MATH2070: LAB 8: Higher Order Interpolation

Introduction	Exercise 1
Parametric Interpolation	Exercise 2
Cubic Hermite Interpolation	Exercise 3
	Exercise 4
Mesh generation branch (do only one branch)	
Two-dimensional Hermite interpolation and mesh generation	Exercise 5
Matching patches	Exercise 6
	Exercise 7
	Exercise 8
	Exercise 9
Spline interpolation branch (do only one branch)	
Cubic spline interpolation	Exercise 10
Splines without derivatives	Exercise 11
Monotone interpolation	Exercise 12
	Exercise 13
	Exercise 14
	Exercise 15
Extra credit	
Extra credit: Bilinear interpolation in two dimensions	Extra credit exercise

1 Introduction

Piecewise linear interpolation has many good properties. In particular, if the data come from a continuously differentiable function $f(x)$ and if the data points are suitably spread throughout the closed interval, then the interpolant converges to the function. The resulting interpolant consists of straight line segments and so has flats and kinks in it, making it inappropriate for many applications.

In this lab you will look at several examples of piecewise polynomial interpolation with continuous derivatives from place to place. You will be looking first at piecewise Hermite cubic interpolation.

In this lab, after completing the sections on Parametric Interpolation and Cubic Hermite Interpolation and completing Exercises 1 through 4, you will be given a choice of two branches. You need to do only one of the branches to be given full credit for this lab. If you choose to do *both* branches, the second branch will be graded as a 10 point extra credit exercise. This is in addition to the 8 point extra credit exercise at the end of the lab. The two branches are:

Mesh generation Read the following sections

- Two-dimensional Hermite interpolation and mesh generation
- Matching patches

and do Exercises 5 through 9.

Spline interpolation Read the following sections

- Cubic spline interpolation
- Splines without derivatives
- Monotone interpolation

and do Exercises 10 through 15

This lab will take three sessions. If you print this lab, you may prefer to use the pdf version.

2 Parametric Interpolation

You are familiar with a *parameterized* curve, in which a special variable, often called t or s, is used to draw objects for which the relationship between x and y may not be functional. For example, one simple definition of a circle is:

$$\begin{aligned} x &= \cos t \\ y &= \sin t \end{aligned} \tag{1}$$

for $0 \le t \le 2\pi$. Writing the circle this way involves regarding both x and y as functions of the independent variable t. (There is no way to write y as a single function of x, because taking square roots introduces "+" and "−" options.)

Suppose we wanted to do interpolation of some complicated curve? If we have a drawing of the curve, we could simply mark points along the curve that are roughly evenly spaced, and make tables of `tdata`, `xdata` and `ydata`, where the values of `tdata` could be `1, 2, 3, ...`.

Now if we want to compute intermediate points on the curve, that's really saying, for a given value of `tdata`, what would the corresponding values of `xdata` and `ydata` be? This means doing two interpolation problems, one for `xdata` as a function of `tdata` and the second for `ydata` as a function of `tdata`.

In the first exercise, you will be dealing with a slightly more complicated curve than a circle. You may recall that a cardiod is a roughly heart-shaped closed curve, but it has no sharp point at the bottom. You can find a nice description of cardioid curves with various ways of defining them at `http://mathworld.wolfram.com/Cardioid.html` That article also remarks that a cardioid is a special case of a limaçon, and parametric equations for a limaçon can be given by

$$\begin{aligned} r &= r_0 + \cos t \\ x &= r \cos t \\ y &= r \sin t. \end{aligned} \tag{2}$$

Exercise 1: Try the following method for plotting a limaçon. Use `eval_plin.m` from last lab, or you can download `eval_plin.m` and `bracket.m` from the web page. Copy the following code to a script m-file named `exer1.m`

```
% generate the data points for a limacon
nintervals = 20;
tdata = linspace ( 0, 2*pi, nintervals + 1 );
r = 1.15 + cos ( tdata );
xdata = r .* cos ( tdata );
ydata = r .* sin ( tdata );

% interpolate them
tval = linspace ( 0, 2*pi, 10*(nintervals+1) );
xval = eval_plin ( tdata, xdata, tval );
yval = eval_plin ( tdata, ydata, tval );

% plot them with correct aspect ratio
plot ( xval, yval )
axis equal
```

You do not need to send me this plot.

You can see that the plot of the limaçon in Exercise 1 is not very smooth. This is because the line segments are straiaght lines that meet at corners. In the following section you will see one way of interpolating curves using curved sections whose derivatives are continuous so that there are no corners.

3 Cubic Hermite Interpolation

Hermite interpolation is discussed by Quarteroni, Sacco, and Saleri in Section 8.5.

If we were trying to design, say, the shape of the sheet metal pattern for a car door, kinks and corners would not be acceptable. If that's a problem, then perhaps we could try to make sure that our interpolant is smoother by matching *both* the function value and the derivative at each point. *(This assumes you can get the derivative value.)*

Suppose we have points x_k, $k = 1, 2, \ldots, N$ and a differentiable function $f(x)$, so that the values $y_k = f(x_k)$ and $y'_k = f'(x_k)$ can be regarded as data points.

Now consider the situation in the k^{th} interval, $[x_k, x_{k+1}]$. There is a unique cubic polynomial on this interval that takes on the function and derivative values y_k and y'_k at the two endpoints. You may recall that Lagrange interpolation polynomials could be used to write `eval_plin` in the previous lab. On the interval $[x_k, x_{k+1}]$ there are two special linear polynomials: $\ell_0(x)$ is 1 at x_k and zero at x_{k+1}, and $\ell_1(x)$ is 1 at x_{k+1} and zero at x_k. This feature makes it easy to construct an arbitrary linear polynomial that matches the values y_k and y_{k+1} at the endpoints as $p_{\text{linear}} = y_k\ell_0(x) + y_{k+1}\ell_1(x)$. For this lab, we want to match *both* the function *and* derivative values at the endpoints, so we will need cubic Hermite polynomials instead of linear ones.

Consder the four Hermite polynomials

$$\begin{aligned}
h_1(x) &= \frac{(x - x_{k+1})^2(3x_k - x_{k+1} - 2x)}{(x_k - x_{k+1})^3} \\
h_2(x) &= \frac{(x - x_k)(x - x_{k+1})^2}{(x_k - x_{k+1})^2} \\
h_3(x) &= \frac{(x - x_k)^2(3x_{k+1} - x_k - 2x)}{(x_{k+1} - x_k)^3} \\
h_4(x) &= \frac{(x - x_{k+1})(x - x_k)^2}{(x_{k+1} - x_k)^2}
\end{aligned}$$

These four cubic polynomials satisfy the following equalities, that are similar to Quarteroni, Sacco, and Saleri, p349.

	x_k	x_{k+1}
$h_1(x)$	1	0
$h'_1(x)$	0	0
$h_2(x)$	0	0
$h'_2(x)$	1	0
$h_3(x)$	0	1
$h'_3(x)$	0	0
$h_4(x)$	0	0
$h'_4(x)$	0	1

(3)

(This table means that, for example, $h_2(x_k) = 0$, $h'_2(x_k) = 1$, $h_2(x_{k+1}) = 0$, and $h'_2(x_{k+1}) = 0$.) If you know Mathematica, Maple or the symbolic toolbox in Matlab, you can check these equalities easily enough. For the purpose of this lab, the easiest one to check is h_2, so check it by hand. (Hint: Do not multiply the factors out! You can check by staring at the formula long enough.)

