University of Pennsylvania
EAS 205: Spring 2010
MATLAB PROJECT: Cubic Spirals
Instructor: Donna Dietz

For this intense but fun project, you will be creating a program which allows the user to adjust two control points of a two dimensional, parametric Bézier Cubic in an attempt to make the resulting cubic have monotonic curvature. (If a curve has monotonic curvature, we will call it a spiral, so long as the curvature does not switch sign.)

This project is (probably) not as hard as it looks, but it is somewhat involved. There are many little pieces, each of which requires some explanation and a little work, so altogether, it looks like a lot, but it's really lots of small and easy things adding up to one big thing. Budget your time. Go slow and steady, and it will work out. Get help if you think you are moving too slowly, so you can finish on time. Time management is important.

For outside reading, start with the Wikipedia article, *http://en.wikipedia.org /wiki/Bézier_curve*. (Now for a disclaimer. There are good times and bad times to reference Wikipedia. This is a good time. Why? It just so happens that I am an expert in this area, and I hereby am telling you that this article, in the state I last saw it, is actually good to read. If I did not already have the background, I could not refer you to this article, but I do, so I can. Students should be cautious of relying too heavily on uncertifyable resources such as this.) Also, the Wikipedia article on "Curvature", particularly the section on "local expressions" is good to read. The article "Monotonic_function" has some diagrams that will suffice for explanation. Wolfram has a nice demonstration of circles of curvature. *http://demonstrations.wolfram.com/CircleOfCurvature/* I will also discuss these concepts in lab and/or in class.

This project will contain 17 required .m files, but you are free to include more if you wish. In my case, the largest file (CubicSpirals.m) is 260 lines, and contains 2 internal subfunctions and 9 external subfunctions. The subfunctions take roughly half the space in the file. The other 16 files are each quite small. Kpure is my second longest file, having only 39 lines. The remainder are roughly 15 or fewer lines. So, this whole project will have under 400 lines of code. (That includes comments and blank lines, of course.)

You will have various forms of assistance in this project. First, I will provide you with a file called *DotMarch.m* which showcases various techniques I want you to become familiar with. This little program shows a figure of a point and invites the user to move it around the screen. It can change color. You can use this file as a jump-off point for your *CubicSpirals.m* file. Another bit of assistance you will have is what is called *pcode* files from my work. This will allow you to play with my project without seeing it directly. Part of the reason I decided to have so many .m files rather than using more subroutines was so that you could poke around at my functions via the pcode files. Since there are so many small components, this

should help reduce project-interia, allowing you to make progress right away as soon as you start to work. Simply read the project specifications for each .m file, write it, then test against mine to assure that you are getting the same results as I am. To run these files, just type the name of the file in your Matlab Command window, as if the .p were a .m. You just can't see the source, that's all! You also can't get help on .p files. And of course, you will have assistance in the form of this document, your TA, and me. If your question is relevant to the entire class, I'll email the class as a whole with the answer.

User interface requirements: You will open one or more figures, not overlapping, which contain legible instructions to the user, a plot of the parametric cubic (with equal spacing on the two axes), a plot of the curvature, and a plot of the derivative of the curvature. (To split a figure into subplots, use *subplot*.) The location of each mobile control point should also be reported back to the user on one of the figures. The leftmost control point should be at $(0, 0)$, and the fourth/last one should be at $(1, 0)$. These two control points are immobile. You will also display an *iterated cubic* (also with equal spacing on the two axes). I will explain this in class. The idea behind this iterated cubic is to extend the cubic in a self-similar fashion (roughly 9 times) so that it is visually obvious when a spiral occurs. At the join points, the tangents and curvatures must match. So, if the original cubic is a spiral, the iterated cubic is also a spiral, but much more enlightening and fun to look at. There also must be some way you communicate to the user that a spiral has been obtained. In my case, I alter the colors of all the plots as well as the text in the welcome window. You can be creative with how you alert the user that success has been achieved.

Who calls whom? This is easiest to visualize in an outline format.

- CubicSpirals
  - BezPolyNumeric
    - EvalPoly
    - BezPolyPure
      - BezCoef
  - Knumeric
    - EvalPoly
    - KPure
      - BezCoef
      - PolyPower
  - KPrimeNumeric
    - Knumeric
  - CrossesZero
  - IterateCubic (recursive)
    - BezPolyNumeric
    - MakeSelfSimilar
      - FindRotAng
      - Rot
      - FindDilRatio
      - Dil
      - Trans

Specifications for each .m file

function [out] = BezCoef (degree, index)
This function gives the polynomial coefficients of the Bézier Polynomial of given degree and index. For example, BezCoef(2,1) will return [-2 2 0], because by definition, $B_1^2 = C_1^2 t(1-t) = -2t^2 + 2t + 0$. Note that on C and B, n is, like i, an index, not a power. C is the *nchoosek* function in Matlab.

$$B_i^n(t) = C_i^n (1-t)^{n-i} t^i$$

function [out] = BezPolyNumeric (weights, t)
The weights (one dimesion only) of the Bézier polynomial go in, along with a vector t. What comes out is a vector the same length of t which has evaluated that polynomial. My code works for any reasonable integer degree, but you are only required to make it work for an input with weights having length four. (The polynomial of degree three has four weights.) Any t vector should be allowed as input, but usually this will be a range from 0 to 1. For example BezPolyNumeric([1 2 3 4],0:0.2:1) will return [1.0 1.6 2.2 2.8 3.4 4.0].

