Peter Woodman
0127119
Amath 352 - B. Bale
Homework 4

1. Well, the obvious iteration function attained by isolating x, $g(x) = \cos(x)$ converges to 0.7391 with $x_0 = .5$. This iteration function is, in fact, guaranteed to reach convergence because of the range of its derivative- the cosine function. I'm a bit hard-pressed to come up with another working fixed point iteration function, as if we attempt to use inverse trig functions, we run into range problems and get imaginary numbers. So, we can also use Newton's method to come up with a fixed point iteration function, giving us $g(x) = x + \frac{\cos(x)-x}{\sin(x)+1}$. This converges to the same answer, although much, much faster (2 iterations for error $< 10^{-4}$). Matlab code for this is one trivial `for` loop, so it will not be included here.

2. (a) It will fail to produce a meaningful answer, as there is no sign change in this region. If you are asking what the particular implementation in the book will do, it will return very close to `xleft`.

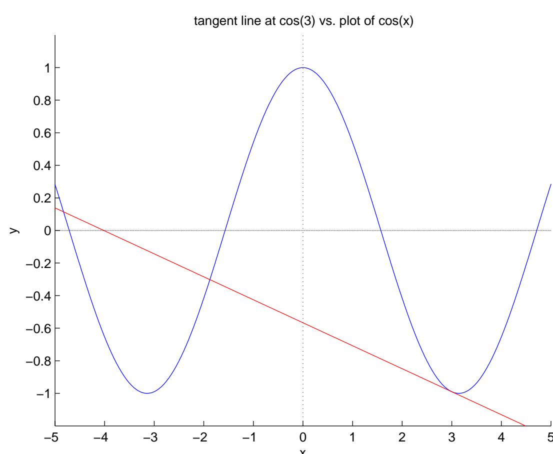
   (b) Here, the bisection method will continually oscillate between values approaching $\pm\infty$, and will stop near one of the two values, depending upon the tolerance and the starting range.

3. (a) Funny, that- we've already done this above, in problem one. Well, nearly.

$$g(x) = x_{k-1} - \frac{f(x_k)}{f'(x_k)} = x_{k-1} + \frac{\cos(x_k)}{\sin(x_k)}$$

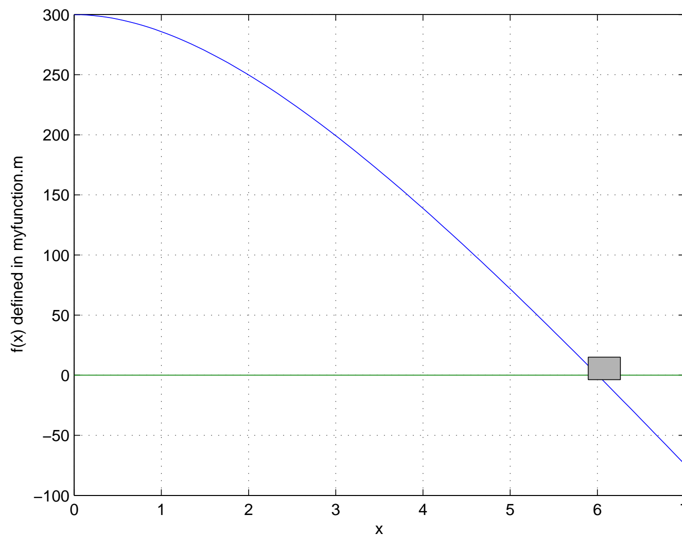
   (b) No, not for this root. One could do some hand waving and say that $x_0 = 3$ is "not close enough" for Newton's method, and this would be more or less correct. A slightly more complete answer would be to state that the initial guess $x_0 = 3$ is outside of the local minima and maxima which border our desired root, $x = \frac{3\pi}{2}$, and furthermore, the tangent line is not sufficiently steep.

   To explain why this condition is necessary for convergence (and also to explain why the ass we get $x = -\frac{3\pi}{2}$ as our root with initial guess $x_0 = 3$), we have to look at how Newton's method works. Newton's method works essentially by using tangent lines as function approximations. The assumption here is that the root of the tangent line is a better guess of the root of the function than the initial guess. This is necessarily not true when outside of the local minima and maxima surrounding the desired root, but also is not true when the tangent line is too shallow, as the next guess will not be closer to the root than the initial guess.



