CS 322 Project 2: Spline Editor

out:	Thursday 1 March 2007
Code due:	**Friday 16 March 2007**

1 Background

Many programs, for instance CAD/CAM in engineering applications or animation systems used in film production, need to represent curves and curved surfaces. A curve in two dimensions is often specified parametrically, using a function $\mathbf{f}(t) = (x(t), y(t))$; that is, there are two functions, one for the x-coordinate as a function of t and one for the y-coordinate as a function of t, and the curve is traced out as t varies, for instance from 0 to 1. Although any pair of functions of one variable defines a parametric curve, the most common curves used in practice are defined by polynomials.

1.1 Splines

To allow easy editing of curves, most vector editors allow the editing of curves using control points. A spline is a set of control points, along with a method for generating a parametric function for a curve from those control points. One commonly used type of spline is cubic Bézier splines, which generate cubic polynomial functions for the curve.[1] A segment of a Bézier spline is described by the positions of 4 control points, where the curve passes through both the first and fourth control points with a derivative specified by the position of the second and third points relative to the first and fourth. A given spline can consist of multiple segments linked together—Bézier segments share the first and fourth control point with their predecessor and successor neighboring segments, respectively. A Bézier spline is C^0 but not necessarily C^1, because at the boundary between neighboring spline segments, one segment could specify one derivative while the other could specify an entirely different derivative. We can enforce a weaker version of continuity (known as G^1 continuity) between spline segments by requiring that the third control point of one segment, the fourth/first control point shared between the two segments, and the second control point of the next segment are all colinear.

Given the positions of the four control points, it is easy to obtain the coefficients of the cubic polynomial for either of the two types. Given a matrix $\mathbf{C}$ of size 4×2 that contains the positions of the control points in order from top to bottom, the coefficients of the cubics $x(t) = a_3t^3 + a_2t^2 + a_1t + a_0$ and $y(t) = b_3t^3 + b_2t^2 + b_1t + b_0$ for Bézier curves are given by the matrix product

[1] Bézier splines exist for any degree of polynomial, but we will stick to the commonly used cubic Béziers.

$$\begin{bmatrix} a_3 & b_3 \\ a_2 & b_2 \\ a_1 & b_1 \\ a_0 & b_0 \end{bmatrix} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{C} \tag{1}$$

Typically, each segment of a spline is drawn on screen by approximating it as a sequence of line segments between the points $(x(t), y(t))$ for a sufficiently dense sequence of t values.

1.2 Polynomials in MATLAB

MATLAB provides several functions to help in using and manipulating polynomial functions. `polyval(coeffs, x)` takes in a vector of coefficients (ordered from highest degree to lowest) and a value, and returns the value $p(x)$, where p is the polynomial with coefficients from the vector. `roots(coeffs)` returns the roots of the polynomial whose coefficients are in coeffs. Note that `roots()` can return imaginary values, so your code should check for and discard those roots (using `imag()` and `real()` as needed). `polyder(coeffs)` returns the coefficients of the derivative of the polynomial with the given coefficients. `conv(coeff1, coeff2)` returns the coefficients of the *convolution* of the polynomials p_1 and p_2 with coefficients from `coeff1` and `coeff2` respectively. The convolution is equivalent to the polynomial $p_3(x) = p_1(x)p_2(x)$, that is, the product of the two polynomials, so `conv` returns the coefficients from the polynomial $p_3(x) = p_1(x)p_2(x)$

One note about mathematical stability—finding the roots of a quadratic polynomial can be done stably (that is, without significant errors) by direct computation in fixed-precision arithmetic. There is a closed form for finding the roots of a cubic, but in general it can not be guaranteed to be stable in fixed-precision arithmetic and so should not be used in practice. There is no useful closed form for quartics, and none at all for higher order roots. In general, `roots()` finds the roots through an iterative algorithm rather than a closed-form solution, and for cubics and higher should be used to find the roots of polynomials.

2 Assignment

In this assignment you will build a simple interactive curve editor that lets the user sketch curves with the mouse, select them, edit them, and slice them. This explores interpolants, linear and nonlinear fitting, and root finding.

