Math 2400 - Numerical Analysis

Homework #2 Solutions

1. Implement a bisection root finding method. Your program should accept two endpoints, a tolerance limit and a function for input. It should then output the final approximation and the number of iterations. Make sure that the program checks that the initial interval is acceptable for this method.
   Here is my code with comments:

```
function [y,iter]=bisec(f,a,b,tol)

if feval(f,a)*feval(f,b)>0
   % Check if we meet conditon for bisection method
   error('Need f(a)f(b)<0');
end
if(abs(b-a)<=tol)
   % Are we starting with a solution ?
   error('Interval too small for give tolerance')
end
%Note: for the bisection method, we cannot get
%      stuck in an infinite loop.  So don't really
%      need a maximum number of iterations.  Ok if we
%      have one though.
iter=1; % we count the iterations for comparison
while(abs(b-a)>tol)
   c=(a+b)/2;  % The midpoint
   % Which inteval has the root?
   if (feval(f,a)*feval(f,c)<=0)
      b=c;
   else
      a=c;
   end
   iter=iter+1;
end
y=c;  % return the best guess.
return
```

2. Implement a Newton's method root finding method. Your program should accept an initial guess, a tolerance(for the relative difference between successive approximations), a function and it's derivative for input. It should then output the final approximation and the number of iterations. Make sure that your program can't be stuck in an infinite loop or divide by 0.
   Here is my code:

```
function [y,iter]=newton(f,fp,x0,tol)

max=100;  % The maximum permited number of iterations
iter=1;   % We start on the first interation

if (feval(fp,x0)==0)
```

```
   %  Check to see before we get div by 0 erro
   error('Derivative of function is 0')
end

x1=x0-feval(f,x0)/feval(fp,x0);  % The first Newton step

while(abs((x1-x0)/x0)>tol)
     x0=x1; % update the guess
     if (feval(fp,x0)==0)
%  Check to see before we get div by 0 erro
       error('Derivative of function is 0')
     end
     x1=x0-feval(f,x0)/feval(fp,x0); % next iteration
     iter=iter+1;
     if (iter>=max)
       error('Maximum iterations exceeded')
     end
end
     y=x1;  % return solution
return
```

3. Use the programs you have written to find roots for the following:

Function	Newton's Method guess	Bisection Method Interval
$\tan(x) - 2x$	1.4	$a = 1$, $b = 1.4$
$65x^4 - 72x^3 - 1.5x^2 + 16.5x - 1$	1	$a = 0$, $b = 1$
$x^3 - 6x^2 + 12x - 8$	3	$a = 1$, $b = 3$

Use 10^{-4} as your tolerance for both methods.
Discuss the appropriateness of the methods and initial guess to the above problems.
I had to write the m-files to provide the functions. Here is a listing of the 6 m=files. The files f1.m f2.m f3.m are for the function values. The files f1p.m, f2p.m and f3p.m are for the derivatives.

