the function $f_9(z)$ above as well as its derivative.

(c) Copy the following code to a script m-file named `exer4_10.m`.

```
NPTS=100;
clear i
clear which
x=linspace(-2,2,NPTS);
y=linspace(-2,2,NPTS);
omega(1)= -1;
omega(2)= (1+sqrt(3)*i)/2;
omega(3)= (1-sqrt(3)*i)/2;

close  %causes current plot to close, if there is one
hold on
for k=1:NPTS
  for j=1:NPTS
    z=x(k)+i*y(j);
    plot(z,'.k')
  end
end
hold off
```

Be sure to add comments that identify the purpose of the file and your name.

(d) Execute this m-file to generate a plot of 10000 black points that fill up a square in the complex plane. Please include a copy of this plot with your summary.

(e) What is the purpose of the statement `clear i`?

(f) What would happen if the two statements `hold on` and `hold off` were to be omitted?

(g) Next, discard the two `hold` statements and replace the `plot` statement with the following statements:

```
root=newton(@f9,z,500);
[difference,whichRoot]=min(abs(root-omega));
if difference>1.e-2
```

```
  whichRoot=0;
end
which(k,j)=whichRoot;
```

(h) If the variable `root` happened to take on the value `root=0.50-i*0.86`, what would the values of `difference` and `whichRoot` be? (Recall that `(sqrt(3)/2)=.866025...`)

(i) If the function `f9` were not properly programmed, it might be possible for the variable `root` to take on the value `root=0.51-i*0.88`. If this happened, what would the values of `difference` and `whichRoot` be?

(j) The array `which` contains integers between 0 and 3 but the value 0 should never occur. Matlab has a "colormap" called "flag" that maps the numbers 1, 2 and 3 into red, white and blue respectively. Add the following instructions after the loop to plot the array `which` and the three roots as black asterisks.

```
imghandle=image(x,y,which');
colormap('flag') % red=1, white=2, blue=3
axis square
set(get(imghandle,'Parent'),'YDir','normal')

% plot the three roots as blacK asterisks
hold on
plot(real(omega),imag(omega),'k*')
hold off
```

Remark: The `image` command does not plot the array `which` in the usual way. In order to get the picture to display in the usual way, I have plotted the transpose of `which` and also used "handle graphics" to restore the y-axis to its usual orientation. For more information about "handle graphics" search for "graphics object identification" in the help documentation or try this link on the MathWorks web site.

(k) Execute your modified m-file. The position (`z=x+i*y`) on this image corresponds to the initial guess and the color values 1 (red), 2 (white) and 3 (blue) correspond to roots 1, 2 and 3 to which the Newton iteration converged starting from that initial guess. This image illustrates that, for example, some initial guesses with large positive real parts converge to the root $\omega_1 = -1$ despite being closer to both of the other roots. Please include this plot with your summary file.

It turns out that the set you have plotted is a Julia set (fractal). You can increase the detail of the plot by increasing `NPTS` and waiting longer. If you wish to learn more about "Newton Fractals," I suggest a Wikipedia article `http://en.wikipedia.org/wiki/Newton_fractal` or a very comprehensive paper by J. H. Hubbard, D. Schleicher, S. Sutherland. "How to Find All Roots of Complex Polynomials by Newton's Method," Inventiones Mathematicæ **146** (2001)1-33. In addition, the chatty but informative article `http://www.chiark.greenend.org.uk/%7esgtatham/newton` contains interesting information and plots.

12 Newton's method revisited

One disadvantage of Newton's method is that we have to supply not only the function, but also a derivative. In the following exercise, we will try to make life a little easier by numerically approximating the derivative of the function instead of finding its formula. One way would be to consider a nearby point, evaluate the function there, and then approximate the derivative by

$$y' \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{5}$$

There are many possible ways to pick step sizes Δx that change as the iteration proceeds. One way is considered in Exercise 13 below, but we will look at simply choosing a fixed value in the following two exercises.

Exercise 11: Make a copy of `newton.m` and call it `newtonfd.m`. The convergence criterion should involve the factor `(1-r1)` in `newtonfd.m` because using an approximate derivative usually leads to slower linear convergence.

(a) Change the signature (first line) to be

```
function [x,numIts]=newtonfd(f,x,dx,maxIts)
```

and change the comment lines to reflect the new function.

(b) Since `maxIts` is now the fourth variable, change the test involving `nargin` to indicate the new number of variables.

(c) Replace the evaluation of the variable `derivative` through a function call with the expression described in Equation (5) using the constant stepsize `dx`.

