1.2.X2

X2. (a) [3 points] The functional iteration you did in HW1 with \( g(x) = \cos(x) \) did converge to a fixed point of \( g \), i.e. to a root of \( f(x) = \cos(x) - x \).

Evaluate \( g'(r) \) for that case and explain how this is consistent with Theorem 1.6.

(b) [1 point] The equation \( 4x(1-x) = x \) has a positive solution. What is it?

(c) [2 points] If you hadn’t realized you could do (b) analytically, and tried iterating the function \( g(x) = 4x(1-x) \) to try to find the solution numerically, explain why Theorem 1.6 says you will be unsuccessful.

---

1.2.X2

In HW #1, found root \( g(x) = \cos(x) - x \) is \( r \approx 0.78908513215161 \).

With \( g(x) = \cos(x) \),

\[
g'(r) = -\sin r \approx -0.673612021883215
\]

Since \( |g'(r)| < 1 \), Thm 1.6 guarantees (at least local) convergence.

\[ 4 \times (1-x) = x \implies x = 0 \text{ or } 4(1-x) = 1 \\
1-x = \frac{1}{4} \\
x = \frac{3}{4} \]

Answer: positive root is \( r = \frac{3}{4} \).

With \( g(x) = 4x(1-x) = 4x - 4x^2 \),

\[
g'(x) = 4 - 8x,
\]

and so

\[
g'(r) = g'(\frac{3}{4}) = 4 - 8 \cdot \frac{3}{4} = 4 - 6 = -2.
\]

Since \( |g'(r)| = 2 > 1 \)

we know that iteration of \( g \) will not converge to \( r \); in fact locally, the distance from \( r \) will approximately double with each step.
(d) (i) [2 points] Make a cobweb diagram for the function $g_1(x) = 2.5x(1-x)$ starting at $x=0.3$. You may use a modification of my code `cobweb.py`. Since unlike the in-class examples, $g$ is not a built-in function, you will have to set up $g_1$ using the "def" construction (see previous examples).

```python
# cobweb_hw3.py
from numpy import *
from pylab import plot, show, xlabel, ylabel, subplot

g = cos # arccos
def g(x):
    return 2.5*x*(1-x)

lo = -pi
hi = pi
xa = linspace(lo, hi, 2000)
gxa = g(xa)

subplot(111, aspect='equal')
xlabel('x(i)')
ylabel('x(i+1)')

x = 0.3 # 0.74
lw = 2
plot(xa, gxa, linewidth=1w)
plot(xa, xa, 'k', linewidth=1w)

def vertical(x):
    plot([x, x], [x, g(x)], 'r', linewidth=1w)

def across(x):
    plot([x, g(x)], [g(x), g(x)], 'g', linewidth=1w)

plot([x, x], [lo, g(x)], 'r', linewidth=1w) # up to g
for i in range(200):
    if i>8: vertical(x)
    across(x)
    x = g(x)

print x
show()
```
(ii)[2 points] Make a cobweb diagram for the function \( g_2(x) = 4x(1-x) \) starting at \( x=0.3 \).

The iterates are contained in the interval between 0 to 1 but appear not to converge to any fixed point.

Just to satisfy curiosity: a zoomed view of the region near the positive fixed point, showing a "near-hit" and subsequent divergence:
1.4.2

(a) \( f(x) = x^3 + x^2 - 1 \)

\[ g(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x^3 + x^2 - 1}{3x^2 + 2x} \]

\[ = \frac{3x^3 + 2x^2 - x^3 - x^2 + 1}{3x^2 + 2x} = \frac{2x^3 + x^2 + 1}{(3x + 2)x} \]

\( x_0 = 1 \)

\[ x_1 = g(x_0) = g(1) = \frac{2 + 1 + 1}{(3 + 2) \cdot 1} = \frac{4}{5} = 0.8 \]

\( x_2 = g(x_1) = g(0.8) = \frac{0.64(2.6) + 1}{0.8(4.4)} = 0.7568 \]

\[ = \frac{333}{440} \]
1.4.X1

X1. (a) [4 points] Using the code I wrote in class on Thursday, Sep 13, we can see that Newton's method succeeds in finding the root (0) of \( f(x) = \tanh(x) \) when the initial guess \( x_0 \) is \( \pm 1.08 \), but fails when \(|x_0| \geq 1.09\). Try the function \textbf{brentq} from \texttt{scipy.optimize} (Google it),

\texttt{from scipy.optimize import brentq}

and see if it converges given an initial bracket of (-1.1,1.09). As always, include hard copy of your code and its output.

(b) [2 points] If \texttt{brentq} does converge, how many function evaluations does it use in order to get an absolute error of less than \( 10^{-15} \).

