MATH 5610 HW4 SOLUTIONS

Problem 1 See attached code & output.

Problem 2 $p(x) = 2 - (x+1) + x(x+1) - 2x(x+1)(x-1)$

interpolates the first four points in table

x	-1	0	1	2	3
$f(x)$	2	1	2	-7	10

The interp. poly at all nodes is of the form

$$r(x) = p(x) + c\, x(x+1)(x-1)(x-2)$$

Since we already have $r(x_i) = f(x_i)$ for first four nodes we can find c by enforcing $r(3) = f(3)$.

$$c = \frac{r(3) - p(3)}{3(3+1)(3-1)(3-2)} = \frac{10-(-38)}{24} = 2$$

thus $\boxed{r(x) = p(x) + 2x(x+1)(x-1)(x-2)}$

Problem 3 See attached code & output

Problem 4 Show that if f is a poly, then $f[x_0, \ldots, x_n]$ is a poly in each of its variables.

proof: i) $f[x_0] = f(x_0)$ = poly in x_0 by hypothesis.

ii) (not needed for induction, but helpful)

$$f[x_0, x] = \frac{f(x) - f(x_0)}{x - x_0} = \frac{r(x)}{x - x_0}$$

$r(x)$ = polynomial with:

$r(x_0) = 0$

$\Rightarrow r(x) = (x - x_0)\, q(x)$, where $q(x)$ is a poly.

$$\Rightarrow f[x_0, x] = \frac{(x - x_0)\, q(x)}{x - x_0} = q(x) \text{ is a poly.}$$

iii)

$$f[x_0, \ldots, x_{n-1}, x] = \frac{f[x_1, x_2, \ldots, x_{n-1}, x] - f[x_0, x_1, \ldots, x_{n-1}]}{x - x_0}$$

Let $r(x) = f[x_1 \ldots x_{n-1}, x] - f[x_0, \ldots x_{n-1}]$

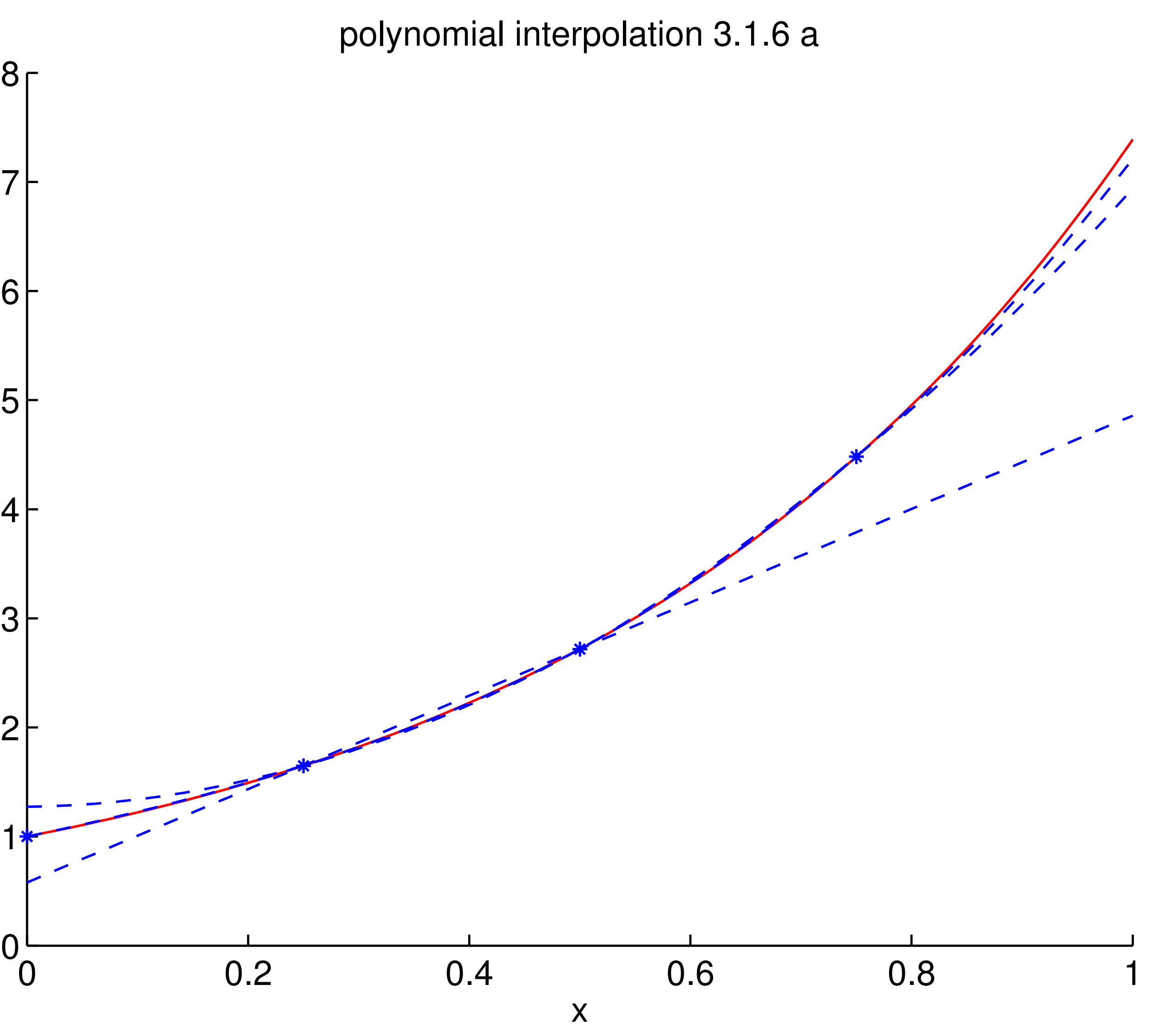
By induction hyp. $r(x)$ is a poly, (it involves only $n-1$ order d.d.)

