Computer Assignment 3 – Solutions

1. a) Matlab m-file `newton_coef.m`:

```
function [ck, ddif] = newton_coef(xi, yi)
% NEWTON_COEF Compute coefficients of the Newton interpolating
%    polynomial using the divided differences table.
  n = length(xi)-1;  % n - degree of the interpolating polynomial
  ddif = zeros(n+1,n+1);
  ddif(:,1) = yi; % 1st column of the table is equal to yi
  for k = 1:n
%    for j = 0:n-k
%      ddif(j+1,k+1) = (ddif(j+2,k)-ddif(j+1,k))./(xi(j+1+k)-xi(j+1));
%    end
% Or use ':' operator instead of the for loop over j:
    ddif(1:n-k+1,k+1) = (ddif(2:n-k+2,k)-ddif(1:n-k+1,k))./(xi(k+1:n+1)-xi(1:n-k+1));
  end
  ck = ddif(1,:)';  % Newton coefficients are in the first row of ddif
```

The result for the test example is as in the Assignment.

b) Matlab m-file `eval_newton.m`:

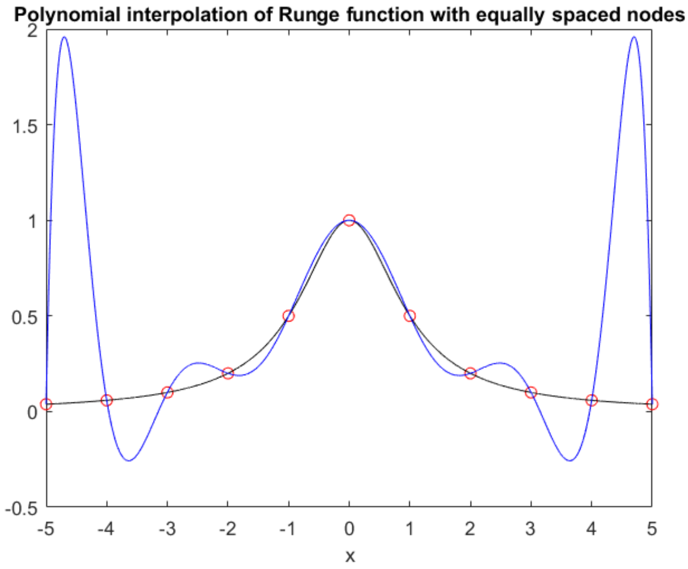
```
function y = eval_newton(x, ck, xi)
%  Evaluate the Newton interpolating polynomial at x (which could be a vector),
%  given nodes xi and coefficients ck.
  n = length(xi);  % number of nodes
  if n ~= length(ck)
    error('Number of coefficients and nodes must be equal');
  end
  y = ck(n)*ones(size(x));  % The size of y is the same as x
  for i = n-1:-1:1
    y = y.*(x - xi(i)) + ck(i);
  end
```

To test that it works correctly, one can evaluate the interpolating polynomial at the nodes `xi`. The result should be `yi`.

```
>> y = eval_newton(xi, ck, xi)
y =
       3
     -10
       2
```

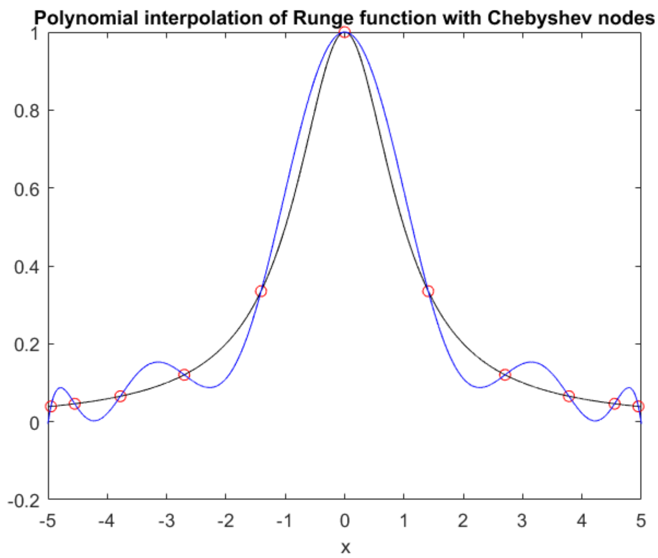
c) The following Matlab script performs the required computations:

```
  f = @(x)1./(1+x.^2);  % Define Runge function
  xi = (-5:5)';         % Define nodes
  yi = f(xi);           % Calculate function values at the nodes
  ck = newton_coef(xi,yi);  % Find Newton polynomial coefficients
  x = (-5:0.001:5)';    % Define values of x at which to compute function and 
polynomials
  plot(xi,yi,'ro',x,f(x),'k-'); hold on;  % Plot the function and interpolation 
points
  y = eval_newton(x,ck,xi);  % Evaluate Newton polynomial at x
  plot(x,y,'b-');            % Plot newton polynomial
  xlabel('x');
  title('Polynomial interpolation of Runge function with equally spaced nodes');
  err = max(abs(f(x)-y)),    % Compute the error

  err =
      1.9157
```