The framework can be started with the command `splineMaker()`. It contains basic drawing capabilities, including the ability to make and edit splines by dragging their control points around. For each spline, it generates the cubic polynomials for each segment and evaluates them over a dense sampling of values of t within $[0, 1]$ and then draws them on screen as a collection of linear segments; as long as the sampling of t is fine enough, it appears onscreen as a curved surface. It supports two types of curves, C^0 Bézier curves and continuous Bézier curves, where the control points are constrained so that the curve remains smooth. It also supports closing curves as you draw – clicking on the first control point in the segment will create a closed curve, filling in any necessary points needed to make it a well-defined curve.

To draw Bézier curves, set the dropdown to Bézier and simply click to create the control points. To

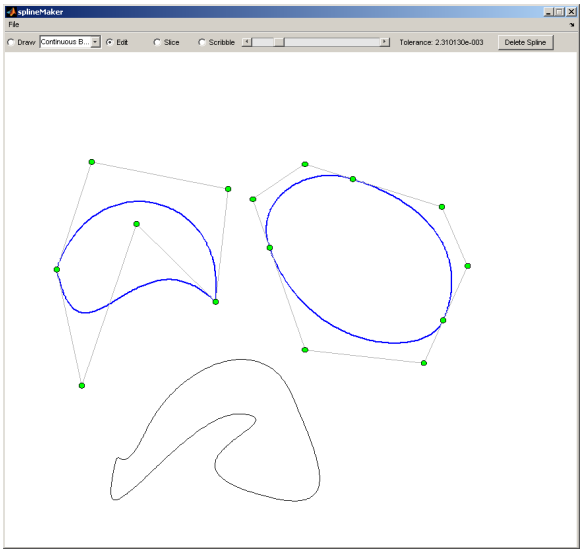

Figure 1: A screenshot of the program in action. The leftmost selected curve is a closed C^0 Bézier curve, while the rightmost selected curve is a continuous Bézier curve

create continuous Bézier curves, set the dropdown to continuous Bézier curves and click and drag to create an endpoint of a segment and the two control points on either side, moving the mouse to adjust the shape of the curve.

We would like to allow additional capabilities, which will be supported by code that you have to write.

1. We would like to have the ability to click on any part of the spline curve (not just the control points) and drag it to a new position. In order to support this, you need to write a function `constrainSplinePoint` that takes the control points for a spline segment, a specified value of t, and a 2D point, and produces a new set of control points such that the curve described by the new points passes through the specified point at t, while moving the control points as little as possible from their original locations.

2. We would like to be able to select multiple splines at once by dragging a rectangle around them, selecting a spline only if it is completely enclosed by the rectangle. In order to support this, you need to write a function `splineIsInside` that takes a rectangle and a spline segment and determines whether or not that spline segment lies completely inside the rectangle.

3. We would like the ability to draw a line on the screen and "cut" any spline that is intersected by the line into multiple splines. To do this, you need to write a function `sliceSpline` that takes a spline segment and a line segment, determines where (if at all) the line intersects, and returns sets of control points for all of the resulting sliced segments (or the original control points if it did not intersect)

4. Finally, we would like the ability to draw on the screen and have our mouse input automatically converted into a spline that reasonably approximates our motion. To do this, you will write a function `matchScribble` that takes a sequence of mouse input and an error tolerance and produces a sequence of spline segments that match the mouse input within the specified error tolerance while using as few spline segments as possible.

3 Guidance

3.1 General Features

The terminology used by the assignment is that a single curve consists of a sequence of *segments*, where each segment is generated as a Bézier curve using 4 control points, and neighboring sequences share their last and first control points with their predecessor and successor segments.

The framework supports both basic Bézier curves and what are termed "continuous Bézier curves". The only difference between the two is that continuous Bézier curves have additional constraints on multiple segments within a curve to ensure that the curve remains continuous. For any function that takes a `type` argument, the type is set to 1 for regular Bézier curves and 2 for continuous Bézier curves.

For control points, they are organized as $k \times 2$ matrices, where the first column is the x-coordinate and the second column is the y-coordinate, and each row is a control point in order. For most of this assignment, you will only be working with segments, which means that you will get a 4×2 matrix of control point data.