$r(x_0) = f[x_1 \cdots x_{n-1}, x_0] - f[x_0, \cdots x_{n-1}]$

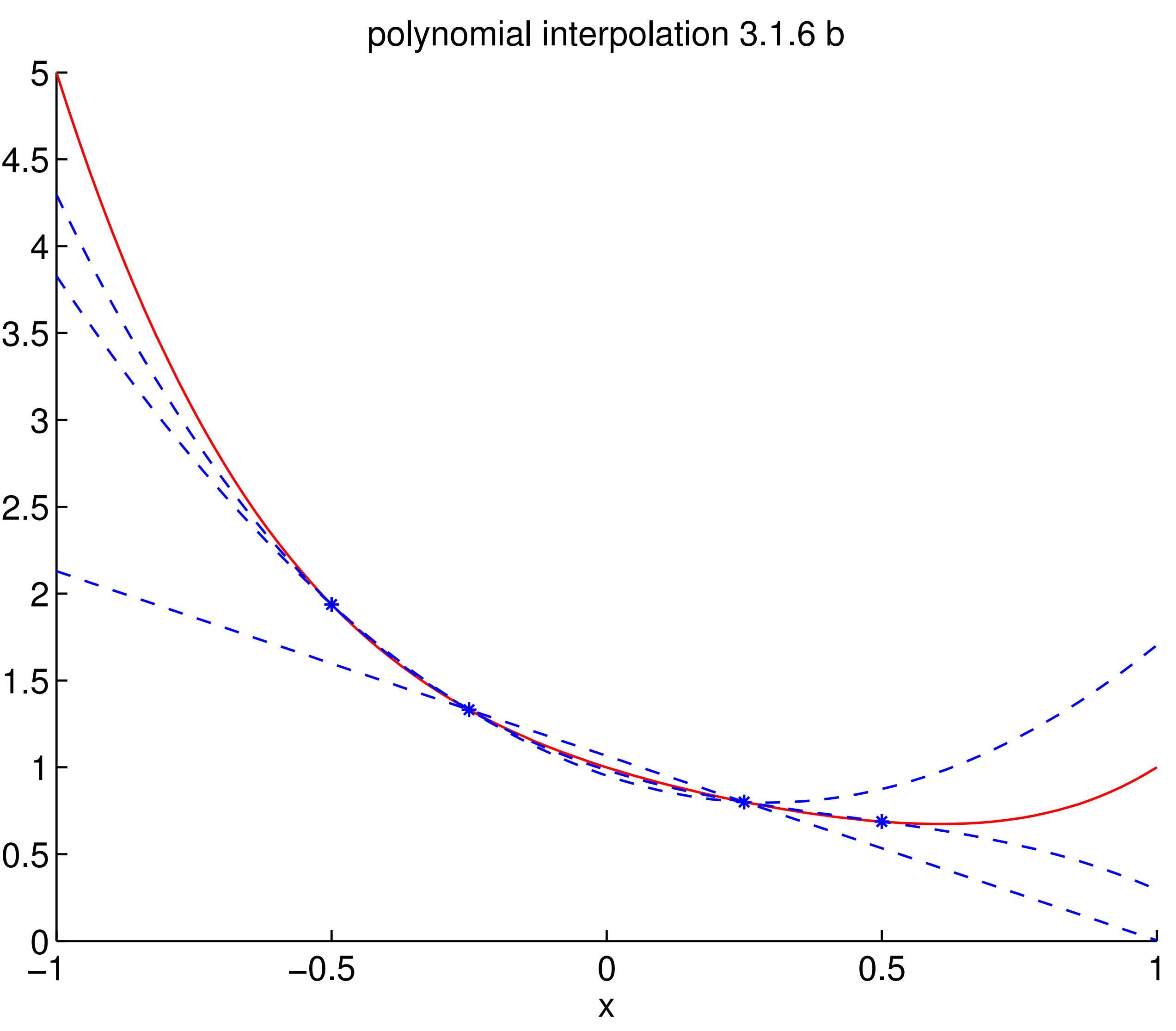
$= 0$ since d.d. are indep. of permutations of arg.