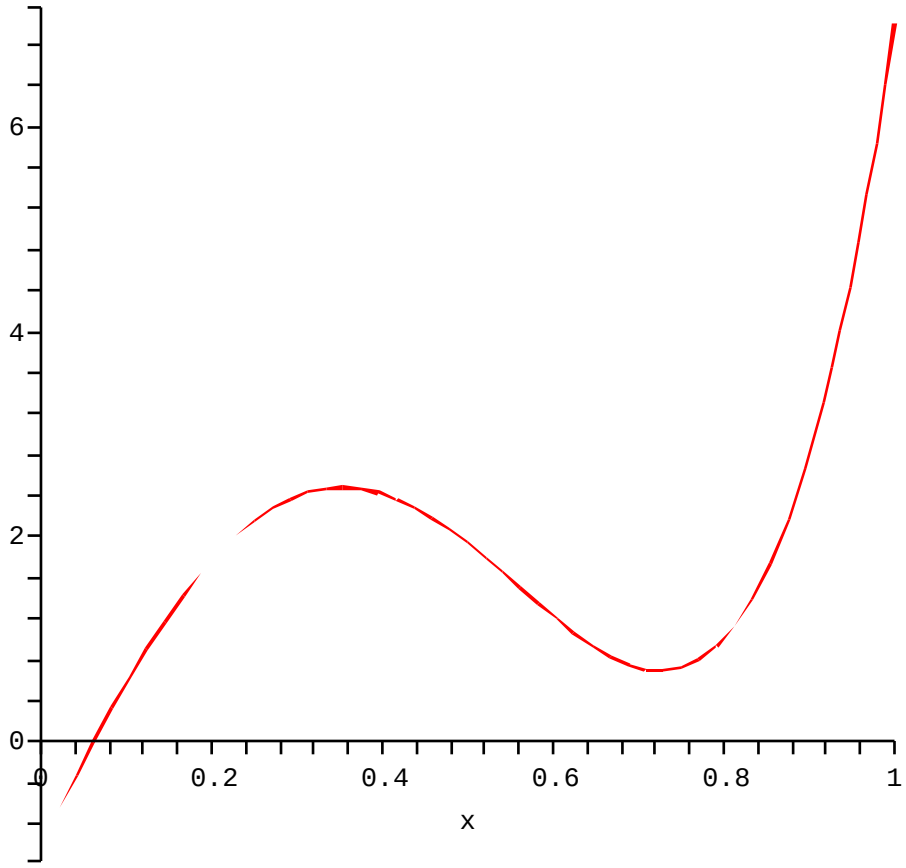
```
function y=f1(x)

y=tan(x)-2*x;

return

function y=f1p(x)
  y=sec(x)^2-2;
return

function y=f2(x)

y=65*x^4-72*x^3-1.5*x^2+16.5*x-1;

return
```

```
function y=f2p(x)
   y=260*x^3-216*x^2-3*x+16.5;
return

function y=f3(x)

y=x^3-6*x^2+12*x-8;

return

function y=f3p(x)
y=3*x^2-12*x+12;
return
```

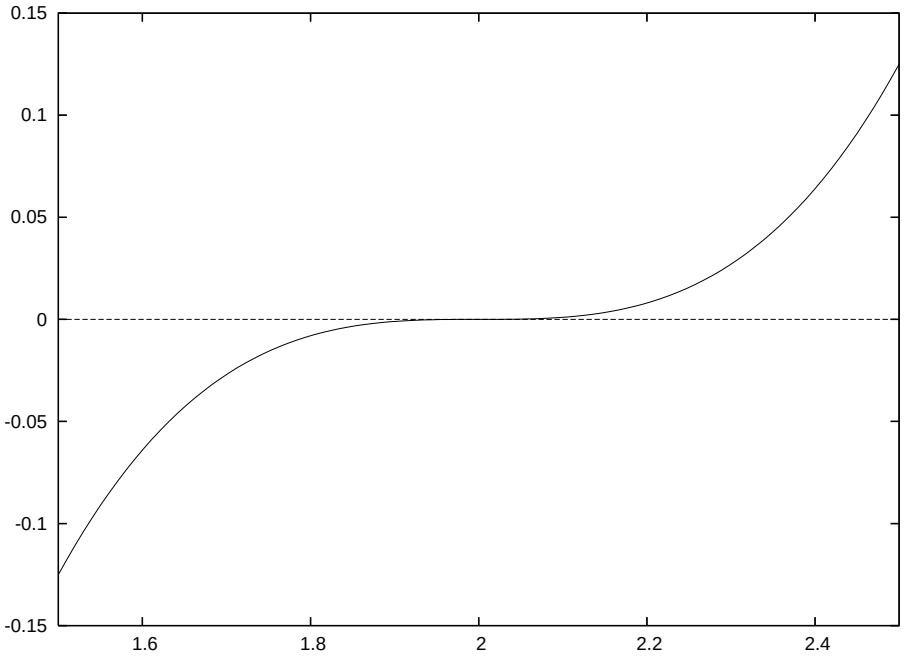
Here is the Matlab output for this question:

```
octave:2> [y n]=newton('f1','f1p',1.4,0.0001)
y = 1.1656
n = 6
octave:3> [y n]=bisec('f1',1,1.4,0.0001)
y = 1.1655
n = 13
octave:4> [y n]=newton('f2','f2p',1,0.0001)
y = 0.061933
n = 15
octave:5> [y n]=bisec('f2',0,1,0.0001)
y = 0.061951
n = 24
octave:6> [y n]=newton('f3','f3p',3,0.0001)
y = 2.0003
n = 20
octave:7> [y n]=bisec('f3',1,3,0.0001)
y = 1.9999
n = 16
octave:8> quit
```

The results for the first function are more or less what we would expect. Newton's method converges must faster then the bisection method. For the second function the performance of the two methods seems comparable. If we take a look at the second function, we can see what is happening.