(a) The Python code below calls \textbf{brentq} from \texttt{scipy.optimize}:

```python
# brent.py
# Ringland 9/14/2011

from scipy.optimize import brentq
from math import tanh

def newt(f,fp,abstol,reltol,x):
    step = 2*abstol
    while abs(step) > abs(x)*reltol + abstol:
        step = -f(x)/fp(x)
        x += step
    return x

def myfunc(x):
    global ncalls
    ncalls += 1
    return tanh(x)

def myfuncprime(x):
    return 1 - tanh(x)**2

ncalls = 0
r = brentq( myfunc, -1.1, 1.09, xtol=1.0e-15 , rtol=0.0 )
print "brentq obtained root", r, "using", ncalls, "function calls"

ncalls = 0
r = newt( myfunc, myfuncprime, 1.0e-15, 0.0, 1.08 )
print "newton obtained root", r, "using", ncalls, "function calls"
```

\texttt{brentq} did converge to the root. The results are:

\texttt{brentq obtained root 3.51088757568e-16 using 7 function calls}

\texttt{newton obtained root 0.0 using 9 function calls}

(b) It took 7 function evaluations to get the desired (absolute) accuracy.

(c) The code also applies Newton's method to find the same root. It took two more iterations, 9 instead of 7, to get the requested accuracy. (Actually, if we recognize that each iteration of Newton requires function calls to both \( f \) and \( f' \), we might say Newton is costing twice as much as Brent.) By any measure, Newton's theoretically faster convergence was \textit{not} an advantage here.
1.4.X3

X3. GRADS ONLY, BUT UNDERGRADS CAN DO FOR EXTRA CREDIT.
(From Atkinson.) We want to create a method for calculating the square root of any (nonnegative) number, a, that uses a number of Newton iterations. We can write
\[ a = b \times 2^m \]
where m is an even integer and \(1/4 \leq b < 1\). Then
\[ \sqrt{a} = \sqrt{b} \times 2^{m/2}, 1/2 \leq \sqrt{b} < 1. \]
Thus the problem reduces to calculating \(\sqrt{b}\) for \(1/4 \leq b < 1\). Use the linear interpolating formula
\[ x_0 = (2b+1)/3 \text{ for } 1/4 \leq b < 1 \]
as an initial guess for Newton iteration to find \(\sqrt{b}\).
(a) [2 points] Find the maximum initial error \(|e_0| = |x_0-\sqrt{b}|\).
(b) [2 points] Show that for any allowed \(b\), \(x_i \geq 1/2\) for all \(i > 0\). Hint: to see what's going on, make a sketch of Newton's "g" for \(f(x) = x^2 - b\). Then use "Calc I" techniques.
(c) [3 points] Obtain a bound for \(e_{i+1}\) in terms of \(e_i\). Hint: Like we did in class, write Taylor's theorem with the remainder at quadratic order, expanding at \(x_i\) and evaluating at \(\sqrt{b}\). I.e. \(f(\sqrt{b})\) (which = 0 ) = \(f(x_i) + (\sqrt{b}-x_i) f'(x_i) + R_2\).
(d) [2 points] From your result in (c), obtain a bound for \(e_i\) in terms of \(|e_0|\).
(e) [2 points] Use your result in (d) to determine how many iterations, \(n\), suffice to be sure that
\[ 0 \leq |e_n| \leq 2^{-53}, \]
which is the limit of IEEE double precision (64-bit floating point), regardless of the value of \(b\) (as long as it's in \([1/4,1]\)).
1.4 X 3 (b) 
We have \( x_{i+1} = \frac{x_i}{2} + \frac{b}{2x_i} \equiv g(x_i) \)
with \( b \in [\frac{1}{4}, 1] \). 

We also know \( x_0 \geq \frac{1}{2} \).

Can we show that if \( x_i \geq \frac{1}{2} \), then \( x_{i+1} \geq \frac{1}{2} \)?

What does \( g \) look like?

\[
g'(x) = \frac{1}{2} - \frac{b}{2x^2} \\
g''(x) = \frac{b}{x^3} > 0 \\
\]

So, since \( g(x) \to \infty \) as \( x \to 0 \) and \( g(x) \to +\infty \) as \( x \to +\infty \), there is a unique local minimum which is a global minimum for \( x > 0 \). Let's find it:

\[
g'(x) = 0 \Rightarrow \frac{1}{2} = \frac{b}{2x^2} \Rightarrow x = \sqrt{b}.
\]

Thus \( g(x > 0) \geq g(\sqrt{b}) = \sqrt{b} \in [\frac{1}{2}, 1] \).

Therefore \( x_i \geq \frac{1}{2} \Rightarrow x_{i+1} \geq \frac{1}{2} \), and if \( x_0 \geq \frac{1}{2} \), then all \( x_i \) are \( \geq \frac{1}{2} \).