   (c) Yes, we can, for the same reasons we couldn't with $x_0 = 3$. 5 is between the same local minima and maxima as $\frac{3\pi}{2}$, and has a sufficiently steep slope to reach a zero which is closer to the real root than the guess.

4. (a) Modified `myfunction.m` appears at back.



This graph indicates that we'd expect to find a root somewhere between 5.8 and 6.3.

(b)
```
>> bisect('myfunction',[0 7],10^(-6),10^-6,1)

Bisection iterations for myfunction.m
    k       xm      fm
   1     3.5000e+00     1.6999e+02
   2     5.2500e+00     5.4210e+01
   3     6.1250e+00    -8.8910e+00
   4     5.6875e+00     2.2977e+01
   5     5.9062e+00     7.1154e+00
   6     6.0156e+00    -8.7047e-01
   7     5.9609e+00     3.1269e+00
   8     5.9883e+00     1.1293e+00
   9     6.0020e+00     1.2969e-01
  10     6.0088e+00    -3.7033e-01
  11     6.0054e+00    -1.2030e-01
  12     6.0037e+00     4.6955e-03
  13     6.0045e+00    -5.7803e-02
  14     6.0041e+00    -2.6553e-02
  15     6.0039e+00    -1.0929e-02
  16     6.0038e+00    -3.1166e-03
  17     6.0037e+00     7.8944e-04
  18     6.0037e+00    -1.1636e-03
  19     6.0037e+00    -1.8708e-04

ans =

    6.0037
```

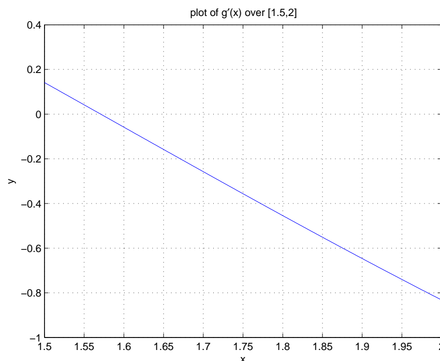
(c) To be somewhat fair, I'm going to set $x_0 = 3.5$ when using Newton's method, as this is the first guess of the bisection method used above.

```
>> newton('fx3n',.1,10^(-6),10^-6,1)

Newton iterations for fx3n.m
  k     f(x)        dfdx   x(k+1)
  1     2.998e+02    -3.154e+00   95.18177700610340
```

```
 2   -7.154e+03   -8.042e+01   6.23018340068387
 3   -1.664e+01   -7.377e+01   6.00468004314774
 4   -6.976e-02   -7.314e+01   6.00372632687388
 5   -1.325e-06   -7.314e+01   6.00372630875878

ans =

    6.0037
```

(d)
```
>> fzero('myfunction',3.5,optimset('TolFun',10^-6,'TolX',10^-6))

ans =

    6.0037
```

5. (a) Well, the convergence therorem states that for any fixed point iteragtion function $g(x)$, iterations upon x_0 will converge as long as $-1 < g'(x) < 1$. In this case $g(x) = 2\sin(x)$ and $g'(x) = 2\cos(x)$. In the region given, $[1.5, 2]$, $g'(x)$ stisfies these conditions, as shown by the graph below.



As we can see, this function will converge monotonically for $1.5 < x_0 < 1.57$, and will oscillate for the rest of our given range.

(b) We can solve the equation given for k-

$$k \approx \frac{\ln(\frac{E_k}{|E_0|})}{\ln|(g'(x^*)|)}$$

For the equation given on the range given, $x^* = 1.8955$. Hence, $E_0 = 2 - 1.8955 = .1045$, and $g'(x^*) = 2\cos(1.8955) = -.6381$. We are given that we wish $|E_k| < 10^{-10}$- I'm going to fib a little bit and set $E_k = 10^{-10}$ for right now.

$$k \approx \frac{\ln(\frac{10^{-10}}{0.1045})}{\ln(.6381)} \approx 46.22$$

This means that to achieve the desired level of accuracy, we must iterate 47 times to reach an appropriate approximation.