3.2 Constraining a Point on a Spline Segment

The goal here is to move the control points as little as possible while still ensuring that the curve passes through the specified point at the specified value of t. This can be formulated as an underconstrained LLS system. Note that because it is underconstrained, there are many possible answers to the system, but we want the one that causes the least total movement in the control points. Since the default behavior of MATLAB when solving underconstrained systems is to minimize the norm of the solution vector, you should formulate your system in terms of the *change* that needs to be made to the control points, rather than the new value they should take (which will simply be the old value plus the computed change). Thus, in this case the minimal norm solution actually means something, in that it can be thought of as the solution that moves the control points "least" (for some definition of least).

For continuous Bézier segments, you should solve for the smallest movement of the control points that both satisfies the constraint and preserves the requirement that the first and second control points should only move along the line between the two of them (and similarly for the third and fourth control points) in order to preserve the continuity. You will probably find it necessary to solve a single system for both the x and y movements (that is, finding a 8×1 vector as opposed to a 4×2 matrix) since there are some constraints that involve both x and y coordinates simultaneously (but in a linear way).

3.3 Checking if a Spline Segment is Contained in a Rectangle

Note that it is not sufficient to simply check to see if the control points are contained in the rectangle, because some curves may be contained in the rectangle while their control points are not. This requires finding the minimal and maximal values of the curve in both x and y and ensuring that they are contained in the rectangle. Finding the minimal and maximal points in x corresponds to finding the roots of the derivative of $x(t)$ and evaluating $x(t)$ at those points, and similarly for y. Note that this is a quadratic function, and so there is a closed form solution for the roots.

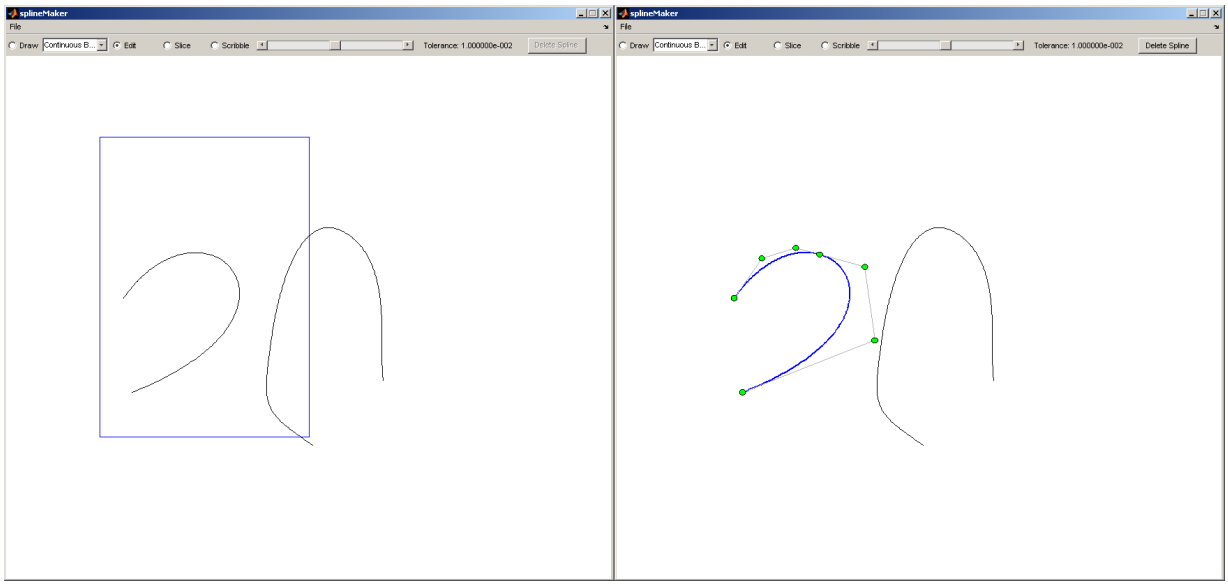

Figure 2: Selecting multiple splines. Note how only one is selected

The rectangle is passed in to the function as $\begin{bmatrix} minX & minY \\ maxX & maxY \end{bmatrix}$; i.e. only the top left and bottom right points are passed in.