Slicker method:
\[
x_{i+1} = \frac{x_i^2 + b}{2x_i} = \frac{x_i^2}{2x_i} + \frac{\sqrt{b}^2}{2x_i}
\]

If \( x_i > 0 \), then since \( a^2 + b^2 \geq 2ab \) (because \((a-b)^2 \geq 0\))
\[
x_{i+1} \geq \frac{2x_i\sqrt{b}}{2x_i} = \sqrt{b} \geq \frac{1}{2}.
\]
Let the function whose zero we’re finding be 
\[ f(x) = x^2 - b. \]

Restrating the analysis we did in class, Taylor says
\[ \frac{f(\sqrt{b}) = f(x_i) + f'(x_i)(\sqrt{b} - x_i) + \frac{f''(c)}{2} (\sqrt{b} - x_i)^2}{0} \]
with c_i between x_i and \( \sqrt{b} \).

Observe that for our particular f, \( f(x) = 2x, f''(x) = 2 \)
so the lack of knowledge doesn’t matter: we have
exactly
\[ 0 = f(x_i) + f'(x_i)(\sqrt{b} - x_i) + (\sqrt{b} - x_i)^2 \]
Dividing by \( f'(x_i) \), which is OK as long as \( x_i \neq 0 \),
and re-arranging, we have
\[ x_i - f(x_i) - \sqrt{b} = \frac{(\sqrt{b} - x_i)^2}{f'(x_i)} \]
or, defining \( e_i = x_i - \sqrt{b} \),

\[
\begin{align*}
\frac{e_i + 1}{e_i} &= \frac{e_i^2}{2x_i} \\
This relationship is exact; next we approximate.
\end{align*}
\]

Since from (b), \( x_i \geq 1/2 \) for \( i > 0 \), and since \( x_0 \geq 1/2 \),
we have \( e_{i+1} \leq e_i^2 \) for all \( i \). QED.

Consequently \( |e_i| \leq |e_0|^{2^i} \leq (\frac{1}{24})^{2^i} \).

We seek \( |e_i| \leq 2^{-s_3} \), which is satisfied if
\[
\left( \frac{1}{24} \right)^{2^i} \leq 2^{-s_3}
\]
or \[ 2^{i \log \left( \frac{1}{24} \right)} \leq -s_3 \log 2 \]
\[ 2^i \geq -s_3 \log 2 = \frac{s_3 \log 2}{\log \left( \frac{1}{24} \right)} \]
or \[ i \log 2 \geq \log \left( \frac{s_3 \log 2}{\log \left( \frac{1}{24} \right)} \right) \]
\[ i \geq \frac{s_3 \log 2}{\log 2} \approx 53.1 log 2 \]

Thus:\
Answer: \boxed{4 \text{ iterations suffice}}
(and 3 iterations probably do not).
2.1X2

I find that on my laptop I get about 80 million operations per second for +,-,\ *, and about half that for /.

Since my laptop's CPU clock speed is 1.33GHz = 1.33e9 cycles per second, this means that on my laptop each +, -, or * operation take about (rounding to nice numbers) 15 clock cycles, and / about 30 cycles.

Let us assume that the architecture of CPU in "copper" is essentially the same, so that it uses the same number of clock cycles per floating-point operation, but it runs at 2.66GHz (twice as fast as my laptop).

Therefore, on copper, we get about $2.66 \times 10^9 / 15 \sim 1.773 \times 10^8$ operations (+,-,\*) per second (177 "megaflops") and half as many divides per second ("\(\Delta\" \sim 2").

(a) For a large matrix, GE takes the equivalent of about $2n^{3/3}$ arithmetic operations. In 1 minute = 60 seconds, copper can perform $60 \times 1.77 \times 10^8 \sim 1.06 \times 10^{10}$ operations (+,-,\*), so in 1 minute we can do GE on an nxn matrix for which

$$2n^{3/3} \sim 1.06 \times 10^{10}$$

i.e.

$$n \sim 2517$$

(Depending on how much rounding you did, you could get answers as much as 3 different from this.)

(b) In 1 hour = 3600 seconds, copper can perform $3600 \times 1.77 \times 10^8 \sim 6.384 \times 10^{11}$ operations (+,-,\*), so in 1 minute we can do GE on an nxn matrix for which

$$2n^{3/3} \sim 6.384 \times 10^{11}$$

i.e.

$$n \sim 9856$$

(Again answers within plus or minus a few from this are ok.)

Note that going from 1 minute to 1 hour gains us only a factor of $60^{1/3} \sim 4$ in n.

(c) Back-solving, or back-substitution for large matrices takes about $n^2$ operations, which for these problems would take about

$$2517^2 / 1.773 \times 10^8 \sim 0.0357 \text{ seconds}, \text{ or } \sim 36 \text{ milliseconds.}$$

and

$$9856^2 / 1.773 \times 10^8 \sim 0.5478 \text{ seconds, or } \sim 0.5 \text{ sec,}$$

respectively. (Hardly any time at all, compared to the time for GE.)

(d) An nxn matrix, with n=9856, requires $9856^2 \times 8 \sim 777 \text{ megabytes}$ to store. On copper we have 6 gigabytes, so both this matrix and the smaller one will fit easily into main memory on this machine.