CS 322: Assignment 2

Due: Monday, February 23, 2004, 4pm

Do not submit work unless you have adhered to the principles of academic integrity as descibed on the course website:

`http://www.cs.cornell.edu/Courses/cs322/2004sp/`

Points will be deducted for poorly commented code, redundant computation that seriously effects efficiency, and failure to use features of MATLAB that are part of the course syllabus. In particular, use vector operations whenever possible. Pay attention to the course website for news that relates to this assignment.

Problem A (8 pts) Big Dipper Evolution

The seven stars that make up the "Big Dipper" in the sky move slowly with time. In this problem you are given location data for each contributing star at five widely spaced times. Using polynomial interpolation you will generate a sequence of frames (i.e., a movie) that display the Big Dipper at a range of intermediate time values.

The first thing to do is to get the script `SHowBigDipper` off the course website and run it:

```
% Script ShowBigDipper
  clear
% Big Dipper Evolution

% The Big Dipper is a constellation consisting of
% seven stars which we index as follows:
%
%
%                 2 *      3*                 7 *
%       1 *                     4 *
%
%                                               6 *
%                                    5 *

% The location of these stars varies with time. We have data
% for times t(1),...,t(5) where

  t  = [-50; -25; 0; 25; 50];

% Think of t as (-50000BC, -25000BC, Now, 25000AD, 50000AD)

% The (x,y) coordinate of star j at time t(i) is specified by
% (x(i,j),y(i,j)) where

  x =[ 15.5   29.0   36.2   45.2   49.5   62.0  55.2;...
       16.0   28.2   34.9   44.6   49.2   61.5  56.5;...
       17.8   27.9   34.3   44.1   48.6   61.0  58.9;...
       18.5   27.5   33.9   43.8   48.5   60.5  61.2;...
       20.3   26.2   33.2   42.9   47.8   59.7  62.5];

  y = [ 0.0   5.0    5.1    5.5   0.0   6.7   14.8 ;...
       -0.4   5.3    4.3    5.2   0.0   4.6   14.3 ;...
        0.5   5.5    4.3    5.2   0.0   4.1   13.1 ;...
        0.4   5.4    4.3    5.7   0.0   4.4   12.7 ;...
       -1.2   5.5    4.4    6.2   0.0   5.0   13.2];

   << etc >>
```

We can use polynomial interpolation to trace a star as it moves across the xy-plane. A simple example illustrates the main idea. Suppose a star is at (u_i, v_i) at time $t = t_i$, $i = 1{:}m$. Construct degree $m-1$ polynomial interpolants p and q such that $p(t_i) = u_i$ and $q(t_i) = v_i$ for $i = 1{:}m$. We then can think of $(p(t), q(t))$ as an interpolating trajectory.

Using `ShowBigDipper` as a guide for how to produce a MATLAB movie, you are to implement the following function so that it performs as advertised:

```
  function M = BigDipperMovie(tVals)
% M is a movie of the Big Dipper Evolution
% tVals is a column n-vector
% M has n frames, the k-th frame depicts the Big Dipper
% at time tVals(k).
```

Proceed as follows

- The function body should begin with the assignments to `t`, `x` , and `y` that you see above. Just cut and paste from `ShowBigDipper`.
- Use the SCMV functions `InterpV` to generate the required polynomial interpolants. There are two for each star and each will have degree four since there are five time points. Use `HornerV` to evaluate these at `tVals`.
- Cut-and-paste the "frame-making" loop in `ShowBigDipper`. Modify it so that during the k-th pass through the loop it assigns to `M(k)` the frame that depicts the Big Dipper at time `tVal(k)`.

Test your implementation with the script `A2A` which is available on the course website. (Note: `A2A` uses the NCM function `bigscreen` which "supersizes" the figure window. Make sure that this function is reachable from your directory.) Submit your implementation of `BigDipperMovie` using the CMS. We will check it out by running `A2A`

Problem B (5 pts) Multiple Polynomial Interpolation

Suppose we are given nine data points $(x_1, y_1), \ldots, (x_9, y_9)$ and want to generate the cubic interpolants at points $\{x_1, x_2, x_3, x_4\}$, $\{x_2, x_3, x_4, x_5\}$, $\{x_3, x_4, x_5, x_6\}$, $\{x_4, x_5, x_6, x_7\}$, $\{x_5, x_6, x_7, x_8\}$, and $\{x_6, x_7, x_8, x_9\}$. We could just apply our usual interpolation "technology" six times to generate these polynomials. In this problem you discover a better way to do this by thinking carefully about the divided differences that are produced when the Newton representation is used. You are to implement efficiently the following function so that it performs as specified:

```
  function C = MultipleInterpN(x,y,m)
%
% Multiple Newton polynomial interpolants.
% x is a column n-vector with distinct components and y is
% a column n-vector.
% m is an integer that satisfies 1<=m<=n
% C is m-by-(n-m+1) matrix with  the property that it's k-th
% column houses the coefficients of the Newton interpolant
% of the m points
%                (x(k),y(k)),...,(x(k+m-1),y(k+m-1))
% I.e., for k = 1:n-m+1, if
%
%    p(x) = C(1,k) + C(2,k)(x-x(k))+ C(3,k)(x-x(k))(x-x(k+1)) +...
%                    ...+ C(m,k))(x-x(k))*...*(x-x(k+m-2))
% then
%                  p(x(k+i-1)) = y(k+i-1), i=1:m.
```