3.4 Slicing a Spline

This task consists of two separate phases. The first phase is finding all points on the spline segment that intersect with the line segment. This can be formulated as finding the roots of a specific cubic polynomial. Consider the line segment (x_s, y_s) to (x_d, y_d). Then there is a direction vector $\mathbf{d} = (x_d - x_s, y_d - y_s)$, which is the vector along the line segment. If we take another direction vector $\mathbf{v}(t) = (x(t) - x_s, y(t) - y_s)$ for some choice of t, i.e. the vector from the start of the line segment to some point on the spline, then $(x(t), y(t))$ can potentially intersect the line segment only if $\mathbf{v}(t)$ is parallel to $\mathbf{d}$. This is equivalent to saying that if there is some vector $\mathbf{d}'$ which is perpendicular to $\mathbf{d}$, then $\mathbf{v}(t)$ must be perpendicular to $\mathbf{d}'$, or $\mathbf{v}(t) \cdot \mathbf{d}' = 0$. Given this, you can come up with a specific cubic polynomial $p(t)$ such that $p(t) = 0$ only if $\mathbf{v}(t) \cdot \mathbf{d}' = 0$, and then you can find the roots of that polynomial to determine possible points of intersection. Note that this will only check that $\mathbf{v}(t)$ intersects the infinite line through (x_s, y_s) along $\mathbf{d}$, so you still need to check that $t \in [0, 1]$ and that the intersection point lies on the line segment before accepting it as an intersection.

Once you have all intersections, you now need to generate new spline segments. For instance, if there were intersections at t_1 and t_2, $0 < t_1 < t_2 < 1$, then three new spline segments need to be generated, corresponding to the old spline evaluated from $[0, t_1]$, $[t_1, t_2]$, and $[t_2, 1]$ respectively. However, we have defined splines as being evaluated on $t \in [0, 1]$, so we need to generate new control points for each new segment that evaluate to the same curves but are evaluated on $[0, 1]$. We can do this using functions $s(t)$ that map $[0, 1]$ to $[t_{low}, t_{high}]$ for each of the given segments, then finding the coefficients of $x(s(t))$ and $y(s(t))$. Given these new coefficients, you can then solve a linear system using the spline matrices to determine which control points result in that polynomial.

The return value is a 3-D array A of size $k \times 4 \times 2$, where k is the number of spline segments. Thus,