By guessing 1, we are getting stuck around the minimum located near 0.7. If we where to use any initial guess less than 0.3, we would get very rapid convergence. As well if we where to decrease the tolerance, Newton's method would start to appear much better.
For the third equation, Newton's method also appears to be worse than the bisection method. The actual root is at exactly 2, so not only did Newton's method take longer to converge, but the answer is less accurate. Again we look at a graph of the function to see what is happening.



As you can see from the graph, the function is very flat near its root at $x = 2$. This means that the derivative will become very small as we approach the root. For this example we also note that the bisection method should do much better than my implementation. The very first guess with these values is the actual root. If I had considered the possibility of one of the guesses being the exact root, my program would have converged after one iteration. Since I didn't consider this possibility, I continue on for an additional 15 iterations and at some point some roundoff error enters the picture. Thus the 16th guess is not as good as the first.

4. In this question, we will find interpolating polynomials of degree at most 6 for the function $f(x) = e^{-2x}$ using equally spaced points and using the Chebyshev roots.
To solve for this problem, I wrote code to find the divided difference table and plot the resulting polynomial. Since this is part of the current assignment I will not include the code, but the derivation of the table can follow the examples in the notes for lab2 and lab3.
Here is a listing of the function I used to plot an interpolating polynomial given its divided difference table and the x values.

```
function [xi,y]=divdifplot(x,F)
     len=length(x);
xi=linspace(x(1),x(len),50)';
        y=zeros(50,1);


for i=1:len
  p=ones(50,1);
  for j=1:(i-1)
            p=p.*(xi-x(j));
          end
          y=y+F(i,i).*p;
        end
```