function [out] = BezPolyPure (weights)
The weights here are the same as above, but the output is the polynomial coefficients, not the evaluated values. For example, BezPolyPure([1 2 1 3]) will return [5 -6 3 1].

$$\mathbf{p}(t) = \Sigma_{\nu=0}^{3} \mathbf{b}_\nu B_\nu^3(t)$$

function [out] = CrossesZero (v)
This is probably the easiest function of the project! You send in a vector and if there are both positive and negative values, the output is 1, otherwise, it's zero. I used the Matlab command *any* to help me. I don't care what you do with zeros.

function CubicSpirals ( ) and the CubicSpirals.m file
This is the main function. It does not take any direct input or output. This is the function which manages the figures, calls other functions, decides what to place where, and so forth. The only two subfunctions you absolutely need to have inside are the analogous ones to the ones in *DotMarch.m*. However, I found it convenient to put several other subfunctions outside the main function, but still inside the .m file. You can guess what those are by looking at the structure of *DotMarch.m*. (Having said this, it is actually possible to break those two subfunctions out of the main function, but it requires more messing around with the plot structures, which is a pain.)

function [M] = Dil (factr)
This takes in a scalar, $factr$ which is then multiplied by $eye(2)$ and returned. Very easy.

function [out] = EvalPoly (p, t)
This takes in a vector p of polynomial coefficients, and t, a vector. It returns a vector the same length as t, which contains the evaluated polynomial at all values of t. If you get stuck, ask for hints on this. It's really short when you vectorize!

function [out] = FindDilRatio (pts)
This gives the dilation ratio for a given set of control points for our 2D Bézier cubic. The dilation ratio is the absolute value of the ratio of the two endpoint curvatures, $out = |K(0)/K(1)|$. So for example, [0 1 2 3; 0 1 1 0] will return 1, and indicates four control points in the plane, starting with (0,0) and ending with (3,0).

function [out] = FindRotAng (pts)
The control points have the same format as above, and the output is the rotation angle associated with traversal of this cubic between the first and last control point. The cubic begins at the first control point, with a path heading directly towards the second control point, so the initial angle of travel is found using *atan*2 on those first two control points. Likewise, the last two control points give the final angle of travel. Play with direction and what that might mean. Imagine you are an insect walking along this cubic. The object is to figure out what angle you need to rotate the self-similar cubic through so it can join up smoothly with the previous copy. Ask for help if you get stuck!

function [newxlist, newylist] = IterateCubic (pts, t, xlist, ylist, depth)
This function calls itself. At the top level, xlist=[ ] and ylist=[ ]. Play with this. You can plot the output by doing a *plot(newxlist, newylist)* after calling this function. This one is fun!

function [out] = Knumeric (pts, t)
This does what you'd think it does. It just calls the other functions and spits out the numerical values for each value in the t vector as input.

function [out] = KprimeNumeric (pts, t)
This function first calls Knumeric, then does the following to produce a vector the same length as t which contains derivatives of the values returned by Knumeric. For indices other than 1 and *end*,

$$\frac{dK}{dt}(i) = \frac{K(i+1) - K(i-1)}{t(i+1) - t(i-1)}$$

For the first index,

$$\frac{dK}{dt}(1) = \frac{K(2) - K(1)}{t(2) - t(1)}$$

and for the last index,

$$\frac{dK}{dt}(end) = \frac{K(end) - K(end-1)}{t(end) - t(end-1)}.$$

So, for example, KprimeNumeric([0 1 2 3; 0 1 2 0],0:0.5:1) returns [-1.8262 -0.1789 1.4684].

function [num, denomSq] = Kpure (pts)
This function calculates the (often huge) coefficients for the actual numerator and square of the denominator of the curvature, and returns them. The Matlab command *polyder* is useful, as is *conv*, which multiplies polynomials. You have to watch out for lengths. You can't add or subtract polynomials of different lenghts,

so you have to adjust them. For example, [1 2 1] - [0 1 1] works, but [1 2 1]- [1 1] is meaningless, so you have to turn [1 1] into [0 1 1]. Remember that

$$K(t) = \frac{x'y'' - y'x''}{\sqrt{((x')^2 + (y')^2)^3}}$$

where x and y are functions of t.

function [newpts] = MakeSelfSimilar (pts)
This function starts with four control points, then translates, rotates, and dilates as necessary (and retranslates at the end) so that the new control points will be correct for the iterated cubic to fit properly. Remember that the new first control point is the same as the old last control point. (Also, the new slope at $t = 0$ is the same as the old slope at $t = 1$, and the new curvature at $t = 0$ is the same as the old curvature at $t = 1$. These conditions are handled by $FindRotAng$ and $FindDilRatio$.)

function [out] = PolyPower (poly, pow)
This function can be written with a loop or with recursion. Recursion is, of course, way cooler. This function is really just for fun, because we only use it up to pow=3. Make use of *conv* and this will be done quickly. You can create Pascal's triangle easily. Try $PolyPower([1 \quad 1], 10)$ for fun.

function [M] = Rot (theta)
This just spits out the rotation matrix for theta in the plane. This is an easy one.

$$M(\theta) = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix}$$

function [pts] = Trans (pts_in, dx, dy)
This shifts a matrix of points ($2 \times n$ matrix) by another point, namely (dx, dy). This is an easy one!

This should be plenty to get you started! Have fun! Don't forget to ask for help if you need it, and read your email!