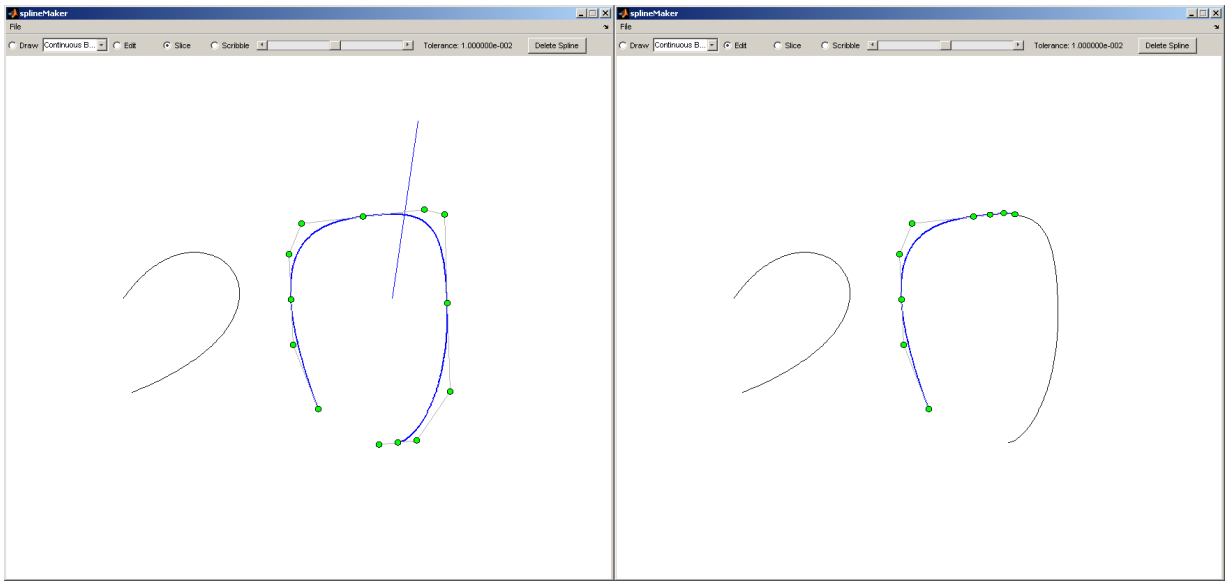

Figure 3: Slicing a spline into two pieces. Note: we have the left half of the result selected

$A(1,:,:)$ should be the control points for the first spline segment, $A(2,:,:)$ should be the control points of the second spline segment (if needed), and so on. If the line segment did not intersect the spline segment, then the original control points should be returned.

3.5 Matching a Scribble

At first glance, this seems like an easy LLS problem: find some cubic polynomial functions that best fit the input, using the methods you know for computing coefficients. However, there are a number of issues that must be considered here. One is that you don't necessarily know how many spline segments you will need ahead of time. Another is that you don't necessarily know what t value you should assign each mouse input—the simple approach above assumes that you know the t-values in order to evaluate the left hand side matrix, but in fact you don't know what the best choice should be. Put another way, you are trying to find some functions $x(t)$ and $y(t)$ that minimizes $(x_i - x(t))^2 + (y_i - y(t))^2$ for inputs x_i, y_i, but not only do you not know the coefficients of the functions, you don't even know the value of t.

The algorithm we propose you implement is the following:

```
P = {all mouse points in order}
until no more points
    S = {the first few points in P}
    until no more points
        guess values of t for each point in S, with first point
                t=0 and last point t=1
        find coefficients of best cubic function to fit points
                at chosen values of t using LLS
        for a few (2 - 3) iterations
            for each point (xi, yi) in S
```

