CS 1112 Spring 2021 Project 6, part B   due Thurs, May 13, by 11 PM EDT

Objectives

Completing part B of this project will reinforce your understanding of recursive functions while providing another opportunity to explore interactive graphics in MATLAB.

2   Drawing Bézier curves

Have you ever used a digital illustration tool like Inkscape or Adobe Illustrator, or drafted a technical drawing in a CAD program? These tools produce images composed of mathematically-defined lines and arcs, rather than using pixels like a digital camera or a painting program. The results are known as *vector graphics* since they are composed of arbitrary (x, y) points rather than the regular rows and columns of the *raster images* we worked with earlier. Even if you've never used any of these programs, you've already created vector graphics by calling the plot() function in MATLAB, which draws straight lines between vectors of points.

Drawing straight lines is convenient, but many things in both art and engineering are curved. How can we represent a smooth curve compactly in code, and how can we instruct a computer to draw them? The most common representation is a *cubic Bézier spline*, where segments of the curve are defined using four "control points." The curve passes through the first and last points, while the second and third points set its initial and final slope. Mathematically, the x and y coordinates of any point along the curve can be calculated from a parametric cubic polynomial that depends on the control points (see part 2.4).

One way to plot the curve, then, is to sample the curve at a large number of parameter values (think linspace()), then connect each of these samples with a line segment using plot(). But how many samples should we choose? And wouldn't this be wasteful if parts of the curve are very flat? Perhaps we can adapt how many lines we draw based on how "curvy" the curve is in different places (see *Insight* §14.3). Thinking recursively, given a curve, we could estimate how well-approximated it would be by a single line connecting its endpoints. If the approximation is good, we draw that line; if not, we divide the curve in half and ask the same question of each half. The trick is finding which control points to represent these two half-curves. Thankfully, mathematician Paul de Casteljau found a geometric procedure for doing just that.

Given control points b_1, b_2, b_3, and b_4 (each of which has an x and y coordinate), consider the lines connecting each pair of adjacent points. Let l_2 be the midpoint of the line connecting b_1 and b_2, let m be the midpoint of the line connecting b_2 and b_3, and let r_3 be the midpoint of the line connecting b_3 and b_4. Next consider the lines connecting those midpoints: let l_3 be the midpoint of the line connecting l_2 and m and let r_2 be the midpoint of the line connecting m and r_3. Finally, consider the line connecting l_3 and r_2; we'll name its midpoint n (see figure).

Figure adapted from Fischer 2000

It turns out that the sequence $\{b_1, l_2, l_3, n\}$ is the set of control points for the first half of the curve, while the sequence $\{n, r_2, r_3, b_4\}$ is the set of control points for the second half of the curve. Implementing this geometric construction in code enables us to divide our problem into two smaller subproblems—one of the key ingredients for recursion.
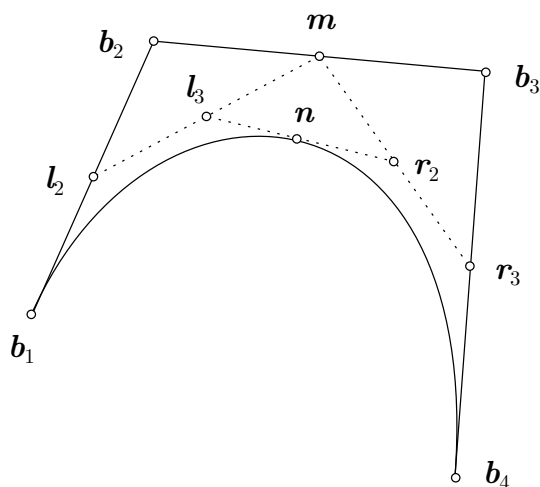
The second ingredient is identifying the base case; that is, knowing when to stop subdividing the curve. It should be safe to replace the curve with a straight line if the maximum distance between the curve and that line would be smaller than some threshold. To estimate this distance, we'll use a formula described by Roger Willcocks (author of the RoPS software). Define the points $\boldsymbol{u}$ and $\boldsymbol{v}$ as follows:

$$\boldsymbol{u} = 3\boldsymbol{b}_2 - 2\boldsymbol{b}_1 - \boldsymbol{b}_4$$
$$\boldsymbol{v} = 3\boldsymbol{b}_3 - \boldsymbol{b}_1 - 2\boldsymbol{b}_4$$

(these are vector equations, meaning they apply separately to both the x and y coordinates of each point). Then the maximum distance d between the curve and the line is bounded by

$$d \le \frac{1}{4}\sqrt{\max(u_x^2, v_x^2) + \max(u_y^2, v_y^2)}$$

So, if the value of the above expression is less than our threshold, we can draw a line between our endpoints. Otherwise, we should keep subdividing.