Given these four polynomials, the unique polynomial with the values

$$\begin{aligned} y_k &= f(x_k) \\ y'_k &= f'(x_k) \\ y_{k+1} &= f(x_{k+1}) \\ y'_{k+1} &= f'(x_{k+1}) \end{aligned}$$

can be written as

$$p(x) = y_k h_1(x) + y'_k h_2(x) + y_{k+1} h_3(x) + y'_{k+1} h_4(x) \tag{4}$$

in the interval $x_k \leq x \leq x_{k+1}$. It is easy to see that both p and p' are continuous at the points $x = x_k$ so that p is C^1.

We are now in a position to write a piecewise Hermite interpolation routine. This routine will be similar to `eval_plin.m`.

Exercise 2:

(a) Begin a Matlab function m-file called `eval_pherm.m` with the signature

```
function yval = eval_pherm ( xdata, ydata, ypdata, xval )
% yval = eval_pherm ( xdata, ydata, ypdata, xval )
% comments

% your name and the date
```

and insert comments. the values in `ypdata` are the derivative values at the points `xdata`.

(b) Use the `bracket` function to find the interval in which `xval` lies. As in `eval_plin.m`, this can be done in a single line.

(c) Evaluate the four Hermite functions, $h_1(x)$, $h_2(x)$, $h_3(x)$, and $h_4(x)$ at `xval`. Use componentwise operations if you can or use loops.

(d) Complete `eval_pherm` with the evaluation of `yval` using the expression (4) for the Hermite interpolating polynomial.

(e) Check your work by using `eval_pherm` four times, once for each of the polynomials $y = 1$, $y = x$, $y = x^2$, and $y = x^3$ interpolated for the interval `xdata=[0,2]`. Each of these four interpolants should be checked at four points of your choice. You can choose integers for these points, either inside or outside `[0,2]` because points outside `[0,2]` are evaluated using the same formula as those inside. Recall that four points uniquely determine a cubic polynomial, so if you get agreement within roundoff at four points, you know your code is correct.

If your code is not correct, you can debug in the following way. You only need to do the following steps if you are debugging.

i. For this test problem, there is only one interval, so the result of `bracket` should be the vector `ones(size(xval))`. You can use the debugger to confirm this fact. This eliminates `bracket.m` as the source of the bug.

ii. If you are using elementwise operations, make sure all your multiplications and divisions are preceeded by a dot.

iii. Try the four polynomials $y = 1$, $y = x$, $y = x^2$, and $y = x^3$ on the interval `xdata=[0,1]` (not `[0,2]`) one at a time. The reason to change the interval to `[0,1]` is that the length of this interval is 1 so the denominators in the definitions of the h_k polynomials all become 1. If all four polynomials work but `eval_pherm` does not, odds are you have an error in the denominators of one or more of the $h_k(x)$.

iv. If you cannot get agreement for the polynomials $y = 1$, $y = x$, *etc.*, on `[0,1]`, then one or more of your $h_k(x)$ is probably wrong. Use `eval_pherm` to reproduce each of the $h_k(x)$ for `xdata=[0,1]`. The table (3) above provides `ydata` and `ypdata` values. Check your results at the points $x = 0$ and $x = 1$ first, and then for the points $x = 0.25$ and $x = 0.5$ (you will have to use the formulæ to get the data values. If these all agree but `eval_pherm` still does not work correctly, then your implementation of (4) is incorrect.

v. If you have completed the previous two steps and fixed all bugs you found, but you still cannot interpolate the monomials $y = 1$, $y = x$, *etc.*, on `[0,2]`, then one or more of your $h_k(x)$ is still wrong. Try interpolating each of the four Hermite polynomials on `xdata=[0,2]`. When you are finished you will surely have found your bugs.

Now, we will use `eval_pherm.m` to see how piecewise Hermite converges.

Exercise 3:

(a) First, modify your Matlab m-file `runge.m` so that it returns both the Runge function $y = 1/(1+x^2)$ and its derivative y'. I will leave it to you to compute the derivative expression, but be sure you are correct. The signature of the function should be

```
function [y,yprime]= runge ( x )
% [y,yprime]= runge ( x )
% comments
```

and it should have appropriate comments. Do not forget to use componentwise syntax.

(b) Copy your `test_plin_interpolate.m` function from last lab, or download my version. Rename it to `test_pherm_interpolate.m` and modify it to use `eval_pherm.m`.

(c) Using equally spaced data points from 0 to 5 for `xdata` and the `runge.m` function to compute `ydata` and `ypdata`. Estimate the maximum interpolation error by computing the maximum relative difference (infinity norm) between the exact and interpolated values at 4001 evenly spaced points. Fill in the following table:

```
 Runge function, Piecewise Hermite Cubic
ndata =  5  Err(  5) = ________
ndata = 11  Err( 11) = ________  Err(  5)/Err( 11) = ________
ndata = 21  Err( 21) = ________  Err( 11)/Err( 21) = ________
ndata = 41  Err( 41) = ________  Err( 21)/Err( 41) = ________
ndata = 81  Err( 81) = ________  Err( 41)/Err( 81) = ________
ndata =161  Err(161) = ________  Err( 81)/Err(161) = ________
ndata =321  Err(321) = ________  Err(161)/Err(321) = ________
ndata =641  Err(641) = ________  Err(321)/Err(641) = ________
```

(d) Estimate the rate of convergence by examining the ratios `Err(41)/Err(81)`, *etc.*, and estimating the nearest power of two. You should observe the theoretical convergence rate. (The rate is larger than 2.)

Now, you are in a position to apply Hermite interpolation to generate a very smooth limaçon.

Exercise 4:

(a) Copy your script m-file `exer1.m` from Exercise 1 and rename it to `exer4.m`.

(b) Differentiate the equations (2) defining the cardiod and add expressions for `xpdata` and `ypdata` to `exer4.m`.

(c) Replace `eval_plin` with `eval_pherm`.

(d) Plot the limaçon. If this plot does not appear smooth or if you can see where the boundaries of the pieces that make it up lie, you probably have an error in your derivatives of the equations (2) Similarly, wiggles and extraneous loops also indicate errors, probably in the derivatives. Please include your corrected plot with your summary file.

4 Two-dimensional Hermite interpolation and mesh generation

In order to solve a partial differential equation, the region over which it is to be solved must be broken into small pieces. These pieces are often called "mesh elements," and the process of generating them for arbitrary regions is called "mesh generation." You can see some examples of complicated meshes at `http://www.geuz.org/gmsh/gallery/spirale.gif` The process of building the outline of a complicated assembly and then using it to generate a mesh is very labor-intensive and can take weeks. The discussion in this lab is most relevant to generation of quadrilateral and hexahedral meshes, not triangular or tetrahedral meshes.

The lowest-level activity in generating meshes in many commercial packages is defining "Coons Patches." These are two- or three-dimensional geometrical entities that can, on the one hand, be combined together smoothly, and, on the other hand, easily be subdivided into mesh elements. These Coons Patches are generally built using bicubic Hermite polynomials in two dimensions, and tricubic Hermite polynomials in three dimensions. We will be focussing on two dimensions in this lab.

Coons patches are also used in computer graphics, but I am not familiar enough with this application to discuss it.

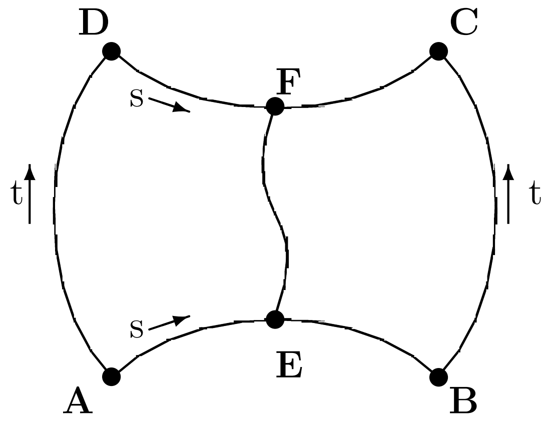
One approach to generating two-dimensional meshes is to take a drawing of the object and cover it with a modest number of patches, each with four curved sides. These patches generally are chosen to match up (to a reasonable approximation) with the boundaries of the object and fit against each other and are no smaller than necessary for these tasks. Once the object in the drawing is completely covered with patches, the patches themselves are broken into smaller mesh pieces. It is important that the mesh lines do not undergo abrupt changes across patch boundaries and it is also important that the user be able to modify the density of mesh lines and to concentrate them near, for example, one boundary. Commercial programs such as Patran and Ansys use this approach.