d) The same script as in c), but with Chebyshev nodes:

```
  f = @(x)1./(1+x.^2);  % Define Runge function
  n = 10; xi = 5*cos((2*(0:n)'+1)/(2*n+2)*pi); % Define Chebyshev nodes
  yi = f(xi);             % Calculate function values at the nodes
  ck = newton_coef(xi,yi);  % Find Newton polynomial coefficients
  x = (-5:0.001:5)';    % Define values of x at which to compute function and
polynomials
  plot(xi,yi,'ro',x,f(x),'k-'); hold on;   % Plot the function and interpolation
points
  y = eval_newton(x,ck,xi);  % Evaluate Newton polynomial at x
  plot(x,y,'b-');  % Plot newton polynomial
  xlabel('x');
  title('Polynomial interpolation of Runge function with Chebyshev nodes');
  err = max(abs(f(x)-y)),  % Compute the error

   err =
       0.1092
```



We see a much smaller maximum error.

2. a) Matlab function `ncspline.m` for computing the 2nd derivatives at the nodes of the natural cubic spline.

```
function ypp = ncspline(ti,yi)
% Determine the 2-nd derivatives of the natural cubic spline.
n = length(ti)-1;  % n is the number of spline intervals
a = zeros(n-1,1);  b = a;  c = a;  d = a;  % Allocate memory for a, b, c, and d
ypp = zeros(n+1,1);  % ypp(1) = ypp(n+1) = 0 by definition of natural cubic spline
for i = 1:n-1, % Define coefficients of the linear system of equations for ypp
  a(i) = (ti(i+1)-ti(i))/6;
  b(i) = (ti(i+2)-ti(i))/3;
  c(i) = (ti(i+2)-ti(i+1))/6;
  d(i) = (yi(i+2)-yi(i+1))./(ti(i+2)-ti(i+1)) - ...
         (yi(i+1)-yi(i))./(ti(i+1)-ti(i));
end
% Naive GE
% Note: In the obtained upper-triangular system, the values of c remain
%       unchanged, only the values of b and d are modified
for i = 2:n-1,
  b(i) = b(i) - (a(i)/b(i-1))*c(i-1);
  d(i) = d(i) - (a(i)/b(i-1))*d(i-1);
end
% Back substitution
ypp(n) = d(n-1)/b(n-1);
for i = n-2:-1:1,
  ypp(i+1) = (d(i) - c(i)*ypp(i+2))/b(i);
end
% Note that ypp(1) and ypp(n+1) remain unchanged (equal to 0).
```

b) Matlab function function `ncseval.m` for evaluating the natural cubic spline.

```
function y = ncseval(ti,yi,ypp,t)
% Given the nodes (ti,yi) and 2nd derivatives at the nodes (ypp),
% calculate the value of the natural cubic spline at t (scalar)
n = length(ti)-1;  % n is the number of spline intervals
% Find i such that ti(i) <= t <= ti(i+1)
if t < ti(1) || t > ti(n+1), % Check that t is in the range
  error('Value of t outside the range of node values ti');
end
switch 'binary'
case 'linear', % Linear search
  i = 1;
  while t > ti(i+1),
    i = i+1;
  end
case 'binary', % Binary search
  il = 1; ih = n+1;
  while il < ih-1,
    im = floor((il+ih)/2);
    if ti(im) < t,
      il = im;
    else
      ih = im;
    end
  end
  i = il;
end
% Evaluate the spline y = S_i(t)
hi = ti(i+1)-ti(i);  dt = t - ti(i);  % Compute coefficients of S_i(t)
a1 = (yi(i+1)-yi(i))/hi - hi*(ypp(i+1)+2*ypp(i))/6;
a2 = ypp(i)/2;
a3 = (ypp(i+1)-ypp(i))/(6*hi);
y = yi(i) + dt*(a1 + dt*(a2 + dt*a3));  % Compute polynomial a nested form
```

c) The following script computes natural cubic spline interpolation of the Runge function at equally spaced nodes:

```
 f = @(x)1./(1+x.^2);     % Define Runge function
 xi = (-5:5)';            % Define nodes
 yi = f(xi);              % Calculate function values at the nodes
 ypp = ncspline(xi,yi);   % Find y'' for the natural cubic spline
 x = (-5:0.001:5)';       % Define values of x at which to compute the spline
 plot(xi,yi,'ro',x,f(x),'k-'); hold on;  % Plot the function and interpolation
points
 y = zeros(size(x));
 for i = 1:length(x)
   y(i) = ncseval(xi,yi,ypp,x(i));
 end
 plot(x,y,'b-');  % Plot newton polynomial
 xlabel('t');
 title('Spline interpolation of the Runge function');
 err = max(abs(f(x)-y)),  % Compute the error

  err =
      0.0220
```

We see that the approximation by the natural cubic spline is much better. So, by sacrificing a bit of continuity of the interpolating function (i.e. infinitely differentiable, vs twice-continuously differentiable), we achieve much more accurate approximation.