Math 128a - Program 2 - Due April 25

Your assignment is to implement a program which computes and plots a curve in the x, y plane, by fitting a cubic spline curve through a given set of points (x_i, y_i) in the plane. You can think of this as the algorithm underlying various drawing programs like xfig or paint. This set of points may be from any source, including clicking on your mouse, or arrays computed by another program, like Program 1 earlier this semester.

The basic idea is that you should be able to click on a matlab graphics window to specify a list of points in the plane, and have the program connect them with a curve. There is an option to identify certain points as "corners", so the program will not try to make the curve smooth there (have a continuous tangent); this lets you draw curves that are partly smooth and partly polygonal. There is another option to make the curve closed, i.e. connect the last point to the first point; this lets you draw circles, for example.

If your points are supplied as an input array, they may be the output of Program 1, which computed them as solutions to $f(x, y) = 0$. So there is also an option to test how well the curve you compute satisfies $f(x, y) = 0$ along its entire length, not just the input points.

We will supply most of the Matlab code for this problem, including a program to test it. You will need to write 4 subroutines, for which we supply detailed input and output descriptions. To help you debug, we will also tell you what some of the correct output should be from the test program. You will turn in your code so we can run it and make sure it works.

The main routine is called **Param2dSpline.m**. Here is a detailed description of its inputs (see also the comments in the code):

1. A flag *mouse* indicating the source of the points in the plane. If $mouse = 0$, they are supplied by the next two input arguments $x(1 : n)$ and $y(1 : n)$. If $mouse = 1$, they should be input using the mouse by pointing at the graphics window and clicking on the desired location of each point. A left mouse click means the point is a "non corner point" (explained below), a right mouse click means the point is a "corner point", and hitting any other key on the keyboard means end of input.

2. Two arrays $x(1 : n)$, $y(1 : n)$ defining the x and y coordinates of the points along the curve. If $mouse = 0$, these are used as input. If $mouse = 1$, they are ignored and instead set by the user clicking on the mouse as just described.

3. The name $func$ of a function that evaluates $f(x, y)$, where the points along the curve including $x(i), y(i)$ are intended to satisfy $f(x(i), y(i)) = 0$. If there is no such function (for example if the points are input by the mouse), then the input function should be the empty string `''`. This is used to test whether you have correctly implemented the splines.

4. A flag *closed* indicating whether the curve should be closed, that is $(x(1), y(1))$ should be connected to $(x(n), y(n))$ (if $closed = 1$), or not ($closed = 0$).

5. An array $cn(1 : m)$ of corner indices. If $mouse = 0$, $cn(1 : m)$ is used as the input, but if $mouse = 1$, cn is determined by mouse-clicking as described above (right mouse clicks). No matter how $cn(i)$ is specified, it indicates that $(x(cn(i)), y(cn(i)))$ is to be treated as a corner in the curve (details below). If $closed = 0$, the endpoints are defined to be corners as well, whether or not 1 and n are included in cn. If $closed = 1$, there may be no corners (cn may be empty, i.e. have length zero).

6. A flag *corners*, which is > 0 to indicate that the program should identify additional corners automatically, and 0 to indicate that it should not. The way a positive value of *corners* is

used to detect a corner is discussed further below. Corners detected automatically are in addition to any specified by *cn*.

The outputs of Param2dSpline will be

1. n, the number of distinct points on the curve.
2. 2 arrays $xout(1:ne)$, $yout(1:ne)$ of points on the curve. If $mouse = 0$ and the curve is not closed, then $ne = n$ and $xout$ and $yout$ are identical to inputs x and y, respectively. If $mouse = 0$ and the curve is closed, then $ne = n + 1$, and $xout(n+1)$ and $yout(n+1)$ are set to $xout(1)$ and $yout(1)$, respectively. If $mouse = 1$, the situation is similar except that $xout(1:n)$ and $yout(1:n)$ are determined by clicking on the mouse.
3. An array $s(1:ne)$ of "pseudo-arclengths"; $s(i)$ is the approximate distance of the point $(xout(i), yout(i))$ from $(xout(1), yout(1))$ along the curve (see below for details). (ne is defined as above.)
4. Two arrays $xc(1:4, 1:ne-1)$ and $yc(1:4, 1:ne-1)$ defining the cubic splines on each interval. The curve connecting the points $(xout(i), yout(i))$ and $(xout(i+1), yout(i+1))$ is defined for s in the interval $[s(i), s(i+1)]$ by the formulas