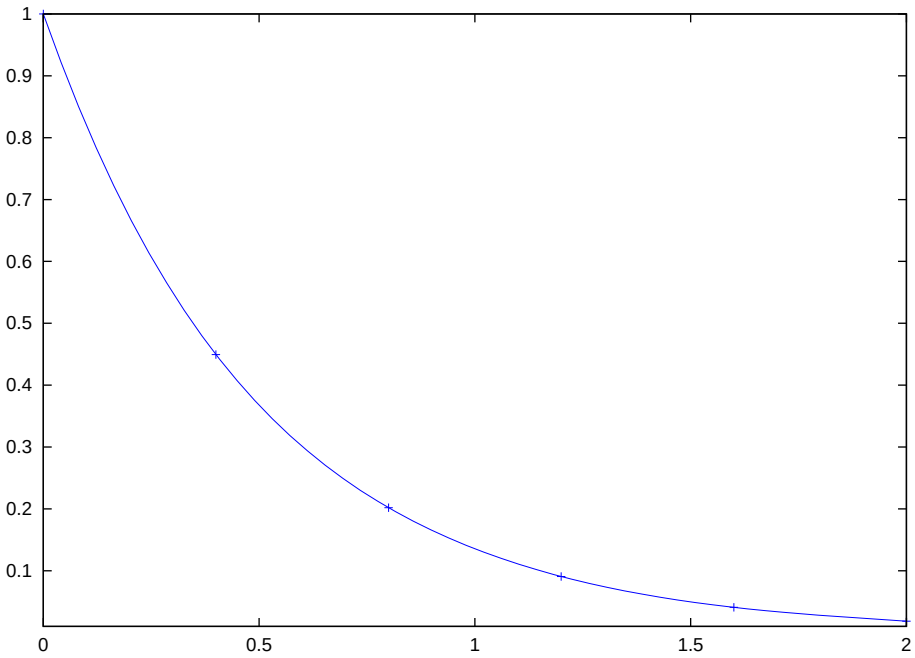
```
end
```

(a) Construct (use the computer or do it by hand) a polynomial of degree at most five, $P_5(x)$, interpolating $f(x)$ at the points $x_k = 0.4(k-1)$, $k = 1, \ldots, 7$ (you should have equally spaced points between 0 and 2). Plot $P_5(x)$ with a solid line and plot the points $f(x_k)$ with a + for $0 < x < 2$.
I made a typo a mistake here the .4 should be $\frac{1}{3}$, or we should be considering a polynomial of degree at most 5. The way the statement now reads, the last point wil l be 2.4. People can do it either way and I won't take of marks. Sorry for the confusion.
Here is the MATLAB session to generate the polynomial and the graph.

```
octave:2> x=linspace(0,2,6)'
x =

   0.00000
   0.40000
   0.80000
   1.20000
   1.60000
   2.00000

octave:3> y=exp(-2.*x)
y =

   1.000000
   0.449329
   0.201897
   0.090718
   0.040762
   0.018316

octave:4> F=divdif(x,y)
F =

   1.00000   0.00000   0.00000   0.00000   0.00000   0.00000
   0.44933  -1.37668   0.00000   0.00000   0.00000   0.00000
   0.20190  -0.61858   0.94762   0.00000   0.00000   0.00000
   0.09072  -0.27795   0.42579  -0.43486   0.00000   0.00000
   0.04076  -0.12489   0.19132  -0.19539   0.14966   0.00000
   0.01832  -0.05612   0.08597  -0.08780   0.06725  -0.04121

octave:5> [x1,y1]=divdifplot(x,F);
octave:6> plot(x1,y1)
octave:7> hold on
octave:8> plot(x,y,'+')
octave:9> print("hw3q4a.eps")
```

```
octave:10> diary off
```

Here is the graph from that session.



(b) Use a linear transformation to move the zeros of the degree 6 Chebyshev polynomial to the interval $[0, 2]$.
We can plug in the formula for the translated Chebyshev roots

$$x_i = \frac{b+a}{2} + \frac{b-a}{2} \cos\left(\frac{(2i-1)\pi}{2n}\right)$$

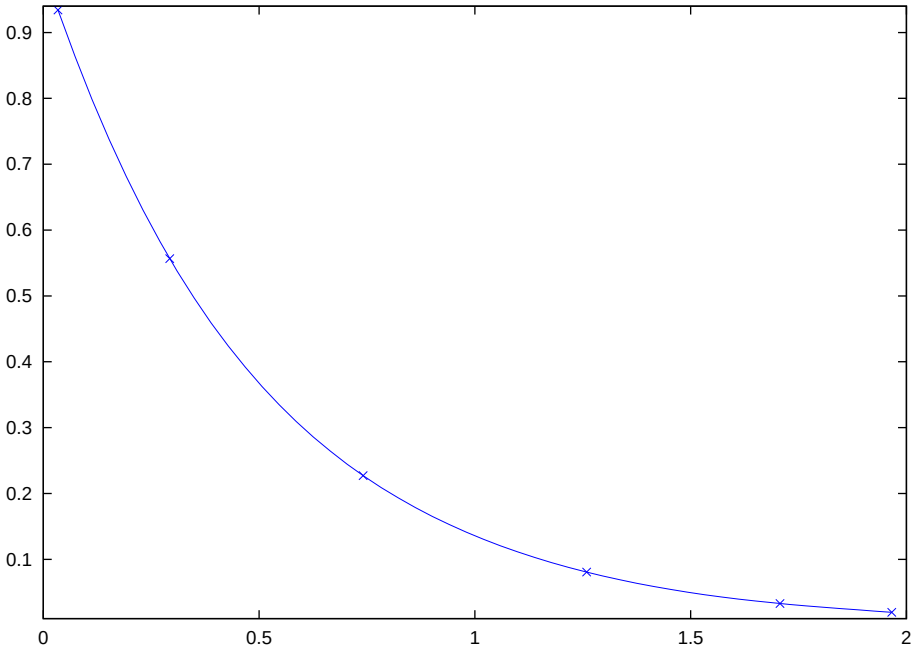
to find the interpolation points. As this goes with then next question, I will find these point and the interpolating polynomial in the MATLAB session posted after the next question.