Definition A two-dimensional bicubic Hermite patch (Coons patch) is a smooth map from the unit square $(s,t) \in [0,1] \times [0,1]$ to a region $(x(s,t), y(s,t)) \in \mathbb{R}^2$. This mapping is a cubic polynomial in s for each fixed t and a cubic polynomial in t for each fixed s and can be written as a linear combination of the sixteen terms $s^m t^n$ for $n, m = 0, 1, 2, 3$.

A similar definition holds for a two-dimensional patch on a surface in $\mathbb{R}^3$, with an additional function $z(s,t)$. The extension of the definition of bicubic Hermite patches to tricubic Hermite patches in three dimensions is straightforward.

Remark A mesh consisting of curved-sided quadrilaterals can be constructed from the union of several bicubic Hermite patches that are adjacent to one another and whose parameterizations along adjacent edges agree. Dividing the square $[0,1] \times [0,1]$ into rows and columns of smaller squares and mapping the smaller squares using the bicubic Hermite mappings results in a computational mesh.

Consider the following sketch.



The four outer curves, AB, BC, DC, and AD will form the boundary of the patch. The two curves AB and DC are parameterized using Hermite cubics in s (with positive direction shown in the figure) and the curves BC, AD and each of the intermediate curves such as EF are parameterized using Hermite cubics in t (with positive direction shown in the figure). Obviously, the coordinates of each of the points A, B, C, and D will be needed. Furthermore, since these are Hermite cubics, the derivatives of $x(s,t)$ and $y(s,t)$ with respect to both s and t will be needed at each of the four corners. Finally, the cross-derivatives $\partial^2 x/\partial s\partial t$ will be needed at each of the four corners. These sixteen data are needed to construct the Coons patch since, in general, a bicubic polynomial is a sum of the sixteen terms $s^m t^n$ for $n, m = 0, 1, 2, 3$.

Among these sixteen data, there is enough information to construct the boundary curve AB parametrically using Hermite polynomials for $x(s,0)$ and $y(s,0)$, where s is the indepentent variable increasing from point A to point B. This interpolation can be done using the `eval_pherm.m` function you wrote earlier. Similarly, the curve DC can be constructed using Hermite polynomials to give $x(s,1)$ and $y(s,1)$. Lines of constant s, such as the line EF, can be constructed using the curves AB and DC. The result will be the parametric coordinate functions $x(s,t)$ and $y(s,t)$ for all $(s,t) \in [0,1] \times [0,1]$. This is the approach taken in the following exercise.

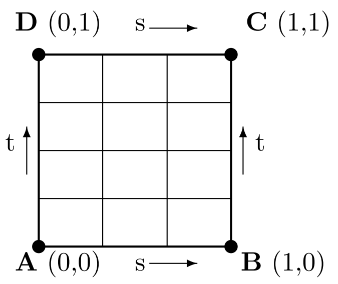
In the following exercise you will write a script m-file that fills a square with smaller squares. This is a special case of mesh generateion. In doing this exercise, we will *not* use any special facts about squares, but will be writing as if the lines were curved. In subsequent exercises, we will see what meshes with curved lines look like.

The four corners, A, B, C, and D, each require data for x and y regarded as functions of s and t. The data are

x	y
$\frac{\partial x}{\partial s}$	$\frac{\partial y}{\partial s}$
$\frac{\partial x}{\partial t}$	$\frac{\partial y}{\partial t}$
$\frac{\partial^2 x}{\partial s\partial t}$	$\frac{\partial^2 y}{\partial s\partial t}$

To see to what these quantities refer, consider the point A. The value of x at point A means its abscissa. The quantity $\frac{\partial x}{\partial s}$ means the derivative of x with respect to the parameter s and is the answer to the question, "How rapidly does x change moving along the curve as s increases from point A?" The quantity $\frac{\partial x}{\partial t}$ is similar, except with respect to the parameter t. Finally, $\frac{\partial^2 x}{\partial s\partial t}$ is the cross derivative. These cross derivatives can be difficult to calculate "in your head" but reasonable meshes can often be generated by taking both $\frac{\partial^2 x}{\partial s\partial t}$ and $\frac{\partial^2 y}{\partial s\partial t}$ to be zero.

Exercise 5: Generate a script m-file named `exer5.m` with the following commands, replacing the `???` with correct expressions. Running this script should produce a square divided into three rectangles horizontally and four rectangles vertically, as shown here



Please include a copy of your plot with the files you send to me.

Hints:

- Read through the entire file before making any changes.
- In this simple case, the mapping $(s, t) \mapsto (x(s, t), y(s, t))$ is the identity mapping.
- To see what quantities such as `dxdsB` should be, answer the question, "As you move along the line from `A` to `B`, how is the coordinate `x` varying as you get to `B`?"
- If you are having trouble getting it to look right, first plot the outline (the lines with $t = 0$, $t = 1$, $s = 0$, and $s = 1$). Then look at the corner points one at a time. You likely have some of them right and others wrong. Fix the wrong ones, one at a time. Then add the lines with varying s, and finally add the lines with varying t.
- *Never* make more than one change to the script before looking at the resulting plot!

```
% Generate a mesh based on Hermite bicubic interpolation

% values for point A
xA       = 0;        yA       = 0;
dxdsA    = 1;        dydsA    = 0;
dxdtA    = 0;        dydtA    = 1;
d2xdsdtA = 0;        d2ydsdtA = 0;

% values for point B
xB       = 1;        yB       = 0;
dxdsB    = ???       dydsB    = ???
dxdtB    = 0;        dydtB    = 1;
d2xdsdtB = 0;        d2ydsdtB = 0;

% values for point C
xC       = 1;        yC       = 1;
dxdsC    = 1;        dydsC    = 0;
dxdtC    = ???       dydtC    = ???
d2xdsdtC = 0;        d2ydsdtC = 0;

% values for point D
xD       = 0;        yD       = 1;
dxdsD    = 1;        dydsD    = 0;
dxdtD    = 0;        dydtD    = 1;
d2xdsdtD = ???       d2ydsdtD = ???
```

```
% Start off with 4 points horizontally,
% and 5 points vertically
s=linspace(0,1,4);
t=linspace(0,1,5);

% interpolate x along bottom and top, function of s
xAB   =eval_pherm([0,1],[xA,xB]      ,[dxdsA,dxdsB],      s);
dxdtAB=eval_pherm([0,1],[dxdtA,dxdtB],[d2xdsdtA,d2xdsdtB],s);
xDC   =???
dxdtDC=???

% interpolate y along bottom and top, function of s
yAB   =???
dydtAB=???
yDC   =???
dydtDC=eval_pherm([0,1],[dydtD,dydtC],[d2ydsdtD,d2ydsdtC],s);

% interpolate s-interpolations in t-direction
% if variables x and y already exist, they might have
% the wrong dimensions.  Get rid of them before reusing them.
clear x y
for k=1:length(s)
  x(k,:)=eval_pherm([0,1],[xAB(k),xDC(k)],[dxdtAB(k),dxdtDC(k)],t);
  y(k,:)=???
end

% plot all lines
plot(x(:,1),y(:,1),'b')
hold on
for k=2:length(t)
  plot(x(:,k),y(:,k),'b')
end
for k=1:length(s)
  plot(x(k,:),y(k,:),'b')
end
axis('equal');
hold off
```

It seems natural that s and t would vary linearly along the sides, but that is not necessary. In the following exercise, you will see how nonlinear variation of s can be used to vary the mesh distribution without changing the outline of the figure. So long as s and t vary smoothly and map onto the square $[0,1] \times [0,1]$, they are essentially arbitrary.