$$\begin{aligned} x(s) &= xc(4,i)*(s(i+1)-s)^3 + xc(3,i)*(s-s(i))^3 + xc(2,i)*(s-s(i)) + xc(1,i)*(s(i+1)-s) \\ y(s) &= yc(4,i)*(s(i+1)-s)^3 + yc(3,i)*(s-s(i))^3 + yc(2,i)*(s-s(i)) + yc(1,i)*(s(i+1)-s) \end{aligned}$$

   where $i = 1, ..., ne-1$.
5. An array *newcn* of all corner indices, in increasing order. For example, if $newcn(1) = 5$, then $(xout(5), yout(5))$ is a corner. *newcn* includes the corners in *cn*, 1 and n if the curve is not closed, and any new ones detected as specified by *corners*. *newcn* should not contain any duplicate entries.
6. An *errorbound*, if $func$ is not '', which is the largest value of $|f(x,y)|$ at a large number of densely sampled points along the curve (details below).
7. A plot of the curve. The noncorner points $(x(i), y(i))$ on the curve are marked with circles, and the corners with plus-signs.

Here is how you will do a spline fit. First we have to define arclength s: $s(1) = 0$, and corresponds to starting the curve at $x(1), y(1)$. Then

$$s(2) = ((x(2) - x(1))^2 + (y(2) - y(1))^2)^{1/2}$$

the distance from $(x(1), y(1))$ to $(x(2), y(2))$. Similarly

$$s(i) = s(i-1) + ((x(i) - x(i-1))^2 + (y(i) - y(i-1))^2)^{1/2} \quad .$$

Now consider a segment of the curve (i.e. beginning and ending at corners) starting at $(x(i), y(i))$ and ending at $(x(j), y(j))$, both of which are corners, with no corners in between them. Then we will build 2 cubic splines, one interpolating the points

$$(s(i), x(i)), (s(i+1), x(i+1)), ..., (s(j), x(j))$$

and the other interpolating

$$(s(i), y(i)), (s(i+1), y(i+1)), ..., (s(j), y(j))$$

The first spline is defined by the coefficients $xc(1:4, i:j-1)$, and the second by $yc(1:4, i:j-1)$. We use the formulation of cubic spline described on pages 350–354 of the text. You should explicitly build the matrix on page 352 (call it S), but you can solve the system of equations there (call it $Sz = v$) via the one-line Matlab command $z = S\backslash v$ instead of using the more efficient algorithm on the next page of the text, if you want (this is not as efficient, but this is not so important here).

Next, suppose that we have a single smooth closed curve, without corners. Then it is not appropriate to use the "natural cubic spline" described in the text, because it will introduce an artificial corner at the initial point of the curve. So in this case, you have to change the formulation of the spline to make sure the first and second derivative are continuous all the way around the curve. Here is a hint on how to do this: Consider just the x coordinates; the y coordinates are analogous. We want to fit splines through the following points (ne = n+1):

$$(s(1), x(1)), ..., (s(n), x(n)), (s(ne), x(ne))$$

where $x(ne) = x(1)$. From the formulation on page 351, we have n cubic polynomials to determine, and so $4n$ unknown parameters. We get $2n$ conditions from having each cubic interpolate the endpoints of each subinterval, $n-1$ conditions from the continuity of the first derivative at each interior point $s(2), ..., s(n)$, and another $n-1$ conditions from the continuity of the second derivative at these points. That leaves 2 conditions. One of them will be matching the derivative at $s(1)$ with the derivative at $s(ne)$, and the other will be matching the second derivatives. Show how to modify the formulation of the natural cubic spline to get a new system of linear equations to solve. You should write up your derivation of this spline at the same level of mathematical detail as the text derives natural splines.