2.1   de Casteljau algorithm

Implement the above algorithm for drawing a cubic Bézier curve using straight lines. Your function deCasteljau.m should take the 4 control points as inputs (each point represented by a vector of length 2), as well as the desired tolerance (maximum distance between linear approximation and true curve). In addition to drawing the curve (using plot() to draw black line segments), it should return the total number of line segments used. The following function header is provided on the course website for your convenience (ignore the comments about animating control points until the next section):

```
function segments = deCasteljau(b1, b2, b3, b4, tol)
% Plot 2D Bezier curve recursively.
% b1, b2, b3, and b4 are the control points for the curve (length-2 row
% vectors in [x, y] order).  Assumes figure hold is on.  Recursively
% subdivides curve at parameter midpoint until max distance from a line
% drawn between endpoints is below `tol`.  Returns the number of line
% segments used to draw the curve.
% In addition, animate control points while error is above 10*tol to
% visualize the algorithm.
```

2.2   Interactive testing

In order to test this algorithm, we need to specify the coordinates of four control points. This is tedious and unintuitive to do numerically, so let's develop a visual, interactive procedure for defining the points. **Write a script** BezierDemo.m to perform the following tasks:

- Open a new figure window

- Set the x and y axis scaling to be equal (since we're dealing with graphics). Set axis bounds to show a rectangle approximately 2.8 units wide by 2 units tall (this nicely fills the default figure size)

- Prompt the user in the plot title to select 4 points. Accept their clicks and draw a circle marker at each point that they click.

- Set hold on and invoke your deCasteljau() function with the user-selected points as the control points and with a tolerance of 0.005.

When you run the script, hopefully you see a smooth curve connecting the first and last points that you click while bending in the direction of the middle two points. If not, don't despair; the next part may help you to visually debug what is going wrong.

2.3 Visualization

With such a geometric algorithm, it would be nice to visualize the steps it takes at each level. Not only will this improve your intuition for recursive behavior, it's a good way to check for bugs in your implementation. Make the following two changes to your `deCasteljau()` function:

1. Before subdividing the curve, check if d is larger than `10*tol`. If so, then plot the four control points as boxes (symbol code `'s'`) connected by lines.

2. At the end of the function, pause for 0.1 seconds. This will animate the curve as it is drawn.

Try out these new features using your test script and watch recursion in action!

2.4 Compare to samples

We mentioned that there is a parametric formula for directly computing the coordinates of any point along the curve. Given a parameter t between 0 (corresponding to the start of the curve) and 1 (corresponding to the end of the curve), the coordinates of this parametric curve, denoted $\boldsymbol{b}(t)$, are given by

$$\boldsymbol{b}(t) = (1-t)^3\boldsymbol{b}_1 + 3(1-t)^2t\boldsymbol{b}_2 + 3(1-t)t^2\boldsymbol{b}_3 + t^3\boldsymbol{b}_4$$

where $\boldsymbol{b}_i$ are the control points (again, this is a vector formula that applies separately to the x and y components of each point). Let's compare the results of our recursive solution to a sample-based approach using the same number of line segments. **Make the following additions** to `BezierDemo.m`:

- When you call `deCasteljau()`, save the number of line segments it used in a variable `s`. Construct an array `t` containing `s+1` points equally spaced between 0 and 1 (inclusive)

- Construct arrays `x` and `y` by evaluating the above formula at each value in `t`

- Open a new figure window with the same axes as before and plot line segments connecting the points in `x` and `y`. Ensure that your two figures are distinguished by their titles

Try out your script for several different choices of control points. The curves should look nearly identical in both windows, so this is a good way to cross-check your work. For simple curves it will be hard to see a quality difference, but remember: without recursion, we wouldn't have known how many points to sample.
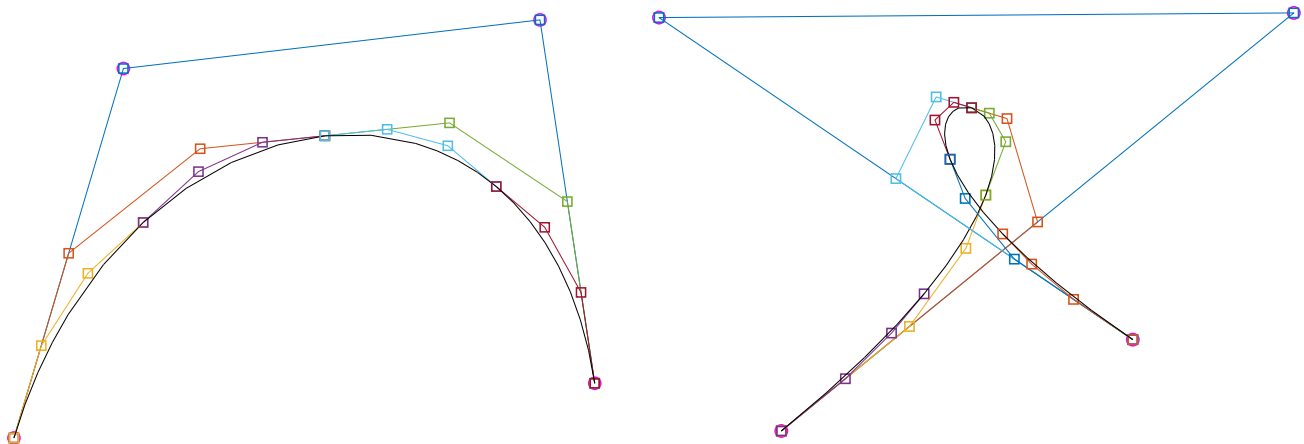
Submission

Submit your files `deCasteljau.m` and `BezierDemo.m` CMS (after registering your group), along with your submissions for Part A.