$\Rightarrow r(x) = (x - x_0)\, q(x)$, where $q(x)$ is a poly

$$\Rightarrow f[x_0, \cdots x_{n-1}, x] = \frac{(x - x_0)\, q(x)}{x - x_0} = q(x) \text{ is a poly.}$$



polynomial interpolation 3.1.6 a
0
1
2
3
4
5
6
7
8
0
0.2
0.4
0.6
0.8
1
x



polynomial interpolation 3.1.6 b
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
−1
−0.5
0
0.5
1
x

```
>> prob1
 --- Part a ---
            z0            z1            z2 iterations          root
   1.00000000    2.00000000    3.00000000        5    4.12310563 +    0.00000000
i
  -4.00000000   -4.50000000   -5.00000000        4   -4.12310563 +    0.00000000
i
   0.00000000    1.00000000    2.00000000        9   -2.49999999 +    1.32287554
i
the remaining root is conjugate of the previous one
double check with Matlab:
  4.123105625617665
 -4.123105625617663
 -2.499999999999999 + 1.322875655532298i
 -2.499999999999999 - 1.322875655532298i

 --- Part b ---
            z0            z1            z2 iterations          root
  -5.00000000   -4.00000000   -3.00000000        4   -3.54823290 +    0.00000000
i
   3.00000000    4.00000000    5.00000000        4    4.38111344 +    0.00000000
i
   0.00000000    1.00000000    2.00000000        5    0.58355973 +    1.49418801
i
the remaining root is conjugate of the previous one
double check with Matlab:
  4.381113440995941
 -3.548232897979697
  0.583559728491880 + 1.494188006011256i
  0.583559728491880 - 1.494188006011256i

>> prob3
 --- B&F 3.1.6 a and 3.1.8 a ---
    degree      Lagrange        Newton     err bound
         1    5.5643e-02    5.5643e-02    1.1294e-01
         2    1.4298e-02    1.4298e-02    2.4094e-02
         3    2.5560e-03    2.5560e-03    5.1801e-03
 --- B&F 3.1.6 b and 3.1.8 b ---
    degree      Lagrange        Newton     err bound
         1    6.6406e-02    6.6406e-02    2.5000e-01
         2    4.6876e-02    4.6876e-02    3.1250e-02
         3    1.5626e-02    1.5626e-02    1.5625e-02
>>
```



```
% HW 4 Problem 1
% B&F 2.6.4 a,b
tol = 1e-5;
maxit = 20;
format long

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
fprintf(' --- Part a --- \n');
p = [1 5 -9 -85 -136];

fprintf('%13s %13s %13s %8s %13s\n','z0','z1','z2','iterations','root');

z0=1; z1=2; z2=3;
[z,k]=muller(z0,z1,z2,p,maxit,tol);
fprintf('%13.8f %13.8f %13.8f %8d %13.8f + %13.8fi\n',z0,z1,z2,k,real(z),imag(z));

z0=-4; z1=-4.5; z2=-5;
[z,k]=muller(z0,z1,z2,p,maxit,tol);
fprintf('%13.8f %13.8f %13.8f %8d %13.8f + %13.8fi\n',z0,z1,z2,k,real(z),imag(z));

z0=0; z1=1; z2=2;
[z,k]=muller(z0,z1,z2,p,maxit,tol);
fprintf('%13.8f %13.8f %13.8f %8d %13.8f + %13.8fi\n',z0,z1,z2,k,real(z),imag(z));

fprintf('the remaining root is conjugate of the previous one\n');

fprintf('double check with Matlab:\n');
disp(roots(p));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
fprintf(' --- Part b --- \n');
p = [1 -2 -12 16 -40];

fprintf('%13s %13s %13s %8s %13s\n','z0','z1','z2','iterations','root');


z0=-5; z1=-4; z2=-3;
[z,k]=muller(z0,z1,z2,p,maxit,tol);
fprintf('%13.8f %13.8f %13.8f %8d %13.8f + %13.8fi\n',z0,z1,z2,k,real(z),imag(z));

z0=3; z1=4; z2=5;