Exercise 6:

(a) Copy your `exer5.m` to `exer6.m` and change it to generate 10 rectangular elements horizontally and 15 vertically (11 points horizontally, 16 points vertically).

(b) Modify $\partial x/\partial s$ at points A and D to be 3.0 and at points B and C to be 0.1. Leave all other values alone. You should see that the mesh elements become much thinner as they get closer to the right side. This is particularly valuable in aeronautical calculations because the “boundary layer” is remarkably thin near an aircraft's skin and mesh elements must be very thin there. Please send me this plot with your work.

Just because you can correctly generate a straight-line mesh does not mean your code is correct. In the next two exercises, you are going to continue debugging your Matlab programming by testing with two more difficult shapes. The first shape is a quadrilateral with straight sides, and the subsequent exercise uses curved sides.

Exercise 7: Copy the file `exer5.m` to a file `exer7.m` (or you can use "Save as" in the File menu). Modify it in the following ways:

- Change the points so that point A is (-1,0) and point C is (1,0.5).
- Change the number of points for s to 20 and the number of points for t to 15.
- Choose the slopes of the lines so that the *boundary* lines are straight with mesh points uniformly distributed along them. For example, both values `dxdtA` and `dydtA` should be 1 because both x and y increase by 1 as t goes from 0 to 1 starting at point A and ending at point D.
- Change the values `d2xdsdt=-1` and `d2ydsdt=-0.5` at all four corner points.

All of the resulting mesh lines (both boundary and interior) should be straight lines with uniformly spaced divisions. Please include a plot of your mesh with the files you send to me.

Hint: If you cannot get it to look right, try the following strategy.

(a) Starting from `exer5.m`, make the three changes:

  i. Change the coordinates of point A to (-1,0) and of point C to (1,0.5);
  ii. Change the number of points of `s` and `t` to 20 and 15, respectively; and,
  iii. Change the values of `d2xdsdt` and `d2ydsdt` to -1 and -0.5 at all four corners.

  Then plot the resulting figure. You will see that some boundary lines are straight and some are not. One of these curved lines is AD.

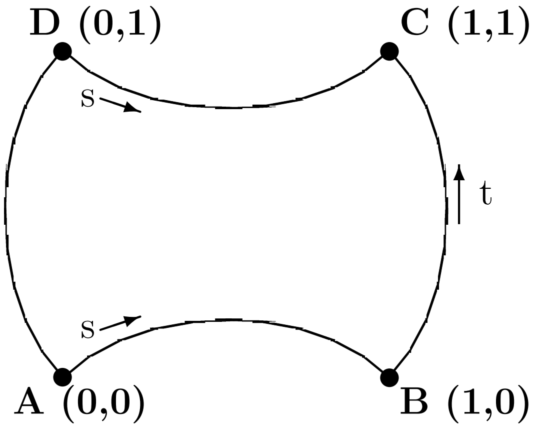
(b) Focus on the line AD. This is a "t-line." Take a piece of paper and use it as a straight edge to see what the line AD should look like. You should see that the slope of the line at A is about right, but the slope at D is not. Since the line AD is supposed to be straight, what should `dxdt` be at D? Put your estimate of `dxdtD` into your script and run it again. If your estimate is correct, the line AD is a straight line.

(c) Similarly, fix the other curved boundaries.

(d) Look at the line BC. It is straight, but the mesh spacing is not uniform. Since this is a "t-line," and since it is vertical, values of `dydtB` and `dydtC` may need to be adjusted.

(e) Similarly, adjust the other boundaries if their mesh spacing is not uniform.

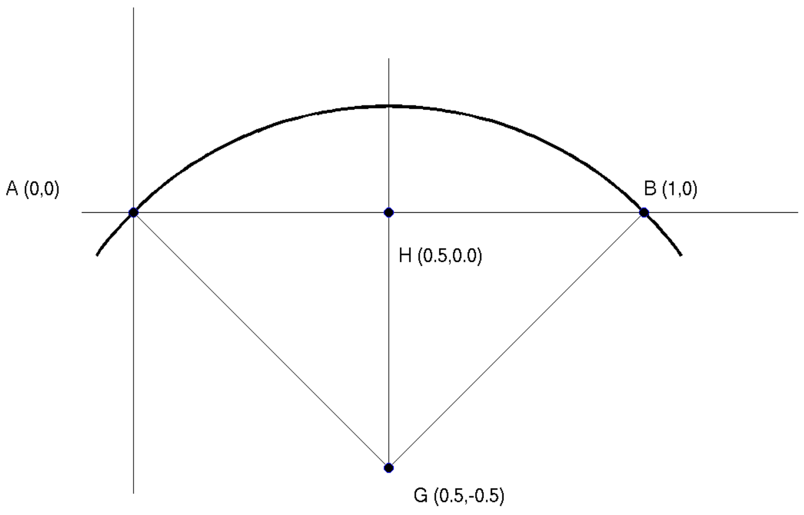
Exercise 8: Consider the following sketch:



Regard each of the four curves as quarter-circles, so the slopes of each boundary curve at each corner are $\frac{dy}{dx} = \pm 1$. (Do not confuse this with $\frac{dx}{ds}$ or $\frac{dy}{ds}$.) You can see that the four corner points are at the corners of the same unit square you used in Exercise 5.

(a) Addressing the bottom curve first, the curve AB is a quarter-circle (an arc with angle $\pi/2$). It is hard to guess $\frac{dx}{ds}$ and $\frac{dy}{ds}$ of this arc at its ends, so in this part of the exercise you will plot the curve parametrically and compute the derivatives. Write a simple set of parametric equations, analogous to (1), for x and y in terms of the variable $s \in [0, 1]$.

You may find the following figure helpful in generating the parametric equations.



As specified, the arc from A to B is a quarter-circle. To see how the coordinates of the center of the circle, point G, were found, consider the following steps. Note that the circle does not have unit radius.

- The x-coordinate of G must be 0.5, by symmetry of the figure.

- The line GH is vertical, parallel to the y-axis.
- The arc is a quarter-circle, so the angle AGB is $\pi/2$.
- Hence, the angle AGH is $\pi/4$, and the hence lines AG and BG have the same length and G must be the center of the circle.

(b) Check your parametric equations by writing a script m-file `exer8a.m` to plot the quarter-circular arc as a function of the parameter $s \in [0, 1]$. (The parameter s is *not* the arc length.) You should see an arc AB very similar to the one shown above. From your parametric equations, what are $\frac{dx}{ds}$ and $\frac{dy}{ds}$ evaluated at the points A and B?

(c) Write a script m-file `exer8b.m` based on `exer5.m` but only for the AB (bottom) portion of the outer boundary. Use 25 points for the `s` values.

