Finger Exercises 6

Bézier Curves & de Casteljau

≈ 60 minutes, plus a 10-minute bonus

CS 453 — Computer Graphics

What this is: Short drills that lock in Lecture 6 — curves you *steer* instead of derive, de Casteljau's repeated-lerp construction, the weights hiding inside it, what each handle actually controls, and how two curves are joined without a seam.

How to do it: On **paper** first, no computer. Every trace in here is deliberately kind arithmetic — sketch the control polygon before you compute anything, then mark each intermediate point on your sketch as it appears. That picture is what makes the algorithm stick; the numbers alone will not. If an item takes longer than 5 minutes, skip it, finish the rest, come back.

Part A • Do You Remember?

RECALL ≈ 5 min

A1. Wednesday's curves each arrived with a formula somebody handed you. Today's arrive with _____ you can drag. Graphics chose the second way because _____.

A2. de Casteljau in one sentence: replace the list of points by the _____ between consecutive pairs; repeat until _____ remains; that point is _____.

A3. n control points give _____ steps and a curve of degree _____. Two points draw a _____, three a _____, four a _____.

A4. Write one lerp so its weights show: $\text{lerp}(\mathbf{P}, \mathbf{Q}, t) =$ _____. The two weights always add to _____.

A5. Inside *one* call to `de_casteljau`, how many different values of t are used? _____

A6. Name the three properties from the slides: (1) _____, (2) _____, (3) _____.

A7. On a cubic, $\mathbf{P}_1$ and $\mathbf{P}_2$ are _____, not _____.

Part B • Counting the Work

COMPUTE ≈ 5 min

B1. Fill the table. "Sizes" means the length of `pts` after each step.

control pts	sizes	steps	degree	total lerps
2	_____	_____	_____	_____
3	_____	_____	_____	_____
4	_____	_____	_____	_____
7	_____	_____	_____	_____

B2. Give the total number of lerps for n control points as a formula, and say in one phrase how the cost grows.

B3. `de_casteljau` is handed a list with exactly *one* control point. How many steps run, and what comes back? What about an empty list?

B4. You sample one cubic by calling `sample_parametric(..., steps=40)`. Careful: that `steps` counts *gaps along the curve*, not de Casteljau steps. How many curve points is that, and how many lerps?

B5. Your animation has 8 such curves and runs for 150 frames. Compare the total lerps if you re-sample every curve every frame against sampling each one *once* at the start. Do the curves have to be motionless for that to be legal?

Part C • A Quadratic by Hand

TRACE ≈ 6 min

Three control points: $\mathbf{P}_0 = (2, 1)$, $\mathbf{P}_1 = (6, 9)$, $\mathbf{P}_2 = (14, 5)$. Sketch them and the two dashed segments first.

C1. Evaluate at $t = 0.25$. Write down $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{p}(0.25)$, and mark all three on your sketch.

C2. Evaluate the same curve at $t = 0.75$.

C3. Write down $\mathbf{p}(0)$ and $\mathbf{p}(1)$ *without* computing anything, and say what the trace does at $t = 0$ that makes it obvious.

C4. Now compute $\mathbf{p}(0.25)$ a second way, from the closed formula $(1-t)^2\mathbf{P}_0 + 2(1-t)t\,\mathbf{P}_1 + t^2\mathbf{P}_2$. What does the agreement prove, and what does it *not* prove?

C5. In $2(1-t)t$, where does the 2 come from? Answer using your trace, not the algebra.

Part D • A Cubic by Hand

TRACE ≈ 6 min

Four control points: $\mathbf{P}_0 = (0, 0)$, $\mathbf{P}_1 = (4, 12)$, $\mathbf{P}_2 = (16, 12)$, $\mathbf{P}_3 = (20, 0)$. Take $t = 0.25$.

D1. Step one: compute $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$.

D2. Step two, then step three. Give $\mathbf{D}$, $\mathbf{E}$ and $\mathbf{p}(0.25)$.

D3. How many lerps did you just perform, and does that match your Part B formula?

D4. Is $(4.25,\ 6.75)$ inside the quadrilateral $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3$? Say how you know *without* looking at your sketch.

D5. Look at the last segment, $\mathbf{D} \to \mathbf{E}$ — the "moving ruler". What does it become at $t = 0$, and what does that tell you about the direction the curve leaves $\mathbf{P}_0$?

Part E • Reading the Weights

COMPUTE ≈ 7 min

The cubic weights are $(1-t)^3$, $3(1-t)^2t$, $3(1-t)t^2$, t^3.

E1. Compute all four at $t = 0.2$ and add them up.

E2. At $t = 0.2$, which handle has the loudest voice? Without recomputing, give the four weights at $t = 0.8$.