(d) Check out your code using the function `f1(x)=x^2-9`, starting from the point `x = 0.1`, using an increment `dx=0.00001`. You should observe essentially the same converged value in the same number (± 1) of iterations as using `newton`.

It turns out that the function `f1(x)=x^2-9` is an "easy" function to solve using Newton iterations. More "difficult" functions require a more delicate choice for `dx`.

Exercise 12: For the two functions `f2(x)=x^5-x-1` and `f6(x)=(x-4)^2`, and the starting point `x = 5.0`, compute the number of steps taken to converge using the true Newton method, and `newtonfd` with different stepsizes `dx`. You will have to increase the value of `maxIts` by a very great deal to get `f6` to converge for the cases of large `dx`.

	Number of steps f2	Number of steps f6
using newton	_____	_____
dx = 0.00001	_____	_____
dx = 0.0001	_____	_____
dx = 0.001	_____	_____
dx = 0.01	_____	_____
dx = 0.1	_____	_____
dx = 0.5	_____	_____

As you can see, the choice of `dx` is critical. It is also quite difficult in more complicated problems.

Remark: As mentioned earlier, Newton's method generally converges quite quickly. If you write a Newton solver and observe poor or no convergence, the first thing you should look for is an error in the derivative. One way of finding such an error is to apply a method like `newtonfd` and print both the estimated derivative and the analytical derivative computed inside your function. They should be close. This trick is especially useful when you are using Newton's method in many dimensions, where the Jacobian can have some correctly-computed components and some components with errors in them.

You saw the secant method in Lab 3, and it had the update formula

$$x = b - \left(\frac{b-a}{f(b)-f(a)} \right) f(b). \tag{6}$$

Exercise 13:

(a) Compare (6) with (1) and show that by properly identifying the variables x, a, and b then (6) can be interpreted as a Newton update using an approximation for $f'(x^{(k)})$. Explain your reasoning.

(b) Starting from your `newton.m`, copy it to a new m-file named `secant4.m` and modify it to carry out the secant method. The secant method requires an extra value for the iterations to start, but there is only the value `x` in the signature line. Take `b=x` and `a=x-0.1` *only for the first iteration.* Do not use `secant.m` from the previous lab because it uses a different convergence criterion. Fill in the following table.

Function	start	Newton numIts	Newton solution	secant4 numIts	secant4 solution
f1=x^2-9	0.1	------	------	------	------
f3=x*exp(-x)	0.1	------	------	------	------
f6=(x-4)^2	0.1	------	------	------	------
f7=(x-4)^20	0.1	------	------	------	------

You should observe that both methods converge to the same solutions but that the secant method takes more iterations, but not nearly so many as `newtonfd` with a larger *fixed* `dx`. This is because the theoretical convergence rate is quadratic for Newton and $(1+\sqrt{5})/2 \approx 1.62$ for secant because of the approximation to the derivative.

13 Extra Credit: Square Roots (8 points)

Some computer systems compute square roots in hardware using an algorithm often taught in secondary school and similar to long division. Other computer systems leave square roots to be computed in software. Some systems, however, use iterative methods to compute square roots. With these systems, great care must be taken to be sure that a good initial guess is chosen and that the stopping criterion is reliable. You can find a very technical discussion of these issues in a paper by Liang-Kai Wang and Michael J. Schulte, "Decimal Floating-Point Square Root Using Newton-Raphson Iteration," Proceedings of the 16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP 05), `http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.65.7220`

Exercise 14: One well-known iterative method for computing the square root of the number a is

$$x^{(k+1)} = .5\left(x^{(k)} + \frac{a}{x^{(k)}}\right). \tag{7}$$

This can be started with, for example, $x^{(0)} = .5(a+1)$. This initial guess is not optimal but it is reasonable because $\sqrt{a}$ is always between a and 1.

(a) Show that (7) is the result of applying Newton's method to the function $f(x) = x^2 - a$.

(b) In the case $a > 0$, Explain why convergence cannot deteriorate from quadratic. For this case, it makes sense to use a relative error estimate to stop iterating when

$$|x^{(k+1)} - x^{(k)}| \leq 10^{-15}|x^{(k+1)}|.$$

(c) Write a Matlab function m-file named `newton_sqrt.m` to compute the square root using (7). Write this function directly from (7)—do not call your previously-written `newton` function.

(d) Fill in the following table using your `newton_sqrt.m`. (The true error is the difference between the result of your function and the result from the Matlab `sqrt` function.)

Value	square root	true error	no. iterations
a=9	_____	_____	_____
a=1000	_____	_____	_____
a=12345678	_____	_____	_____
a=0.000003	_____	_____	_____

Last change $Date: 2016/08/31 15:08:57 $