(c) See back for code for `fixedpoint.m`.

```
>> fixedpoint('tempf',2,50)

Fixed point iterations for tempf.m
 k   x                       err         ratio
 1   2.00000000000000e+00    1.045e-01                NaN
 2   1.81859485365136e+00    7.690e-02   0.73583918508683
 3   1.93890945306856e+00    4.342e-02   0.56457109521188
 4   1.86601601636051e+00    2.948e-02   0.67898478311246
```

```
  5   1.91347649344167e+00    1.798e-02   0.61001674105013
  6   1.88371495915469e+00    1.178e-02   0.65505280671217
  7   1.90287832191698e+00    7.384e-03   0.62686661717865
  8   1.89073127521091e+00    4.763e-03   0.64503743519604
  9   1.89851175825404e+00    3.017e-03   0.63352853251670
 10   1.89356034515336e+00    1.934e-03   0.64090389650956
 11   1.89672465092032e+00    1.230e-03   0.63621178221753
 12   1.89470779214427e+00    7.865e-04   0.63921097995578
 13   1.89599548716914e+00    5.012e-04   0.63729960322540
 14   1.89517422792746e+00    3.200e-04   0.63852005150592
 15   1.89569836932464e+00    2.041e-04   0.63774172125107
 16   1.89536400109798e+00    1.303e-04   0.63823848121344
 17   1.89557736648665e+00    8.310e-05   0.63792158731698
 18   1.89544123929507e+00    5.303e-05   0.63812380481395
 19   1.89552809845519e+00    3.383e-05   0.63799479110668
 20   1.89547267997845e+00    2.159e-05   0.63807711178773
 21   1.89550804010622e+00    1.377e-05   0.63802458918448
 22   1.89548547901365e+00    8.788e-06   0.63805810162622
 23   1.89549987411364e+00    5.607e-06   0.63803671949945
 24   1.89549068943477e+00    3.578e-06   0.63805036229786
 25   1.89549654969131e+00    2.283e-06   0.63804165771736
 26   1.89549281059084e+00    1.456e-06   0.63804721145819
 27   1.89549519630831e+00    9.293e-07   0.63804366828165
 28   1.89549367411428e+00    5.929e-07   0.63804592869922
 29   1.89549464534313e+00    3.783e-07   0.63804448683958
 30   1.89549402565557e+00    2.414e-07   0.63804540581952
 31   1.89549442104423e+00    1.540e-07   0.63804482123553
 32   1.89549416876848e+00    9.827e-08   0.63804519214532
 33   1.89549432973179e+00    6.270e-08   0.63804495795145
 34   1.89549422702995e+00    4.000e-08   0.63804510313042
 35   1.89549429255835e+00    2.552e-08   0.63804501793088
 36   1.89549425074828e+00    1.629e-08   0.63804505628872
 37   1.89549427742499e+00    1.039e-08   0.63804506051018
 38   1.89549426040405e+00    6.630e-09   0.63804501259548
 39   1.89549427126418e+00    4.230e-09   0.63804511256036
 40   1.89549426433493e+00    2.699e-09   0.63804497030015
 41   1.89549426875610e+00    1.722e-09   0.63804521473185
 42   1.89549426593519e+00    1.099e-09   0.63804487904757
 43   1.89549426773506e+00    7.011e-10   0.63804537408960
 44   1.89549426658666e+00    4.473e-10   0.63804455097467
 45   1.89549426731939e+00    2.854e-10   0.63804612855316
 46   1.89549426685188e+00    1.821e-10   0.63804408373607
 47   1.89549426715017e+00    1.162e-10   0.63804741721984
 48   1.89549426695985e+00    7.413e-11   0.63804134704689
 49   1.89549426708128e+00    4.730e-11   0.63805099543839
 50   1.89549426700380e+00    3.018e-11   0.63803819216253

ans =

    1.8955
```

This is not exactly what I expected to see. Above, I predicted that it would take 47 iterations to reach the desired level of error. Here, we see it happen in 43 iterations. Upon further inspection, I realize it is because $g'(x^*) \neq g'(x_k)$. This is the source of the error. This makes it somewhat more difficult to compute, as $g'(x_k)$ is slightly different for each x_k.

