University of Colorado
Department of Computer Science

Numerical Computation CSCI 3656

Spring 2016

Problem Set 2 Solutions

1. (a) *[10 pts]* Write a program that finds a root of the equation $f(x) = sin(x) + 0.25$ using the bisection method, starting from initial guesses of $x = -1$ and $x = 1$. Note the number of iterations required to get $|f(x)| < 0.0001$ and observe the pattern in how that error changes at each step (i.e., does it decrease smoothly?)

Here's some Matlab code that does this:

```
% root is the result; errors is a vector containing the sequence of errors
% (the percent is the Matlab comment character)
function [root,errors] = bisection(a,b,tolerance)
% the function:
function retn_val = f(x)
% "function blah=foo(args)" is matlab's syntax for a function named foo to
% return a value called blah.  someplace in the function body, you need to
% put the value that you want to return in the variable blah
   retn_val = sin(x) + 0.25;
end
% the algorithm:
errors=[];  % this sets up an empty vector in which to store these
if f(a)*f(b)>0   % check if guesses are on either side of root
    disp('guesses do not bracket root!')
else
    root = (a + b)/2;  err = abs(f(root));
    while err > tolerance
    if f(a)*f(root)<0
        b = root;
    else
        a = root;
    end
    root = (a + b)/2; err = abs(f(root));
    errors = [errors,err];  % append current error to the errors vector
    end
end
end
```

That code takes 12 iterations to get the root to within 0.0001. The error does *not* decrease monotonically:

```
>> [root,errors]=bisection(-1,1,0.0001);
>> root = -0.2527
>> errors = 0.2294    0.0026    0.1163    0.0574    0.0276    0.0125
                   0.0050    0.0012    0.0007    0.0002    0.0002    0.0000
```

(b) *[7 pts]* What happens to the number of iterations required to get $|f(x)| < 0.0001$—and the pattern in that error as the answer converges—if the initial guesses are changed to $x = -1$ and $x = 0$? Why?

It takes 11 iterations to get the root to within 0.0001:

```
>> [root,errors]=bisection(-1,0,0.0001);
>> errors = 0.0026    0.1163    0.0574    0.0276    0.0125    0.0050
                   0.0012    0.0007    0.0002    0.0002    0.0000
```

Here, too, the error decreases nonmonotonically. (The error after the second step, for instance, is a lot bigger than the error after the first step.) This is because the root is very close to one of the guesses, so the "bisect the interval and take the midpoint as the next guess" actually bounces the iteration sequence *away* from the root before it gets closer.

(c) *[7 pts]* Now mess around with the initial guesses, exploring the interval between $x = -1$ and $x = 7$, and see if you can find some other roots. Turn in the x values of the other roots that you find, along with the values of the initial guesses that you used in each case.

This function has two other roots in the range $x = [-1, 7]$: one at 3.3943 (which `bisection` will find if you give it 3 and 5 as guesses) and and another at 6.0305 (which `bisection` will find if you start it up with 5 and 7 as guesses).

2. *[10 pts]* Write a program that finds a root of the equation $f(x) = sin(x) + 0.25$ using the *secant* method, starting from initial guesses of $x = -1$ and $x = 1$. Note the number of iterations required to get $|f(x)| < 0.0001$.

In writing this, you can recycle most of the bisection function. One big difference is that the guesses do not have to bracket the root, so that test goes away. Another difference is that you always use the same equation to compute the new guess from the last two guesses, so there's no need for the "`if f(a)*f(root)<0 b = root; else a = root;`" stuff. You *do* have to make some sort of (arbitrary) choice about which of the guesses at the first round to use in round two, but after that the iteration sequence is pretty straightforward:

```
function [root,errors] = secant(a,b,tolerance)
% the function:
function retn_val = f(x)
   retn_val = sin(x) + 0.25;
end
% the algorithm:
errors=[];
```

```
root = (a*f(b) - b*f(a))/(f(b) - f(a)); err = abs(f(root));
prev = b;  % arbitrary choice of one
while err > tolerance
   root = (root*f(prev) - prev*f(root))/(f(prev) - f(root));
   err = abs(f(root)); errors = [errors,err];
end
end
```

This method converges to the root **much** more quickly:

```
>> [root,errors]=secant(-1,1,0.0001);
>> errors = 0.0043    0.0005    0.0001
```

3. *[10 pts]* Write a program that finds a root of the equation $f(x) = sin(x) + 0.25$ using Newton's method, starting from an initial guess of $x = -1$. Note the number of iterations required to get $|f(x)| < 0.0001$.

Again, we can recycle a fair bit of that code:

```
function [root,errors] = newton(root,tolerance)
% the function
 function retn_val = f(x)
    retn_val = sin(x) + 0.25;
 end
% the derivative
 function deriv = f_prime(x)
        deriv = cos(x);
 end
% the algorithm
    errors=[];  err = abs(f(root));
    while err > tolerance
    root = root - f(root)/f_prime(root);
    err = abs(f(root)); errors = [errors,err];
    end
end
```

That method takes four iterations to find the root, starting from $x = 1$:

```
>> [root,errors]=newton(1,0.0001);
>> errors = 0.6022    0.3803    0.0002    0.0000
```

4. *[8 pts]* Compare the convergence patterns of the three methods (bisection, secant, Newton) that you observed in the previous problems. Are they consistent with what you know about the theoretical convergence rates and patterns of these two methods? Explain why or why not.

In general, bisection converges linearly, Newton converges quadratically, and secant converges as a 1.62 power—i.e., super-linearly. (That is, the error for bisection is proportional to the error at the previous step. The error for the secant method is proportional to the previous error raised to the 1.62. The error for Newton's method is proportional to the square of the previous error times "M." See Sauer section 1.4.1-1.4.2).

The results make sense with regards to the known convergence rates. For example, observing the error values at each iteration, Newton's method indeed seems to be reducing the error in proportion to the square of the previous error – i.e. $0.6022^2 = 0.3626$, which is close to 0.3803. The same is true for the error in secant where 0.0005 is indeed proportional to 0.0043 to a power between 1 and 2 (approx. 1.4); and 0.0001 is proportional to 0.0005 to a power between 1 and 2.

It makes complete sense that bisection takes more iterations and converges slower than the secant method or Newton's method. The fact that secant beats Newton is less straightforward. The issue here is that the derivations of those convergence rates (linear, quadratic, etc.) do not factor in the shape of the function and the position of the guesses. They are *general* results, and they may not hold for all functions and all guess choices.

5. *[8 pts]* Use the fixed point method and an initial guess of $x = 1$ to find roots of $f(x) = x^3 - 3x^2 - x + 3$. Try a number of rearrangements until you find at least one that converges and at least one that doesn't. Explain this result: what is it about the rearrangement that makes one converge and the other not? (You're going to have to explore guesses near $x = 1$ to answer this completely...)

Answers will vary. Examples of rearrangements that lead to convergence are $g(x) = (2x^2 - 3)/(x^2 - x - 1)$ and $k(x) = \sqrt[3]{3x^2 + x - 3}$. An example of a rearrangement that leads to a failure of the sequence to converge is $h(x) = x^3 - 3x^2 + 3$. In general, this method will lead to convergence on the root when the absolute value of the derivative of the rearrangement is less than one. There was a bit of a trick to this one, since $x = 1$ **is** a root of this function. To know whether a particular arrangement converged or not, you had to play with guesses *near* that value.