E3. Write the four weights at $t = 0.5$ and simplify $\mathbf{p}(0.5)$ into a single fraction. Then apply it to Part D's cubic.

E4. The interior weight of a *quadratic*, $2(1-t)t$, peaks at $t = \frac{1}{2}$ with value $\frac{1}{2}$. The interior weight of a *cubic*, $3(1-t)^2t$, peaks at $t = \frac{1}{3}$. Compute its peak value, and say what both numbers mean for whether a curve can ever reach an interior handle.

E5. Every control point of a curve is shifted by the same vector $(+60, -25)$. What happens to the curve, and which single fact about the weights guarantees it?

Part F • What the Handles Actually Do

CLASSIFY ≈ 6 min

Predict first, in words. One sentence of reason per item.

F1. You drag $\mathbf{P}_1$ of a cubic upward by 30 pixels. Which parts of the curve move, and which do not?

F2. Two cubics share the same $\mathbf{P}_0$ and $\mathbf{P}_3$, but the second has both handles three times further out along the same directions. How do the curves differ?

F3. All four control points lie on one straight line. What is drawn, and what in the algorithm forces it?

F4. A *quadratic* has $\mathbf{P}_1$ exactly at the midpoint of $\mathbf{P}_0\mathbf{P}_2$. Does the curve pass through $\mathbf{P}_1$? Reconcile your answer with "the curve does not pass through its interior handles".

F5. You reverse the control list, feeding $[\mathbf{P}_3, \mathbf{P}_2, \mathbf{P}_1, \mathbf{P}_0]$ instead. What curve comes out?

Part G • The Rubber Band Earns Its Keep

DESIGN ≈ 5 min

The canvas is 800×600: x from 0 to 799, y from 0 to 599, with y growing down.

G1. Curve U has handles $(820, 100)$, $(960, 180)$, $(910, 420)$, $(830, 500)$. Draw it or skip it? Why is that decision *exact* — no risk of dropping something visible?

G2. Curve V has handles $(150, 100)$, $(320, 700)$, $(560, 700)$, $(720, 100)$. What does the hull test report? Now use the midpoint formula from E3 to find $\mathbf{p}(0.5)$, and say what actually happens.

G3. If all four control points are inside the canvas, is the whole curve guaranteed inside? One line of reasoning.

G4. Most renderers test the axis-aligned *bounding box* of the control points instead of the true hull. Which is looser, and why is looser acceptable here?

G5. State the asymmetry that makes culling safe, as a rule you could apply to any future accelerator.

Part H • Joining Two Curves

DESIGN ≈ 6 min

Curve 1 is the cubic $\mathbf{P}_0 = (40, 300)$, $\mathbf{P}_1 = (90, 140)$, $\mathbf{P}_2 = (210, 120)$, $\mathbf{P}_3 = (300, 200)$. Curve 2 has handles $\mathbf{Q}_0 \ldots \mathbf{Q}_3$ and must continue where curve 1 stops.

H1. Where must $\mathbf{Q}_0$ be so that there is no gap? Why is the join exact rather than "close enough at this resolution"?

H2. For a *smooth* join, where must $\mathbf{Q}_1$ lie? Give the $\mathbf{Q}_1$ that continues at the same pace, i.e. with $\mathbf{Q}_0\mathbf{Q}_1$ the same length and direction as $\mathbf{P}_2\mathbf{P}_3$.

H3. You instead pick $\mathbf{Q}_1 = (345, 240)$. Is the join still smooth? Does anything change for an object flying along the pair?

H4. Give one $\mathbf{Q}_1$ that produces a deliberate sharp corner, and name something that wants one.

H5. Why do fonts chain many short cubics instead of using one curve with forty control points? Two reasons, both already on today's slides.

Part I • Find the Bug

FIX ≈ 8 min

Each is *almost* `de_casteljau`. Name the bug and give the fix.

I1. Symptom: the first curve of the run looks right; every curve after it is wrong, and the caller's control list has been emptied.

```
pts = control
while len(pts) > 1:
    for i in range(len(pts) - 1):
        pts[i] = pts[i] + (pts[i+1] - pts[i]) * t
    pts.pop()
```

Bug & fix: _____

I2. Symptom: the program hangs; no frame is ever written.

```
while len(pts) > 1:
    pts = [p + (q - p) * t
           for p, q in zip(pts, pts)]
```

Bug & fix: _____

I3. Symptom: `IndexError: list index out of range` on every call.

```
while len(pts) > 0:
    pts = [p + (q - p) * t
           for p, q in zip(pts, pts[1:])]
return pts[0]
```