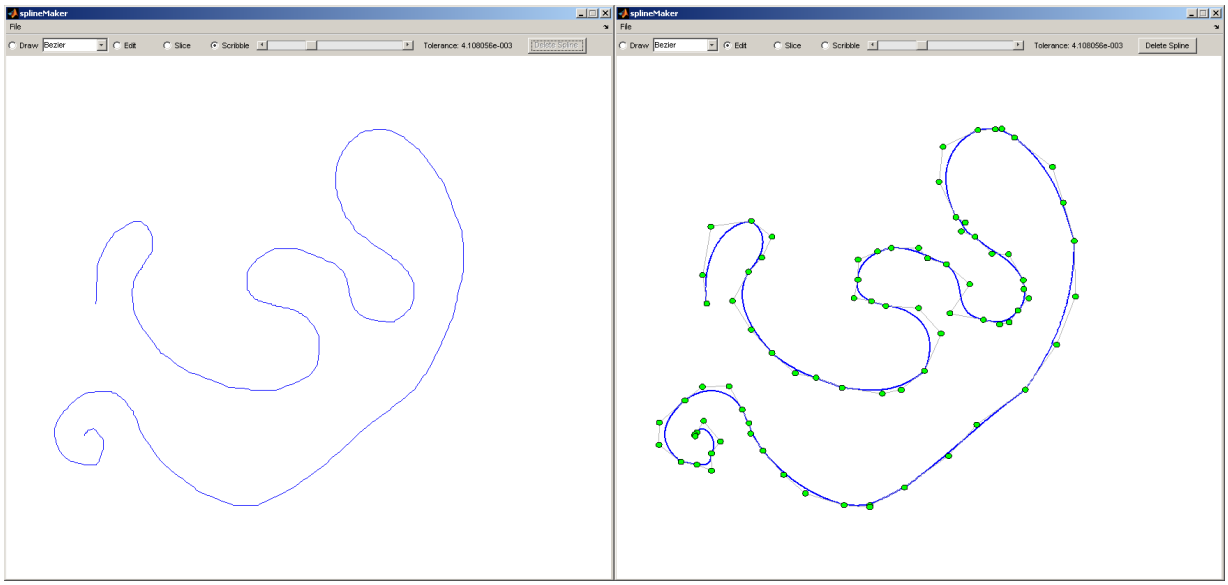

Figure 4: Scribbling with the mouse, and the resulting matched spline

```
                find tn that minimizes ||(x(t), y(t)) - (xi, yi)||
                        for current cubic coefficients
                set value of t for (xi, yi) to tn
            end
            find new coefficients of best cubic function to fit
                    points at new chosen values of t using LLS, set
                    as new coefficients to use.
        end

        err = least squares error of last fit.

        if (err < tolerance)
            bestKnownCubic = last fit
            add a couple more points from the beginning of P to S
        else
            create spline segment from best known cubic so far
            remove points in S from T
            S = {the first few points remaining in T}
        end
    end
end
```

The algorithm proposed is a greedy one—it attempts to find a spline segment that matches as many points as possible from the beginning of the mouse input, and when it can no longer keep the error below the tolerance it creates a spline segment from the last set of points that it knew was below the tolerance, removes those points from consideration, and then repeats until there are no more points in the input. In order to fit a spline to the points currently being considered, it guesses values for t for each point, fits a cubic polynomial using LLS, updates the values of t for each point to the

best values of t for the current cubic, then finds a new cubic for the new values of t, repeating this process for a few times in an attempt to find the best combination of t values for the mouse points and cubic coefficients to match the specified points.

Some things to note:

1. You need to keep track of the last known cubic fit that was under the error tolerance—you shouldn't generate the cubic segment that was over the error tolerance
2. You are generating a sequence of segments, unlike the previous parts which only dealt with a single spline segment. Remember that a spline composed of multiple segments will share some control points between segments—the last control point in a Bézier segment is also the first control point in the subsequent segment, and for a continuous Bézier segment the second control point must lie on the line between the first control point of the current segment and the third control point of the previously generated segment. If you are not fitting the first segment, make sure you set up your LLS system to account for the fact that the positions of one (or more) control points may already be known and/or constrained in some fashion.
3. Minimizing $\sqrt{(x_i - x(t))^2 + (y_i - y(t))^2}$ is equivalent to minimizing $(x_i - x(t))^2 + (y_i - y(t))^2$, which is a 6th degree polynomial, so minimizing it involves finding the roots of its derivative, which is a 5th degree polynomial.
4. There are many strategies to use for an initial guess of t for each point. One is to treat the points of the input as the vertices of line segments, compute the length of each segment (that is, the distance between neighboring points), and set the t value of a given mouse point to be the cumulative length of all line segments from the first point in your fit to the specified point, scaling all t-values so that the first point in the fit has t-value of 0 and the last has a t-value of 1.

4 Submission and FAQ

Submission will be through CMS. A FAQ page will be kept on the course website detailing any new questions and their answers brought to the attention of the course staff. Your submission should include all MATLAB code you used. Your code should be well-commented, and it should describe the mathematical problem(s) your code is solving at each step of the algorithm for each of the functions you need to implement.

5 Starred Problems

1. The time complexity of the scribble algorithm as we have presented it is rather poor. Do something clever to speed it up. Some ideas include reusing t-values for some previously fit points for a few iterations (i.e. only do the nonlinear fit of t values for new points in S on the current iteration, using known t values for previous points, and do a global t optimization over all points in S less often), or binary search to add many points into the set S at once, or something more clever than our current greedy approach to finding spline segments.

2. We are solving a nonlinear minimization problem for scribble with an iterative two-phase strategy (solving a linear system for values of t, then solving for best t values given the results of the linear system solve). Formulate the function to be minimized as a single nonlinear function in all variables (that is, the coefficients of the spline curve and all t values for each mouse point) and use the Optimization Toolbox in MATLAB to find the minimum value. Measure the performance of your solution with respect to the proposed scribble algorithm. Do you get better results? Is it faster?

   Some starting points you may want to look at in MATLAB are `fmincon` and `fminunc`, among others.