- Use the values of x, dx/ds and dy/ds at A and B that you found above.
- Plot the arc at 25 points, with marks showing the intermediate points. You can use a command such as

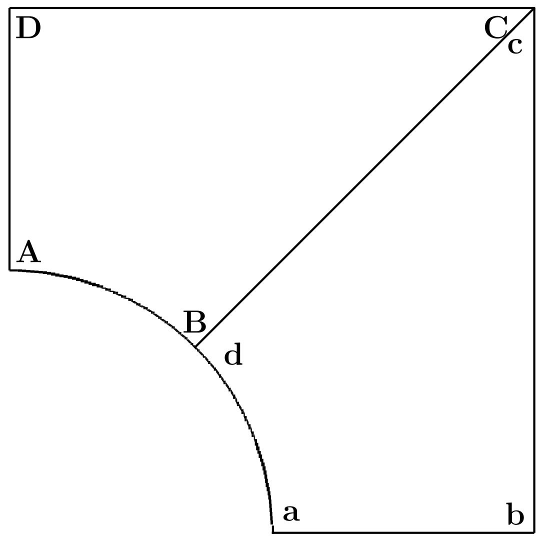
```
plot(x,y,'r*-') %color red, * at each data point, lines between *
```

- Use `exer8a.m` to plot the quarter-circle on the same plot. You should see 25 points that appear to be uniformly distributed along a red arc connecting the points A and B. The red arc should be close to, but not on top of, the quarter-circle from `exer8a.m`. Please include this plot when you send me your work.

(d) Copy the file `exer5.m` to another file `exer8.m` and use it to generate an 25×25 mesh on the curved-sided quadrilateral. Your mesh should clearly show two-fold symmetry (left-right and top-bottom), and the mesh intervals along the perimeter should be roughly uniform. Please include a plot of your mesh with the files you send to me. **Hint**: You already know the parameters for the bottom boundary (AB). Insert these into the file and plot to be sure you have them right. Ignore the other boundaries and the interior lines. Once you have AB correct, put the correct parameters into the BC line, plot to be sure, and continue.

5 Matching patches

So far, you have not seen a reason for using Hermite cubic interpolation over any other method. There is a good reason for this choice, namely, that two patches can be placed adjacent to each other and, if the derivatives at the endpoints of the common side are given the same values, the mesh will smoothly transition from one to the other. In the following exercise, there are two patches that touch along one edge. Consider the following sketch.



Point	Coordinates
A	$(0,1)$
B	$(\sqrt{2}/2, \sqrt{2}/2)$
C	$(2,2)$
D	$(0,2)$
a	$(1,0)$
b	$(2,0)$
c	$(2,2)$
d	$(\sqrt{2}/2, \sqrt{2}/2)$

Exercise 9: In this exercise, you will generate a mesh for the two-patch region in the sketch by generating meshes for the two patches separately and seeing that they naturally match up. The inner boundary in the sketch is circular, but we will be using a Hermite cubic approximation, which is pretty good. If more accuracy were needed for this circular boundary, more patches could be used.

(a) Copy the following commands for patch ABCD to a file `exer9a.m`.

```
% Generate a mesh for the patch ABCD
sqrt2on2=sqrt(2)/2;

% values for point A
xA       = 0;          yA       = 1;
dxdsA    = pi/4;       dydsA    = 0;
dxdtA    = 0;          dydtA    = 1;
d2xdsdtA = 0;          d2ydsdtA = 0;

% values for point B
xB       = sqrt2on2;   yB       = sqrt2on2;
dxdsB = sqrt2on2*pi/4; dydsB    = -sqrt2on2*pi/4;
dxdtB    = 2-sqrt2on2; dydtB    = 2-sqrt2on2;
d2xdsdtB = 0;          d2ydsdtB = 0;

% values for point C
```

```
xC        = 2;           yC        = 2;
dxdsC     = 2;           dydsC     = 0;
dxdtC     = 2-sqrt2on2;  dydtC     = 2-sqrt2on2;
d2xdsdtC  = 0;           d2ydsdtC  = 0;

% values for point D
xD        = 0;         yD        = 2;
dxdsD     = 2;         dydsD     = 0;
dxdtD     = 0;         dydtD     = 1;
d2xdsdtD  = 0;         d2ydsdtD  = 0;
```

(b) Using your earlier exercises as a model, add appropriate commands to complete `exer9a.m` so that it generates and plots a 20×30 mesh on patch ABCD.

(c) Make a copy of `exer9a.m` and call it `exer9c.m`. Add a second set of commands to `exer9c.m` so that it generates a 20×30 mesh on patch abcd. Make sure that the number of points along edge BC is the same as the number of points along edge dc, so that the mesh lines are continuous. Your mesh should be symmetric about the line BC (dc). Please include the plot of the combined meshes on these two patches when you send your work to me.

(d) Hermite cubics will generate smooth mesh lines if the outline of the region is smooth. The outer boundary DCcb has a corner, and the interior mesh lines clearly echo that corner. Make a copy of `exer9c.m`, call it `exer9d.m`, and "round the corner off" by moving the points C and c from (2,2) to (1.8,1.8), and making the boundary lines at point C (and c) have slope -1. It doesn't matter too much what parametric slopes `dxdsC` and `dydsC` (and `dxdsc` and `dydsc`) you use, so long as the ratio is $dy/dx = (dy/ds)/(dx/ds) = -1$, but parametric slopes of ± 0.5 generate a nice picture. Please include this plot when you send me your work.

6 Cubic spline interpolation

One disadvantage of the Hermite interpolation scheme is that you need to know the derivatives of your function. But it's very possible that you don't have any formula for your data, just the values at the data points. A second problem is the Hermite interpolant is smooth, but not as smooth as it could be. The discontinuities in the second derivative at the data points might be noticeable.

Cubic splines are piecewise cubic interpolants that are very smooth. They are continuous, with continuous first *and* second derivatives. Only the third derivative is allowed to jump at the join points (called knots).

Suppose you have broken an interval $[a, b]$ into subintervals

$$a = x_1 < x_2 < \ldots < x_n = b$$

then $s(x)$ is a "spline function" if there is a value $m \geq 1$ with

1. $s(x)$ is a polynomial of degree $\leq (m-1)$ on each subinterval $[x_{k-1}, x_k]$; and,
2. $\frac{d^r s}{dx^r}(x)$ is continuous on $[a, b]$ for $0 \leq r \leq (m-2)$.

Given data values y_k at each of the "knots" x_k, we are interested in interpolating the data, so that $s(x_k) = y_k$.

The continuity and interpolation conditions are almost enough to specify the cubic. If we add one extra condition at each endpoint, such as the value of the derivative, then the spline is determined. Unlike Hermite interpolation, the derivatives are needed only at the endpoints of the curve, not at the endpoints of each subinterval.

As with our other interpolation methods, we will first compute the parameters of the spline interpolant. Given those parameters, we will compute the first derivatives $s'(x_k)$ at the knots, and use the `eval_pherm` function you have already written to evaluate the spline function.

Quateroni, Sacco, and Saleri, on pp. 357ff, describe how to find the second derivatives (M or s'') of the "complete" cubic spline interpolant. (this work uses indices starting with $k = 0$, but we will obey Matlab's convention and start indices with $k = 1$.)