[z,k]=muller(z0,z1,z2,p,maxit,tol);
fprintf('%13.8f %13.8f %13.8f %8d %13.8f + %13.8fi\n',z0,z1,z2,k,real(z),imag(z));


z0=0; z1=1; z2=2;
[z,k]=muller(z0,z1,z2,p,maxit,tol);
fprintf('%13.8f %13.8f %13.8f %8d %13.8f + %13.8fi\n',z0,z1,z2,k,real(z),imag(z));

fprintf('the remaining root is conjugate of the previous one\n');

fprintf('double check with Matlab:\n');
disp(roots(p));
```

```
% HW4 Problem 3
%
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% B&F 3.1.6 a and 3.1.8 a

fprintf(' --- B&F 3.1.6 a and 3.1.8 a --- \n');

% interpolation node absicssas (we reordered st the node at which we evaluate
% the interpolation is always inside the interval)
xd = [0.25,0.5,0.75,0];
% ordinates
yd = [1.64872, 2.71828, 4.48169,1];
% evaluation point
x  = 0.43;
% true function (USING VECTORIZED OPERATIONS)
f = @(x) exp(2*x);

% The function and first few derivatives are:
%  f(z)     = exp(2*x)
%  f'(z)    = 2*exp(2*x)
%  f''(z)   = 4*exp(2*x)
%  f'''(z)  = 8*exp(2*x)
%  f''''(z) = 16*exp(2*x)

% here we give the part of the bound depending on f. It would be better
% to find the max of this functions over the interpolation interval [0,0.75]

% for interp poly of degree 1 we bound  f''
fmax(1) = 4*exp(2*0.75);
% for interp poly of degree 2 we bound  f'''
fmax(2) = 8*exp(2*0.75);
% for interp poly of degree 2 we bound  f''''
fmax(3) = 16*exp(2*0.75);

% vectors to store interpolation polynomial evaluated at x
% and error
lagr = zeros(3,1); newt = zeros(3,1);  err = zeros(3,1);
for k=1:3, % loop on degree
 % Lagrange interpolation
 % little trick to take only  the first k nodes
 lagr(k) = lagrange(xd(1:k+1),yd(1:k+1),x);

 % Newton interpolation using divided differences
 d = newtondd(xd(1:k+1),yd(1:k+1));
 newt(k) = newtonev(xd(1:k+1),d,x);

 % compute error bound
 err(k) = fmax(k)/factorial(k+1);
 for j=1:k+1;
  err(k) = err(k)*(x-xd(j));
 end;% for j
end;% for k

% display errors
fprintf('%10s %13s %13s %13s\n',...
  'degree','Lagrange','Newton','err bound');
for k=1:3,
 fprintf('%10d %13.4e %13.4e %13.4e\n',...
  k,abs(lagr(k)-f(x)),abs(newt(k)-f(x)),abs(err(k)));
end;% for k

% Show all three interpolation polynomials in a plot


figure(1); % IF DOING MORE THAN ONE FIGURE, CHANGE FIGURE NUMBER
clf;       % clears figure
xs = linspace(0,1);  % equally space points in an interval containing interp. po
```



```
ints

% we are going to do several plots, so we tell Matlab to
% accumulate all the plot commands
hold on;

% Plots the true function
plot(xs,f(xs),'r-');

% Plots the interpolation nodes
plot(xd,yd,'*');

% Plot the interpolation polynomials
for k=1:3,
 % Newton interpolation using divided differences
 d = newtondd(xd(1:k+1),yd(1:k+1));
 newt = newtonev(xd(1:k+1),d,xs);
 plot(xs,newt,'b--');
end;

% We don't want to add anymore plots to the current figure
hold off;

% put some labels
xlabel('x');
title('polynomial interpolation 3.1.6 a');

% print to a file
print('-depsc2','prob3_a.eps');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% B&F 3.1.6 b and 3.1.8 b

fprintf(' --- B&F 3.1.6 b and 3.1.8 b --- \n');