(d) This is approaching $g'(x^*)$ - as a matter of fact, at each step k, it is very close to $|g'(x_k)|$.

Also, at each step, we notice that $E_{k+1} = E_k \cdot \frac{E_k}{E_{k-1}}$.

(e) With the bisection method, at each step, we are ostensibly cutting the error in half. So, $E_k = E_0 \cdot .5^k$. Solve for k, and we get $k = \frac{\ln(\frac{E_k}{E_0}}{\ln(.5)}$. Plugging in our desired E_k and our known E_0 for $x_0 = 2$, we find that $k \approx 30$. We'd expect 30 steps for convergence within tolerance.

(f) I've added a command switch to invoke Newton's method to `fixedpoint.m`.

```
>> fixedpoint('f2sinx',2,7,1)

Fixed point iterations for f2sinx.m
  k   x                       err         ratio
  1   2.00000000000000e+00    1.045e-01                NaN
  2   1.90099559420391e+00    5.501e-03   0.05264139118298
  3   1.89551164537959e+00    1.738e-05   0.00315893693959
  4   1.89549426720871e+00    1.747e-10   0.00001005460788
  5   1.89549426703398e+00    2.220e-16   0.00000127076913
  6   1.89549426703398e+00    2.220e-16   1.00000000000000
  7   1.89549426703398e+00    2.220e-16   1.00000000000000

ans =

    1.8955
```

As we can see, this converges much, much faster. The ratio is much, much smaller, meaning the error shrinks much, much faster. When it has reached Matlab's maximum precision, the ratio becomes 1, and we see no more improvement.

Code Snippets

-myfunction.m-

```
function f = myfunction(x)
% myfunction evaluates f(x) = x - x^(1/3) -2
m = .25;
g = -32.17;
k = .1;
t = x;
snot = 300;

f = snot + (m*g/k)*t - (m^2*g/k^2)*(1 - exp((-k)*t/m));
```

-fx3n.m-

```
function [f,dfdx] = fx3n(x)
% fx3n: Evaluate f(x) = x - x^(1/3) -2 and dfdx for Newton algorithm.
m = .25;
g = -32.17;
k = .1;
t = x;
snot = 300;

f = snot + (m*g/k)*t - (m^2*g/k^2)*(1 - exp((-k)*t/m));
dfdx = -(m*g/k)*exp((-k)*t/m)+(m*g/k);
```

-fixedpoint.m-

```
function r = fixedpoint(fun,x0,maxit,newton)
% newton  Newton's method to find a root of the scalar equation f(x) = 0.
%
% Input: fun  = (string) name of mfile that returns f(x) and f'(x)
%    x0   = initial guess.
%    xtol = (optional) absolute tolerance on x.  Default: xtol = 5*eps
%    ftol = (optional) absolute tolerance on f.  Default: ftol = 5*eps
%    verbose = (optional) flag.  Default: verbose = 0, no printing.
%
% Output: r = the root of the function.

if nargin < 3, maxit = 50; end
if nargin < 4, newton = 0; end
 fprintf('\nFixed point iterations for %s.m\n',fun);
 fprintf('  k   x                       err         ratio\n');
x = x0;                                       % initial guess
k = 1;                                        % initialize the iteration number
if newton,
  xzero = fzero(fun,x0);                        % proper solving
else
  xzero = fzero(@(tmp) tmp - feval(fun,tmp),x0); % cheat shamelessly
end
ek = abs(x0 - xzero);
ratio = NaN;
if newton,
  [f,dfdx] = feval(fun,x);
  while k <= maxit
    fprintf('%3d   %18.14e %12.3e %18.14f\n',k,x,ek,ratio)
    dx = f/dfdx;
    x = x-dx;
```

```
    ekl = ek;
    ek = abs(xzero - x);
    ratio = ek/ekl;
    k = k+1;
    [f,dfdx] = feval(fun,x);
  end;
else
  while k <= maxit
    fprintf('%3d   %18.14e %12.3e %18.14f\n',k,x,ek,ratio)
    k = k+1;
    x = feval(fun,x);
    ekl = ek;
    ek = abs(xzero - x);
    ratio = ek/ekl;
  end;
end;

r = x;
```