Since the desired spline is a piecewise C^2 cubic polynomial, its second derivative is continuous. Thus the values $M_k = s''(x_k)$ are well-defined. Since $s(x)$ is a piecewise cubic polynomial, its second derivative $s''(x)$ is piecewise linear. Thus,

$$s''(x) = \frac{(x_{k+1} - x)M_k + (x - x_k)M_{k+1}}{h_k} \text{ for } k = 1, \dots, n-1$$

For each of the intervals $[x_k, x_{k+1}]$, this expression can be integrated twice to yield

$$s(x) = \frac{(x_{k+1} - x)^3 M_k + (x - x_k)^3 M_{k+1}}{6h_k} + C_k(x_{k+1} - x) + D_k(x - x_k),$$

where C_k and D_k are constants of integration. Since $s(x)$ is continuous and interpolates, $s(x_k) = y_k$, so the values of C_k and D_k can be evaluated, yielding the expression

$$\begin{aligned} s(x) = \frac{(x_{k+1} - x)^3 M_k + (x - x_k)^3 M_{k+1}}{6h_k} &+ \frac{(x_{k+1} - x)y_k + (x - x_k)y_{k+1}}{h_k} \\ &- \frac{h_k}{6}\Big((x_{k+1} - x)M_k + (x - x_k)M_{k+1})\Big). \end{aligned} \tag{5}$$

So far, the M_k remain unknown and we have used continuity of $s(x)$ and of $s''(x)$. Continuity of $s'(x)$ yields equations for M_k. Differentiating (5) yields, for the interval $[x_k, x_{k+1}]$

$$s'(x) = \frac{-(x_{k+1} - x)^2 M_k + (x - x_k)^2 M_{k+1}}{2h_k} + \frac{y_{k+1} - y_k}{h_k} - \frac{(M_{k+1} - M_k)h_k}{6}, \tag{6}$$

where

$$h_k = x_{k+1} - x_k \text{ for } k = 1, 2, \dots, (n-1). \tag{7}$$

Applying (6) to evaluate $s'(x_k)$ yields

$$s'(x_k) = -M_k \frac{h_k}{3} - M_{k+1}\frac{h_k}{6} + \frac{y_{k+1} - y_k}{h_k}. \tag{8}$$

For $k = 2, 3, \dots, n-1$, the knot x_k is a member of *two* intervals, $[x_k, x_{k+1}]$, as above, and also $[x_{k-1}, x_k]$. Replacing k with $(k-1)$ in (6), to make it refer to the interval $[x_{k-1}, x_k]$, and applying it to the knot x_k yields

$$s'(x_k) = M_k \frac{h_{k-1}}{3} + M_{k-1}\frac{h_{k-1}}{6} + \frac{y_k - y_{k-1}}{h_{k-1}}. \tag{9}$$

Putting these together gives $(n-2)$ equations for the n values M_k. We still need two more equations. One way would be to require that values for $s'(x_1) = y'_1$ and $s'(x_n) = y'_n$ be specified as part of the problem. Using these two relations and equating (8) with (9) yields the following system of equations.

$$\begin{array}{cccccccc} M_1\frac{h_1}{3} & +M_2\frac{h_1}{6} & & & & & = & \frac{y_2-y_1}{h_1} - y'_0 \\ \frac{M_1}{6} & +\frac{h_1+h_2}{3}M_2 & +M_3\frac{h_2}{6} & & & & = & \frac{y_3-y_2}{h_2} - \frac{y_2-y_1}{h_1} \\ & & & \dots & & & = & \dots \\ & & & M_{n-2}\frac{h_{n-1}}{6} & +M_{n-1}\frac{h_{n-1}+h_{n-1}}{3} & +M_n\frac{h_{n-1}}{6} & = & \frac{y_n-y_{n-1}}{h_{n-1}} - \frac{y_{n-1}-y_n}{h_n} \\ & & & & M_{n-1}\frac{h_{n-1}}{6} & +M_n\frac{h_{n-1}}{3} & = & y'_n - \frac{y_n-y_{n-1}}{h_{n-1}} \end{array} \tag{10}$$

This system is equivalent to the matrix equation

$$AM = D,$$

with the matrix

$$A = \begin{pmatrix} \frac{h_1}{3} & \frac{h_1}{6} & 0 & 0 & & \cdots & \\ \frac{h_1}{6} & \frac{h_1+h_2}{3} & \frac{h_2}{6} & 0 & & & \\ 0 & \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{6} & & 0 & \\ & \ddots & \ddots & \ddots & & \ddots & \ddots \\ & 0 & \frac{h_{n-3}}{6} & \frac{h_{n-3}+h_{n-2}}{3} & & \frac{h_{n-2}}{6} & 0 \\ & & 0 & \frac{h_{n-2}}{6} & & \frac{h_{n-2}+h_{n-1}}{3} & \frac{h_{n-1}}{6} \\ & \cdots & & 0 & & \frac{h_{n-1}}{6} & \frac{h_{n-1}}{3} \end{pmatrix} \tag{11}$$

and the vectors

$$D = \begin{pmatrix} \frac{y_2-y_1}{h_1} - y_1' \\ \frac{y_3-y_2}{h_2} - \frac{y_2-y_1}{h_1} \\ \vdots \\ \frac{y_n-y_{n-1}}{h_{n-1}} - \frac{y_{n-1}-y_{n-2}}{h_{n-2}} \\ y_n' - \frac{y_n-y_{n-1}}{h_{n-1}} \end{pmatrix} \tag{12}$$

and

$$M = \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-1} \\ M_n \end{pmatrix}.$$

Since each diagonal element of A is strictly larger than the sum of the off-diagonal elements, the Gershgorin disk theorem shows that A is nonsingular so this equation can be solved for M.

Exercise 10: In this exercise you will write a function to implement the matrix A, the right side D, and solve the resulting matrix system. Then, it will construct the values of the derivatives of the complete cubic spline at the knots. This implementation uses loops rather than vector (componentwise) operations because it is easier to see what is happening with loops. If you wish, you can replace the loops with vector (componentwise) operations.

(a) Begin writing a file named `ccspline.m` to compute the values M_k and the values $s'(x_k)$. Its signature should be

```
function sprime=ccspline(xdata,ydata,y1p,ynp)
% sprime=ccspline(xdata,ydata,y1p,ynp)
%    ... more comments ...

% your name and the date
```

(b) Choose the proper value of `n` and write a loop to define the vector of interval lengths h_k as given in (7). It should be similar to the following:

```
for k=1:n-1
  h(k)= ???
end
```

(c) Write a loop to define the column vector D_k in (12). It will use the quantities y_1' =`y1p` and y_n' =`ynp` and be similar to the following:

```
D(1,1)= ???
for k=2:n-1
  D(k,1)= ???
end
D(n,1)= ???
```

(d) Write a loop to define the matrix A in (11). Your loop should be similar to the following:

```
A=zeros(n,n);
A(1,1)= ???
A(1,2)= ???
for k=2:n-1
  A(k,k-1)=???
  A(k,k)  =???
  A(k,k+1)=???
end
A(n,n-1)=???
A(n,n)  =???
```

(e) Solve the matrix system $AM = D$ for the vector `M` using the backslash operator.

(f) Write a loop to define the derivatives of the spline function given in (8). This vector should be a row vector if `ydata` is a row vector and a column vector if `ydata` is a column vector. Recall that, for a complete cubic spline, s_1' =`y1p` and s_n' =`ynp`. Your loop should look something like the following:

```
sprime=zeros(size(ydata));
sprime(1)=???
for k=2:n-1
  sprime(k)=???
end
sprime(n)=???
```

(g) Test your function using the polynomial $y(x) = x^3$ on the interval $[0, 3]$ using unequal subintervals. Since you are using a cubic polynomial, the resulting spline function is x^3 back again, and you know the derivatives.

```
xdata=[0 1 2  4];
ydata=[0 1 8 64];
y1p=0;
ynp=48;
```

If your results for `sprime` are correct, go on the the next exercise. If not, debug your work using the following strategy.

i. Double-check your code for `sprime` against (8).

ii. Print the matrix A and check that it is symmetric. If it is not, fix your code and test it again now.

iii. The Gershgorin theorem holds because the sum of the off-diagonal terms in each row is strictly smaller than the diagonal term. Double-check that this fact holds for your matrix.

iv. If A is symmetric, then try your code on the interval $[0, 2]$ using the following data

```
xdata=[0 1 2];
ydata=[0 1 8];
```

```
y1p=0;
ynp=12;
```

If your results are correct, then you have an mistake in your subscripts `h(k)`. (Notice that `h(k)=1` for all `k` in this case but not in the previous case.) Fix your code and test it again now.

v. If the results of the previous test are not correct, consider the case

```
xdata=[0 1 3];
ydata=[0 1 27];
y1p=0;
ynp=27;
```

and compute the 3×3 matrix A and the vector D *by hand* from equations (11) and (12). Compare with the Matlab values, and fix the mistake.

vi. If your A and D are correct, compute `sprime` *by hand* from equation (8). Compare with the Matlab values, and fix the mistake.