% interpolation node absicssas (we reordered st the node at which we evaluate
% the interpolation is always inside the interval)
xd = [-0.25,0.25,-0.5,0.5];
% ordinates
yd = [1.33203, 0.800781, 1.93750, 0.687500];
% evaluation point
x  = 0;
% true function (USING VECTORIZED OPERATIONS)
f = @(x) x.^4 - x.^3 + x.^2 - x +1;

% The function and first few derivatives are:
%  f(z)     = z^4 - z^3 + z^2 - z +1
%  f'(z)    = 4*z^3 - 3*z^2 + 2*z - 1
%  f''(z)   = 12*z^2 - 6*z + 2
%  f'''(z)  = 24*z - 6
%  f''''(z) = 24

% here we give the part of the bound depending on f. It would be better
% to find the max of this functions over the interpolation interval [0,0.75]

% for interp poly of degree 1 we bound  f''
fmax(1) = 12*(-0.5)^2 - 6*(-0.5) + 2;
% for interp poly of degree 2 we bound  f'''
fmax(2) = 24*0.5 - 6;
% for interp poly of degree 2 we bound  f''''
fmax(3) = 24;

% vectors to store interpolation polynomial evaluated at x
% and error
lagr = zeros(3,1); newt = zeros(3,1);  err = zeros(3,1);
for k=1:3, % loop on degree
 % Lagrange interpolation
 % little trick to take only  the first k nodes
 lagr(k) = lagrange(xd(1:k+1),yd(1:k+1),x);
```

```
 % Newton interpolation using divided differences
 d = newtondd(xd(1:k+1),yd(1:k+1));
 newt(k) = newtonev(xd(1:k+1),d,x);

 % compute error bound
 err(k) = fmax(k)/factorial(k+1);
 for j=1:k+1;
  err(k) = err(k)*(x-xd(j));
 end;% for j
end;% for k

% display errors
fprintf('%10s %13s %13s %13s\n',...
  'degree','Lagrange','Newton','err bound');
for k=1:3,
 fprintf('%10d %13.4e %13.4e %13.4e\n',...
  k,abs(lagr(k)-f(x)),abs(newt(k)-f(x)),abs(err(k)));
end;% for k

% Show all three interpolation polynomials in a plot


figure(2);
clf;       % clears figure
xs = linspace(-1,1);  % equally space points in an interval containing interp. p
oints

% we are going to do several plots, so we tell Matlab to
% accumulate all the plot commands
hold on;

% Plots the true function
plot(xs,f(xs),'r-');

% Plots the interpolation nodes
plot(xd,yd,'*');

% Plot the interpolation polynomials
for k=1:3,
 % Newton interpolation using divided differences
 d = newtondd(xd(1:k+1),yd(1:k+1));
 newt = newtonev(xd(1:k+1),d,xs);
 plot(xs,newt,'b--');
end;

% We don't want to add anymore plots to the current figure
hold off;

% put some labels
xlabel('x');
title('polynomial interpolation 3.1.6 b');

% print to a file
print('-depsc2','prob3_b.eps');
```



```
% function [z3,k] = muller(z0,z1,z2,p,maxit,tol)
%
%   Finds a polynomial root given the initial guesses z0,z1,z2 using
%   a quadratic mode (Muller's method). This method allows convergence
%   to a complex root even if the z0,z1,z2 are real.
%
%  Inputs
%   z0,z1,z2  initial guesses
%   p         vector of polynomial coefficients, from highest to lowest degree
%   maxit     maximum number of iteration
%   tol       tolerance
%
%  Outputs
%   z3        approximation of the root
%   k         number of iterations
function [z3,k] = muller(z0,z1,z2,p,maxit,tol)
[q,fz0] = horner(p,z0); [q,fz1] = horner(p,z1); [q,fz2] = horner(p,z2);
for k=1:maxit,
 % compute coefficients of quadratic
