Math 226A
Homework 4
Due Monday, December 11th

1. (a) Show that the polynomial

$$p(x) = 2^{-n} \left(T_{n+1}(x) - T_{n-1}(x) \right),$$

is the unique monic polynomial of degree $n+1$ with roots at the Chebyshev points $x_k = \cos\left(\frac{k\pi}{n}\right)$ for $k = 0 \ldots n$.

Note that in class we showed that the roots of the Chebyshev polynomials provide the optimal interpolation points for smooth functions based on the form of the error in the polynomial interpolant. However, we often use the locations of the extreme points (i.e., the Chebyshev points above) as the interpolation points. In this problem you show that

$$p(x) = \prod_{j=0}^{n} (x - x_j),$$

which appears in the form of the error of an interpolant. Thus the maximum of this polynomial goes down exponentially as the number of points increases.

(b) The barycentric form of the interpolating polynomial is

$$p(x) = \frac{\sum_{k=0}^{n} \frac{w_k}{(x - x_k)} f_k}{\sum_{k=0}^{n} \frac{w_k}{(x - x_k)}},$$

where

$$w_k = \left(\prod_{j \neq k} (x_k - x_j) \right)^{-1}.$$

Show that for the Chebyshev points, $x_k = \cos(k\pi/n)$, the interpolation formula can be written as

$$p(x) = \frac{\sum_{k=0}^{n} \frac{\hat{w}_k}{(x - x_k)} f_k}{\sum_{k=0}^{n} \frac{\hat{w}_k}{(x - x_k)}},$$

where

$$\hat{w}_k = \begin{cases} (-1)^k & 1 \leq k \leq n-1 \\ (-1)^k/2 & k = 0, n \end{cases}.$$

Hint: You can relate the polynomial from the previous part to w_k and obtain a formula for w_k.

2. On the last page of this assignment, there is a MATLAB function that takes as input vectors x and y, and returns an array, P, which contains the coefficients of the natural cubic spline through the points (x_i, y_i). The cubic spline is defined as $S(x) = S_i(x)$ for $x_i \le x \le x_{i+1}$, and

$$S_i(x) = P_{i1} + P_{i2}(x - x_i) + P_{i3}(x - x_i)^2 + P_{i4}(x - x_i)^3.$$

   (a) Write a routine that takes as input the arrays x and P and the scalar z and returns $S(z)$.

   (b) In the absence of derivatives at the endpoints, one commonly uses the *not-a-knot* boundary condition rather than natural boundary conditions. If the points are indexed from $j = 0 \ldots n$, then the not-a-knot condition requires that $S'''(x)$ be continuous at x_1 and x_{n-1}. For $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$, in class we derived the system of linear equations for the values of c_i:

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$$

   for $i = 1 \ldots n-1$, where $h_i = x_{i+1} - x_i$. We need two more equations to be able to solve for the c values. Derive the two equations that come from the not-a-knot conditions involving only c values and h values.

   (c) Write a routine to compute the coefficients of a cubic spline with not-a-knot boundary conditions. This requires only minor changes from the natural spline code provided.

   (d) Plot the natural and not-a-knot splines that interpolate $f(x) = \cos(2\pi x)$ at the points $x_j = j/5$ for $j = 0, \ldots, 5$. For each spline, compute maximum error in using the spline to approximate $f(x)$ between $x = 0$ and $x = 1$.

   (e) In class we saw that interpolating the function $f(x) = (25x^2 + 1)^{-1}$ on $[-1, 1]$ with a single polynomial using equally spaced points led to large oscillations which diverged as the number of points increased.

   - Make a plot of the interpolating cubic spline using 21 equally spaced points.
   - Compute the approximation error (in max norm) of the cubic spine approximation as a function of the number of points used for interpolation. Plot the error versus the number of points and discuss the convergence rate.

3. (a) Write a composite trapezoidal rule routine to compute an approximation to the integral of a function, f, over an interval $[a, b]$. Your routine should take as inputs: the integrand f, the endpoints a and b, and the number the number of subintervals n.

   (b) Write a composite Simpson's rule integration routine that takes the same inputs as your composite trapezoidal rule routine.

   (c) Write a composite 3-point Gaussian quadrature routine that takes the same inputs as your composite trapezoidal rule routine.