You may *not* use the built in cubic spline functions in matlab, except to help you debug.

Here is how to detect corners automatically. To determine whether $(x(i), y(i))$ is a corner, compute the angle θ between the two line segments L_- and L_+, where L_- connects $(x(i), y(i))$ and $(x(i-1), y(i-1))$, and where L_+ connects $(x(i), y(i))$ and $(x(i+1), y(i+1))$. If all three points are colinear, with points $i-1$ and $i+1$ separated by point i, this means $\theta = \pi$, and otherwise $0 \le \theta < \pi$. Then if $\theta \le \pi - corners$, $(x(i), y(i))$ is considered a corner. For example, if $0 < corners \le \pi/2$, then the corners of a perfect rectangle would be identified as corners, but not if $corners > \pi/2$.

Finally, here is how to test how well the computed curve satisfies an implicit equation $f(x, y) = 0$. If input string $func$ has nonzero length, compute 1000 equally spaced points along the curve (equally separated in s from $s(1)$ to $s(ne)$), evaluate the splines for x and y at all these points, and compute $errorbound = \max_{1 \le i \le 1000} |f(x_i, y_i)|$.

The program is organized as follows. Each subroutine is described, including the four you have to write: **FindCorners**, **EvalSpline**, **ClosedSpline2d**, and **NSpline2d**.

- **Param2dSpline.m** is the main program. After a few error checks, it prepares the points, computes the spline coefficients, plots the curve and, if required, computes **errorbound**. This main program calls the following functions:

    1. **PreparePoints.m** A rather messy function that gets the list of points and corners ready for analysis. First, if required, it gets points from the mouse. Then it removes repeat points from the list. If required it performs corner detection. It also fixes up the list of points to repeat the first point at the end if the curve is closed. It calls the following functions:

(a) **GetMousePts.m** Uses the Matlab "ginput" command to get locations of mouse clicks and which button was used.

(b) **FindCorners.m** *You need to write this.* This is called if $corners > 0$ to identify corners automatically. It takes as inputs the coordinates of the points on the curve, a threshold (equal to the value of *corners*) and the *closed* flag. If the curve is not closed ($closed = 0$) the end points are considered corners. Otherwise, for every point $(x(i), y(i))$ with 2 neighbors, it is tested for being a corner as described before. FindCorners returns the list of indices of corners in increasing order. For example, if points $i = 1$, $i = 3$ and $i = 4$ are considered corners, then FindCorners should return $[1, 3, 4]$.

2. **GenSpline2d.m** This is the general function to compute the coefficients describing the 2D splines. It has a lot of cases to consider. The only case in which we use the closed curve algorithm is when the curve is closed and there are no corners. If the curve is not closed we make sure that **cn** points to the beginning and end of the list, and then we just do natural splines for each smooth segment from one corner to the next. Although the function **NSpline2d** returns arclengths starting at 0 for each segment, we can just shift these after the fact and all we have done is translated the appropriate splines (the coefficients of the splines don't change under translation). The subtlety comes when the curve is closed but there is at least one corner which is not the first point. In this case we have to shift indices and wrap around so that we start at a corner. Then we do natural splines for each smooth segment as before. Then we have to shift all the indices back so that the indices for the arclengths and the coefficients match the indices for the points. In this function we call:

   (a) **ClosedSpline2d.m** *You need to write this.* This function first computes the necessary cumulative arclengths s and then invokes the periodic spline algorithm (which you also need to write) twice to compute the coefficients of the splines describing x and y as *periodic functions* of arclength.

   (b) **NSpline2d.m** *You need to write this.* This function first computes the cumulative arclengths s and then invokes the natural spline algorithm (which you also need to write) twice to compute the coefficients of the splines describing x and y as functions of arclength.

3. **Plot2dSpline.m** Plots a given number of equally spaced points along the parametrized spline curve defined by given coefficients and knots. This function calls the following function twice, once for x and once for y.