Now that you are confident of your `ccspline` function is correct, you can use it to interpolate a function. Recall that one of the most common applications of spline interpolation is to interpolate tabular data so that computing derivatives is a major difficulty.

Exercise 11:

(a) Copy your `test_pherm_interpolate.m` to `test_ccspline_interpolate.m` and modify it so that the derivative values are computed using `ccspline`. You will still need derivatives at the end-points. You can continue using `eval_pherm` to evaluate your spline approximation because you know both function values (`ydata`) and derivative values (`y1p`, `ynp`, and `sprime`).

(b) Using equally spaced data points from 0 to 5 for `xdata` and the `runge.m` function to Estimate the maximum interpolation error by computing the maximum relative difference (infinity norm) between the exact and interpolated values at 4001 evenly spaced points. Fill in the following table:

```
  Runge function, Complete Cubic Spline
 ndata =  5  Err(  5) = ________
 ndata = 11  Err( 11) = ________  Err(  5)/Err( 11) = ________
 ndata = 21  Err( 21) = ________  Err( 11)/Err( 21) = ________
 ndata = 41  Err( 41) = ________  Err( 21)/Err( 41) = ________
 ndata = 81  Err( 81) = ________  Err( 41)/Err( 81) = ________
 ndata =161  Err(161) = ________  Err( 81)/Err(161) = ________
 ndata =321  Err(321) = ________  Err(161)/Err(321) = ________
 ndata =641  Err(641) = ________  Err(321)/Err(641) = ________
```

(c) Estimate the rate of convergence by examining the ratios `Err(41)/Err(81)`, *etc.*, and estimating the nearest power of two.

7 Splines without derivatives

If you have only function data (perhaps tabulated) without any way to compute derivatives, you need another set of equations to completely specify M_k for $k = 1, 2, \ldots, n$. One natural way of doing it is to assume that $M_1 = M_n = 0$, whereby the resulting approximation is assumed linear at the endpoints. Such splines are called "natural cubic splines."

An alternative approach is to require the $s'''(x_k)$ is continuous at the two points $k = 2$ and $k = n - 1$, as well as $s''(x_k)$, $s'(x_k)$, and $s(x_k)$ itself. Since $s(x)$ is piecewise cubic, if those four conditions hold, then $s(x)$

is a *single cubic* on the intervals $[x_{k-1}, x_{k+1}]$, $k = 2$ and $k = n - 1$, not two cubics meeting at x_k. Hence, x_k is not properly a knot, and this is called the "not-a-knot" condition.

In the following two exercises, you will examine each of these alternatives.

Exercise 12:

(a) Copy your `ccspline.m` to a file `ncspline.m` and modify the signature to be

```
function sprime=ncspline(xdata,ydata)
% sprime=ncspline(xdata,ydata)
%     ... more comments ...

% your name and the date
```

(b) Modify the first and last lines of the matrix A and the first and last components of the vector D so that they embody the equations

$$\begin{aligned} M_1 &= 0, \text{ and} \\ M_n &= 0 \end{aligned}$$

Hint: The matrix system $AM = D$ above is given explicitly by (10). Look at the first line of this system. How would you change the matrix A so that the first line of the system in (10) becomes $M_1 = 0$? Similarly for the last line.

(c) Use (8) to get $s'(x_1)$ and (9) to get $s'(x_n)$.

(d) The following case represents a linear equation

```
xdata=[0 1 2 3];
ydata=[0 2 4 6];
```

Use it to test that `ncspline` works correctly for one case.

(e) Temporarily print the solution vector `M`, pick some nonlinear function to approximate and verify that `M(1)=M(n)=0`. (You cannot exactly reproduce such a function using natural cubic splines, but you can check the endpoints.) Remove the temporary printing.

(f) Test that `ncspline` and `ccspline` agree when given the same data. Choose

```
xdata=[0,1,2];
ydata=[0,-3,-16];
```

and compute s' using `ncspline` and call it `ncsprime`. Next compute s' using `ccspline` and using `ncsprime(1)` and `ncsprime(end)` for the derivative values. Call the results of `ccspline` `ccsprime`. Check that `ncsprime` and `ccsprime` are the same up to roundoff.

(g) Copy your `test_ccspline_interpolate.m` to `test_ncspline_interpolate.m` and modify it so that the derivative values are computed using `ncspline`.

(h) Using equally spaced data points from 0 to 5 for `xdata` and the `runge.m` function to Estimate the maximum interpolation error by computing the maximum relative difference (infinity norm) between the exact and interpolated values at 4001 evenly spaced points. Fill in the following table:

```
  Runge function, Natural Cubic Spline
 ndata =  5  Err(  5) = ________
 ndata = 11  Err( 11) = ________  Err(  5)/Err( 11) = ________
 ndata = 21  Err( 21) = ________  Err( 11)/Err( 21) = ________
 ndata = 41  Err( 41) = ________  Err( 21)/Err( 41) = ________
 ndata = 81  Err( 81) = ________  Err( 41)/Err( 81) = ________
```

```
ndata =161  Err(161) = ________  Err( 81)/Err(161) = ________
ndata =321  Err(321) = ________  Err(161)/Err(321) = ________
ndata =641  Err(641) = ________  Err(321)/Err(641) = ________
```

(i) Estimate the rate of convergence by examining the ratios `Err(41)/Err(81)`, *etc.*, and estimating the nearest power of two.

In the following exercise, you will see how to use a spline generated using the not-a-knot condition. The not-a-knot condition is more difficult to program, so we will take this opportunity to introduce the Matlab `spline` function, whose default constructive behavior is to use the not-a-knot condition. Complete cubic splines can also be generated using `spline`, and the syntax for that is presented in a remark following the exercise.

Exercise 13:

(a) Copy your `test_ncspline_interpolate.m` to `test_spline_interpolate.m` and modify so that the Matlab `spline` function is used to evaluate the spline function itself using the following syntax:

```
yval=spline(xdata,ydata,xval);
```

where `xdata` are the data points, `ydata` are the data values, and `xval` are the points at which the interpolant is to be evaluated.

(b) Using equally spaced data points from 0 to 5 for `xdata` and the `runge.m` function to Estimate the maximum interpolation error by computing the maximum relative difference (infinity norm) between the exact and interpolated values at 4001 evenly spaced points. Fill in the following table:

```
  Runge function, Not-A-Knot Cubic Spline
ndata =  5  Err(  5) = ________
ndata = 11  Err( 11) = ________  Err(  5)/Err( 11) = ________
ndata = 21  Err( 21) = ________  Err( 11)/Err( 21) = ________
ndata = 41  Err( 41) = ________  Err( 21)/Err( 41) = ________
ndata = 81  Err( 81) = ________  Err( 41)/Err( 81) = ________
ndata =161  Err(161) = ________  Err( 81)/Err(161) = ________
ndata =321  Err(321) = ________  Err(161)/Err(321) = ________
ndata =641  Err(641) = ________  Err(321)/Err(641) = ________
```

(c) Estimate the rate of convergence by examining the ratios `Err(41)/Err(81)`, *etc.*, and estimating the nearest power of two.

Remark 1: It should be clear that the complete cubic spline is smooth and accurate, but requires some derivative values. The natural cubic spline requires no derivative values, but is less accurate than the complete cubic spline. The not-a-knot spline retains the asymptotic accuracy of the complete cubic spline without requiring any derivative information.