Start by reading §2.4.1 in SCMV paying particular attention to page 100 where there is a diagram of the difference table that underlies the implementation of `InterpN2`. Notice that the "top edge" of the difference table has the values $f[x_1]$, $f[x_1, x_2]$, $f[x_1, x_2, x_3]$, and $f[x_1, x_2, x_3, x_4]$ which are the coefficients of the Newton interpolant at $\{x_1, x_2, x_3, x_4\}$. Just below those values you find $f[x_2]$, $f[x_2, x_3]$, $f[x_2, x_3, x_4]$, and $f[x_2, x_3, x_4, x_5]$ which are the coefficients of the Newton interpolant at $\{x_2, x_3, x_4, x_5\}$. Thus, we only need the first four "columns" in the difference table to generate all the necessary cubic interpolants.

More generally, if we have n points and want all the "shifted" interpolants of degree $m-1$, then we just need the first m columns of the full difference table. Thus, your approach towards this problem should be one of generalizing

```
  function c = InterpN2(x,y)
% c = InterpN2(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is  a column n-vector with the property that if
%
%      p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))*...*(x-x(n-1))
% then
%      p(x(i)) = y(i), i=1:n.

  n = length(x);
  for k = 1:n-1
     y(k+1:n) = (y(k+1:n)-y(k:n-1)) ./ (x(k+1:n) - x(1:n-k));
  end
  c = y;
```

Indeed, if $m = n$ then `MultipleInterpN` is equivalent to `InterpN2`.)

Test your implementation with the function `ShowMultipleInterpN(n,m)` which is available on the course website. Submit `MultipleInterpN` via CMS and we'll see how it does on some modest values of n and m, e.g., $n = 16$, $m = 4$.

Problem C (7 pts) Efficient Trigonometric Interpolation

Read about trigonometric interpolation in SCMV §2.4.4. When we interpolate with linear combinations of sines and cosines we must solve a linear system. (Just as we do when we interpolate with linear combinations of 1, x, x^2,... in the polynomial interpolation setting.) The function

```
  function F = CSInterp(f)
%
% f is a column n vector and n = 2m.
% F.a is a column m+1 vector and F.b is a column m-1 vector so that if
% tau = (0:n-1)'*pi/m, then
%
%         f = F.a(1)*cos(0*tau) +...+ F.a(m+1)*cos(m*tau) +
%             F.b(1)*sin(tau)   +...+ F.b(m-1)*sin((m-1)*tau)
```

that is developed in §2.4.4 also solves a linear system, but it is very inefficient. In this problem you are to develop an equivalent function `newCSInterp(f)` which is much faster. There are three reasons why `CSInterp` is suboptimal:

- `CSInterp` solves a linear system of the form $Py = f$ using the \ operator, i.e., $y = P\backslash f$. This is an $O(n^3)$ operation. A better approach is to recognize that the matrix P has the property that $P^TP = D$ where D is a diagonal matrix, e.g.,

$$D = \begin{bmatrix} d_1 & 0 & 0 & 0 \\ 0 & d_2 & 0 & 0 \\ 0 & 0 & d_3 & 0 \\ 0 & 0 & 0 & d_4 \end{bmatrix}$$

  Note that $Py = f$ has the same solution as $(P^TP)y = P^Tf$, i.e., $Dy = P^Tf$. Since P^Tf is $O(n^2)$ and solving a diagonal system is $O(n)$, it is possible to reduce the linear system solving cost from $O(n^3)$ to $O(n^2)$. To implement this idea in `newCSinterp`, you will have to develop recipes for the diagonal components of P^TP. Find those recipes (they are very simple) by taking a look at P^TP for some small values of n. Proof not required.

- In `CSinterp` the n-by-n matrix P is build up by "concatenating" columns. This makes the MATLAB memory manager work hard as it has to reallocate space with every operation of the form

```
P = [ P nextCol].
```

It is more efficient to set `P = ones(n,n)` at the start and then "fill it up" column-by-column, e.g.,

`P(:,j) = nextCol`.

- The previous efficiency suggestion can be further improved by recognizing that in MATLAB we can apply the `sin` and `cos` functions to matrices. Hmmm. What does the matrix `(1:5)'*(1:3)` look like? Can you set up `P` without any loops?

Submit your implementation of `newCSINterp` via CMS. We'll see what happens when we run the test script `A2C` which is available on the course website for you to use as you develop `newCSInterp`. `A2C` compares execution times for `CSInterp` and `newCSInterp`. Note, `A2C` invokes the SCMV function `CSEval`, so make sure that is accessible.