   (d) Apply each of these composite quadratures to approximate

$$\int_0^1 \mathrm{e}^x \, dx.$$

   Make a table of the results for $n = 2, 4, 8, 16, 32$, and a table of the errors. How is the order of accuracy demonstrated in the table of errors? Comment on your results.

4. On the last page of this assignment a MATLAB code for adaptive Simpson quadrature is given.

   (a) This code is written for clarity, not for efficiency. Explain why this program is not an efficient implementation of adaptive Simpson's rule.

   (b) Use the given adaptive Simpson's rule routine and MATLAB's build-in adaptive Simpson's quadrature, `quad`, to evaluate the the integral below with an error tolerance of 10^{-10}.

   $$\int_0^1 \frac{1}{x + 0.01}\, dx$$

   What are the actual errors in each case? How many function evaluations are necessary in each routine. Comment on your results.

   (c) Analytically derive a bound for the number of points needed to guarantee that the error is below 10^{-10} for the composite Trapezoidal rule and for the composite Simpson's rule each using equally spaced points for the integral above.

   (d) Experiment with your composite Simpson's rule code to estimate (within 1000 points) the actual number of points needed to get an error below 10^{-10}. Comment on your results taking into account the analytic bound on the number of points and the number of function evaluations used by the (efficient) adaptive routines.

```
% Natural spline coefficients
%
%  Input:   x - n by 1 vector of the nodes
%           y - n by 1 vector with the corresponding function values
%  Output:  P - n-1 by 4 matrix
%           the elements of the ith row of P give the coefficients
%           of the cubic between [x(i),x(i+1)] as
%           Si(x) =  P(i,1) + P(i,2)*(x-x(i))
%                  + P(i,3)*(x-x(i))^2 + P(i,4)*(x-x(i))^3
%
function P=naturalspline(x,y);

  % initialize P
  %
  n = length(x);
  P = zeros(n-1,4);

  % compute the distances between the points
  %
  h = x(2:n) - x(1:n-1);

  % set up the tridiagonal linear system to solve
  %
  d0 = 2*(h(1:n-2)+h(2:n-1));    % diagonal
  d1 = h(2:n-2);                 % super and sub diagonal

  % form the matrix and rhs
  %
  A = diag(d0,0) + diag(d1,-1) + diag(d1,1);
  b = 3*(y(3:n)-y(2:n-1))./h(2:n-1) - 3*(y(2:n-1)-y(1:n-2))./h(1:n-2);

  % solve the linear system
  %
  z = A\b;

  % append the natural boundary conditions
  %
  z = [0; z; 0];

  % compute the coefficients
  %
  P(:,1) =  y(1:n-1);
  P(:,2) = (y(2:n)-y(1:n-1))./h - h.*(z(2:n)+2*z(1:n-1))/3;
  P(:,3) =  z(1:n-1);
  P(:,4) = (z(2:n)-z(1:n-1))./(3*h);
```

```
% adaptive quadrature using Simpson's rule
%
%  input: f     - integrand, scalar function that takes one input
%         a,b   - integration is over interval [a,b]
%         ep    - error tolerance
%         count - number of times the integrand is evaluated
%                   this should not be passed in by the user, it
%                   is only used in recursive calls
%
%  output: q         - the approximation to the integral
%          new_count - number of times the integrand was evaluated
%
%  note: this function requires that the quadrature rule S be
%        defined in the same file
%
function [q,new_count]=adapt(f,a,b,ep,count);

  % if count was not passed in, initialize it to zero
  %
  if( nargin < 5 )
    count = 0;
  end

  new_count = count + 9;

  sab = S(f,a,b);

  c   = 0.5*(a+b);
  sac = S(f,a,c);
  scb = S(f,c,b);

  factor = 15;
  if( abs(sac+scb-sab) < factor*ep )
    q = sac + scb;
  else
    [q1,new_count]=adapt(f,a,c,ep/2,new_count);
    [q2,new_count]=adapt(f,c,b,ep/2,new_count);
    q =q1+q2;
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Simposon's Rule
%   input: f   - integrand, scalar function that takes one input
%          a,b - integration is over [a,b]
%
%   output: q  - simpson's rule approximation to integral of f over [a,b]
%
function q=S(f,a,b);
  q = (b-a)/6*( feval(f,a) + 4*feval(f,(a+b)/2) + feval(f,b));
```