(c) Construct a polynomial of degree at most six, $\bar{P}_6(x)$, interpolating $f(x)$ at the points found in 4b. Plot $\bar{P}_6(x)$ with a solid line and plot the points $f(x_k)$ with a + on the interval $0 < x < 2$.
Here is the MATLAB session I used to answer the previous two questions:

```
octave:1> x=zeros(6,1)
x =

   0
   0
   0
   0
   0
   0

octave:2> for i=1:6
> x(i)=1+cos((2*i-1)*pi/12);
```

```
> end
octave:3> x
x =

   1.965926
   1.707107
   1.258819
   0.741181
   0.292893
   0.034074

octave:4> y=exp(-2.*x)
y =

   0.019607
   0.032902
   0.080650
   0.227101
   0.556668
   0.934122

octave:5> F=divdif(x,y)
F =

   0.01961   0.00000   0.00000   0.00000   0.00000   0.00000
   0.03290  -0.05137   0.00000   0.00000   0.00000   0.00000
   0.08065  -0.10651   0.07798   0.00000   0.00000   0.00000
   0.22710  -0.28292   0.18263  -0.08545   0.00000   0.00000
   0.55667  -0.73517   0.46820  -0.20193   0.06962   0.00000
   0.93412  -1.45837   1.02276  -0.45280   0.14995  -0.04158

octave:6> [xxx,yyy]=divdifplot(x,F);
octave:7> plot(xxx,yyy)
octave:8> hold on
octave:9> plot(x,y,'x')
octave:10> print("hw3q4c.eps")
```

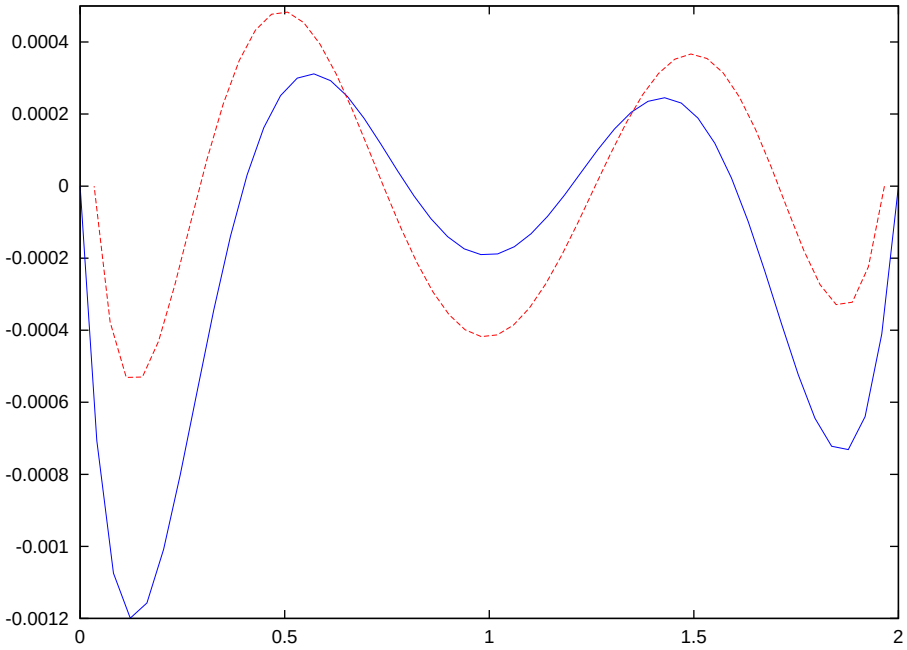
Here is the graph resulting from the above MATLAB session:



(d) Plot $f(x) - P_5(x)$ and $f(x) - \bar{P}_5(x)$ on the same graph for $0 < x < 2$. Explain why one of the polynomials has a lower maximum error. Here is the tail end of the MATLAB session I used to generate the final graphs

```
octave:12> plot(x1,exp(-2.*x1)-y1,'b')
octave:13> plot(xxx,exp(-2.*xxx)-yyy,'r')
octave:14> hold on
octave:15> plot(x1,exp(-2.*x1)-y1,'b')
octave:16> print("hw3q4d.eps")
```

Here is the graph generated.



The error for P_5 is graphed with the dotted cure. As you can see from the graph both

polynomials provide a good approximation in the middle, but near the endpoints, $\bar{P}_5$ gives a much better approximation. Actually in the interior, P_5 is slightly better than $\bar{P}_5$, but using the Chebyshev roots as interpolation points spreads the error more evenly over the interval, so the **global** error bound we have for $\bar{P}_5$ is much lower.