Remark 2: It is possible to use Matlab's `spline` function to compute the complete cubic spline as well as the not-a-knot cubic spline. Syntax for the complete cubic spline is

```
% complete cubic spline for row-vector ydata
yval = spline( xdata, [yp1, ydata, ypn], xval);
```

8 Monotone interpolation

Suppose you have a closed metal box containing some water and you heat it while you measure its temperature. You will observe that the temperature rises monotonically until it reaches 100° C. The temperature will then remain constant even as you apply heat until all the water boils away, and then it will begin to rise again. Adding heat will *always* cause a nonnegative temperature change.

If you take this temperature data and attempt to interpolate it using splines, you may find non-physical oscillations in temperature near the boiling point. If the temperature behavior near the boiling point is important to you, you need to interpolate it using a method that respects monotonicity, even if it means lower asymptotic accuracy.

Monotone interpolation is a topic of theoretical importance, and a seminal paper was written by Fritsch and Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM J. Numer. Anal.* **17**(1980)238-246. Matlab provides a function called `pchip` that implements Fritsch and Carlson's algorithm. In the following exercise you will compare a `spline` interpolation with a `pchip` interpolation of data inspired by the water experiment described above.

Exercise 14: Consider the function given by

```
xdata=[0, 1, 2, 3, 4];
ydata=[0, 1, 2, 2, 2];
```

where `xdata=2` represents the time just after boiling starts.

(a) Plot the data using circles with no lines between them.

(b) Construct the not-a-knot spline interpolant using the Matlab function `spline` and plot it on the same frame using at least 100 points.

(c) Construct the monotone cubic interpolant using the Matlab function `pchip` (using the same syntax as `spline` but with the name `pchip`) and plot it on the same frame using at least 100 points. Please include this plot with your summary. You should observe that the `pchip` interpolant is more physically logical than the `spline` interpolant.

Exercise 15:

(a) Copy your `test_spline_interpolate.m` to `test_pchip_interpolate.m` and modify so that the Matlab `pchip` function is used to evaluate the interpolant function.

(b) Using equally spaced data points from 0 to 5 for `xdata` and the `runge.m` function to Estimate the maximum interpolation error by computing the maximum relative difference (infinity norm) between the exact and interpolated values at 4001 evenly spaced points. Fill in the following table:

```
 Runge function, Monotone Cubic Interpolation
ndata =  5  Err(  5) = ________
ndata = 11  Err( 11) = ________  Err(  5)/Err( 11) = ________
ndata = 21  Err( 21) = ________  Err( 11)/Err( 21) = ________
ndata = 41  Err( 41) = ________  Err( 21)/Err( 41) = ________
ndata = 81  Err( 81) = ________  Err( 41)/Err( 81) = ________
ndata =161  Err(161) = ________  Err( 81)/Err(161) = ________
ndata =321  Err(321) = ________  Err(161)/Err(321) = ________
ndata =641  Err(641) = ________  Err(321)/Err(641) = ________
```

(c) Estimate the rate of convergence by examining the ratios `Err(41)/Err(81)`, *etc.*, and estimating the nearest power of two. You should observe a lower rate of convergence than for the Matlab `spline` function.

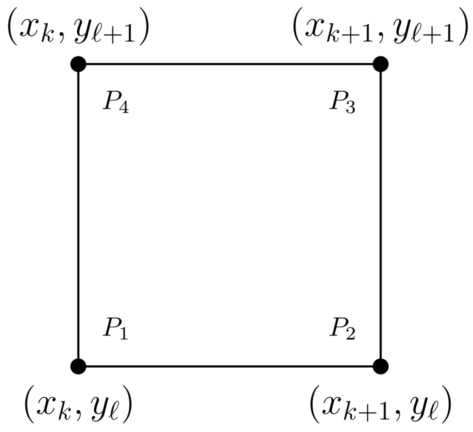
9 Extra credit: Bilinear interpolation in two dimensions (8 points)

This extra credit exercise involves interpolation in two dimensions using piecewise bilinear functions. For simplicity, all work will be performed on a square.

In the previous sections, you used bicubic Hermite functions to map the unit square into a curved-sided region in space. In this exercise, you will be using piecewise *bilinear* functions to interpolate a given function defined over a square, and then measuring the interpolation error much as you did in the previous lab.

Matlab remark: Array (vector) syntax is more complicated in two dimensions than in one. I recommend you use `for` loops in this exercise instead of array syntax. If you do wish to use array syntax, you should look carefully at the documentation for the `ndgrid` command and generate *matrix* values for `xval` and `yval`.

First, note that a bilinear function (a bilinear function is the product of a linear function in x with a linear function in y) can be constructed over an arbitrary rectangle as a linear combination of four functions.



$$\phi_1(x,y) = \left(\frac{x_{k+1}-x}{x_{k+1}-x_k}\right)\left(\frac{y_{\ell+1}-y}{y_{\ell+1}-y_\ell}\right)$$

$$\phi_2(x,y) = \left(\frac{x-x_k}{x_{k+1}-x_k}\right)\left(\frac{y_{\ell+1}-y}{y_{\ell+1}-y_\ell}\right)$$

$$\phi_3(x,y) = \left(\frac{x-x_k}{x_{k+1}-x_k}\right)\left(\frac{y-y_\ell}{y_{\ell+1}-y_\ell}\right)$$

$$\phi_4(x,y) = \left(\frac{x_{k+1}-x}{x_{k+1}-x_k}\right)\left(\frac{y-y_\ell}{y_{\ell+1}-y_\ell}\right)$$

Exercise 16:

(a) Fill in a table similar to the following with the values $\phi_m(P_n)$

	ϕ_1	ϕ_2	ϕ_3	ϕ_4
P_1				
P_2				
P_3				
P_4				

and use it to explain why the unique bilinear function $I(x,y)$ that interpolates a given function $f(x,y)$ at the four points $\{P_n\}_{n=1,2,3,4}$ can be written as

$$I(x,y) = \sum_{n=1}^{4} f(P_n)\phi_n(x,y) \tag{13}$$

(b) Write a function m-file named `runge2d` to evaluate the function $f(x,y) = 1/(1+(x+y)^2)$.

(c) Write a function m-file named `eval_pbilin.m` with beginning

```
function zval=eval_pbilin(f,xdata,ydata,xval,yval)
% fval=eval_pbilin(f,xdata,ydata,xval,yval)
% find the piecewise bilinear interpolant, I, of a function f
% xdata=vector of x data coordinates
% ydata=vector of y data coordinates
% xval=vector of x coordinates at which to compute the interpolant
% yval=vector of y coordinates at which to compute the interpolant
% zval=MATRIX of interpolated values
%      zval(k,ell)= I(xval(k),yval(ell))

% your name and the date
```

Hint 1 It is much easier to use simple for-loops in this code than to try to use elementwise operations.
Hint 2 Use `bracket` twice, once for xval, once for yval.

(d) Test that your code *interpolates* the `runge2d` function on the square $[0,1]\times[0,1]$ (`xdata=[0,1]` and `ydata=[0,1]`) by computing

```
zval=eval_pbilin(@runge2d,xdata,ydata,xdata,ydata)
```

and comparing with the four exact values of runge2d at the four corners of the square.

(e) Write a function m-file `test_pbilin_interpolate.m` along the lines of `test_pherm_interpolate.m` that compares the interpolant with the exact function value. Use only 401 values (not 4001) for each of `xtest` and `ytest`. Do not use 4001 values for each!

(f) Test your code by computing the errors for the three functions $f(x,y)=1$, $f(x,y)=x$, $f(x,y)=y$, and $f(x,y)=xy$. Errors should be roundoff for these four bilinear functions.

(g) Compute the errors for the `runge2d` function using 11, 21, 41, 81, and 161 data values in the interval [0,5] in each of the x and y directions. Estimate the rate of convergence of bilinear interpolation.

Last change `$Date: 2016/08/10 23:19:23 $`