 c = fz2;
 h1 = z1-z0;
 h2 = z2-z1;
 d1 = (fz1-fz0)/h1;
 d2 = (fz2-fz1)/h2;
 a = (d2 -d1)/(z2-z0);
 b = d2 +h2*a;

 % double check that a,b,c are OK
 %qd = @(z) a*(z-z2)^2 + b*(z-z2) + c; % quadratic model
 %[qd(z2)-fz2, qd(z1)-fz1, qd(z0)-fz0] % quadratic model should interp f

 delta = sqrt(b^2 - 4*a*c);

 % choose root with smallest magnitude
 if (abs(b-delta)<abs(b+delta))
  z3 = z2 + (-b+delta)/2/a;
 else
  z3 = z2 + (-b-delta)/2/a;
 end;

 % check convergence
 if abs(z3-z2)<tol break; end;

 % prepare next iteration
 [q,fz3] = horner(p,z3);

 if (abs(fz3)<tol) break; end;
 fz0=fz1; z0=z1; fz1=fz2; z1=z2; fz2=fz3; z2=z3;
end;
```

```
% Horner's algorithm to divide a polynomial p by the
% linear factor z-z0
%
% i.e.
%
%  p(z) = q(z) (z-z0) + r
%
% Inputs
%  p     vector of polynomial coefficients, from highest to lowest degree
%  z0    root of linear factor
%
%  Outputs
%  q     vector of polynomial coefficients for the quotient polynomial
%  r     residual
function [q,r] = horner(p,z0)
n = length(p);
q = zeros(size(p));
q(1) = p(1);
for k=1:n-1,
 q(k+1) = p(k+1) + z0 * q(k);
end;
r = q(n);
q = q(1:n-1);
```



```
% function y = lagrange(xd,yd,x)
%
% Evaluates Lagrange interpolation polynomial at some points
%
% Inputs:
%   xd    absissa of data points
%   yd    ordinate of data points
%   x     abscissa where we want to evaluate Lagrange interp. polynomial
%
% Outputs:
%   y     p(x)
%
%   One way ot test this is to run the following:
%      x = linspace(0,5);
%      xd = [1 2 3 4];
%      yd = [-1 2 -2 5];
%      y = lagrange(xd,yd,x) ;
%      plot(x,y,xd,yd,'r+')

function y = lagrange(xd,yd,x)
y = zeros(size(x));
for i=1:length(xd),
 yb = yd(i);
 for j=1:length(xd),
  if (j ≠ i)
   yb = yb .* (x-xd(j))/(xd(i)-xd(j));
  end;
 end;
 y = y + yb;
end;
```

```
% computes Newton's divided differences, needs only a vector for storage
%
% Inputs
%   x   interpolation nodes
%   f   function values
%
% Outputs
%   d   coefficients of Newton interpolation polynomial
%
%   One way ot test this is to run the following:
%      x  = linspace(0,5);
%      xd = [1 2 3 4];
%      yd = [-1 2 -2 5];
%
%      d = newtondd(xd,yd);
%      y = newtonev(xd,d,x);
%      plot(x,y,xd,yd,'r+')


function d = newtondd(x,f)
n = length(x)-1;
if (length(f)≠n+1) fprintf('length of input vectors must be the same'); end;
d = f;
for j=1:n,
 for i=n:-1:j,
  d(i+1) = (d(i+1)-d(i))/(x(i+1)-x(i-j+1));
 end;
end;
```



```
% function y = newtonev(xd,d,x)
%
% uses Horner's algorithm to evaluate the Newton interpolation polynomial
% at the points x, where the interpolation nodes are given in xd
%
% Inputs
%   xd  interpolation nodes
%   d   coefficients of the Newton interpolation polynomial
%   x   points of evaluation of the Newton interpolation polynomial
%
% Outputs
%   y   p(x)
%
function y = newtonev(xd,d,x)
% sanity check
if (length(d)≠length(xd))
 error('there must be as many divided differences as interpolation nodes');
end;

n = length(d); y = zeros(size(x));
% Horner's algorithm
y = d(n);
for k=n-1:-1:1,
 y = (x-xd(k)).*y + d(k);
end;
```