   (a) **EvalSpline.m** *You need to write this.* This evaluates a cubic spline at a list of x-values given the coefficients in a matrix c and the list of knots $t_1, \ldots t_n$. For efficiency's sake you may assume that the x-values be in increasing order. Values of x less than t_1 are evaluated using the first cubic and values greater than t_n are evaluated using the last cubic. Note that, if the coefficients are intended to represent a periodic function this will not know that, but that as long as $t_1 \leq x \leq t_n$ that doesn't matter anyway.

4. **MaxAbsFSpline.m** This evaluates **func** at a given number of equally spaced points along the curve and returns the maximum absolute value of **func** at those points. It is almost identical to *Plot2dSpline* and also calls:

   (a) **EvalSpline.m**

- **Test_Param2dSpline** tests Param2dSpline by passing it sets of points on known curves defined by various functions $f(x, y) = 0$, computing spline fits (some closed, some not closed, sometimes with corner detection, sometimes without). It tries each $f(x, y)$ with an increasing sequence of values of n = number of points along the curve; our intuition is that using more points should make the splines better approximations to the true solution curve of $f(x, y) = 0$. We test this intuition by computing the largest value of $|f(x, y)|$ for many points along each computed spline, and seeing how it decreases as n increase. We also print the number of corners detected. Test_Param2dSpline calls a number of routines besides Param2dSpline:

  1. **Quintic2** computes $f(x, y)$ for y a degree 5 polynomial in x. For this example, the curve is not closed ($closed = 0$), and corner detection is turned off ($corners = 0$).
  2. **Circle2** computes $f(x, y)$ for a circle of radius .5 centered at the origin. For this example, the curve is closed ($closed = 1$), and corner detection is turned off ($corners = 0$).
  3. **Cusp2** computes $f(x, y)$ for the same cusp as in Programming Assignment 1. For this example, the curve is not closed ($closed = 0$), and corner detection is turned on ($corners = 1$).
  4. **Rose2** computes $f(x, y)$ for a "rose" with 5 petals centered at the origin. For this example, the curve is closed ($closed = 1$), and corner detection is turned off ($corners = 0$).
  5. **Jagged2** computes $f(x, y)$ for y a piecewise linear function of x. This is tested twice, both times with the curve not closed ($closed = 0$), and once with corner detection turned off ($corners = 0$) and once with corner detection turned on ($corners = .1$).

  For debugging purposes, here are some correct output values that Test_Param2dSpline should print out:

Test Case	n	# corners detected	max $\|f\|$ along curve
Quintic	10	2	7.9093e+01
Quintic	90	2	5.0585e-02
Circle	10	0	2.2358e-04
Circle	90	0	3.0968e-08
Cusp	11	3	2.8868e-03
Cusp	91	3	2.9945e-05
Rose	6	0	4.0910e-01
Rose	100	0	2.3852e-04
Jagged ($corner > 0$)	11	7	3.9504e-02
Jagged ($corner > 0$)	81	6	9.7145e-15
Jagged ($corner = 0$)	11	2	7.6430e-02
Jagged ($corner = 0$)	81	2	8.8181e-03

What you should turn in.

1. Documentation describing how you implemented the 4 requested routines. In particular, you should supply the code itself electronically so we can run it, and paper documentation describing how your algorithms were derived, in particular corner detection and periodic splines (exclude the natural splines already done in the text.)

2. Include all the tabular output of Test_Param2dSpline (the tables of n, # corners, and $\max|f|$ for each test case) and pictures of the computed solution graphs just for the smallest and largest values of n for each test case. We will also run your code to make sure it agrees with what you turn in.

3. Discuss interesting features of the output, which corners were discovered, or not (if it is not obvious), and whether the the error $\max|f|$ decreased as fast as you "expected".

4. Test the mouse input feature of your code by supplying a few graphical examples of your own artwork (we will not grade on artistic ability). Make sure to test all features of your code ($closed = 0$ or 1, $corners = 0$ or > 0).

On teamwork. Please reread the rules posted on the web page about doing assignments on your own, and talking to others being ok but having to turn in your own work.