Bug & fix: _____

I4. Symptom: `cubic_bezier(p0,p1,p2,p3,0.0) == p0` passes and `... 1.0) == p3` passes, but the drawn curve clings to $\mathbf{P}_0$ far too long before whipping across.

```
while len(pts) > 1:
    pts = [p + (q - p) * t
           for p, q in zip(pts, pts[1:])]
    t = t * t
```

Bug & fix: _____

I5. Symptom: endpoints are exact, but the middle of every cubic sags toward the top-left corner of the canvas.

```
u = 1 - t
return (p0 * u**3 + p1 * 2*u*u*t
        + p2 * 3*u*t*t + p3 * t**3)
```

Bug & fix: _____

Part J • Straight Into A2

EXPLAIN ≈ 6 min

A2 was released today and is due **Friday 18 September, 11:59 pm**.

J1. `T10` is worth 10 points and asks for three functions: `de_casteljau`, `quadratic_bezier` and `cubic_bezier`. Roughly how long should the last two be, and why?

J2. The handout says the checker tests degrees 0 through 4. Name the two inputs at the ends of that range that a hardcoded cubic cannot survive, and what your loop must do with each.

J3. Write the two lines that put an object at parameter u on a fixed cubic during frame i of a clip, and say which of them belongs inside the frame loop.

J4. That one B1 checklist line — "something moves along a curve you evaluate" — is scored 3 points. Using Part B's numbers, how many lerps does the *motion* cost across 150 frames, next to the 246 that draw the path?

J5. The handout's cheap upgrade is to make the object face where it is going. Give the recipe in one line, and name the one value of u where it needs care.

Part K • Bonus: Cut the Curve in Two ★

DESIGN ≈ 10 min

For the brave. You have been treating one de Casteljau evaluation as a way to get *one point*. It quietly computes something much larger, and finding it is the last big idea about this algorithm.

Take the cubic $\mathbf{P}_0 = (0,0)$, $\mathbf{P}_1 = (0,8)$, $\mathbf{P}_2 = (8,8)$, $\mathbf{P}_3 = (8,0)$ — an arch. Use $t = \frac{1}{2}$.

K1. Run the full evaluation. Write down every point the algorithm produces — $\mathbf{A}, \mathbf{B}, \mathbf{C}$, then $\mathbf{D}, \mathbf{E}$, then the survivor $\mathbf{F}$ — and mark all six on a sketch.

K2. Now collect two chains out of that trace: the points you met walking down one side, $[\mathbf{P}_0, \mathbf{A}, \mathbf{D}, \mathbf{F}]$, and down the other, $[\mathbf{F}, \mathbf{E}, \mathbf{C}, \mathbf{P}_3]$. Write both out as coordinates.

K3. **The claim:** the left chain is the control polygon of the *first half* of the original curve. Test it. Evaluate the left chain at $t = \frac{1}{2}$, evaluate the *original* four points at $t = \frac{1}{4}$, and compare.

K4. Where do those two chains sit in your sketch, and what have you actually been given for the price of one evaluation?

K5. Two payoffs, both things you have already met today. First: compare the bounding box of the original four handles with that of the left chain. Second: what happens if you keep subdividing until a chain is nearly straight?

Stuck on more than three items? Re-read the Lecture 6 slides — everything above Part K comes straight from them. Start with "The Whole Algorithm, in One Sentence" and the three property slides; almost every answer here is one of those three facts wearing different clothes.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides.

A1. **handles** (control points). Because a steered curve can be *edited*: nudge a handle and the curve follows. With a derived formula, every nudge means solving for new coefficients from scratch — and nobody sketches a swoosh by choosing a, b, c, d.

A2. the **lerps**; until **one point** remains; that point is **on the curve** at parameter t.

A3. $n-1$ steps, degree $n-1$. Two: a **line**. Three: a **parabola** (quadratic). Four: a **cubic**.

A4. $(1-t)\mathbf{P} + t\,\mathbf{Q}$, which is just $\mathbf{P} + t(\mathbf{Q}-\mathbf{P})$ regrouped. The weights $(1-t)$ and t add to **1** — that single fact is where all three properties come from.

A5. **One.** Every lerp in every step uses the same t. That shared dial is the whole algorithm; vary it between steps and you are computing some other curve entirely.

A6. (1) the **ends are pinned**: $\mathbf{p}(0) = \mathbf{P}_0$, $\mathbf{p}(1) = \mathbf{P}_n$. (2) the curve stays inside the **convex hull** of the handles (the rubber band). (3) the **first and last segments of the control polygon are the launch and arrival directions**.

A7. **handles**, not **stops**. The curve leans toward them and normally never touches them.

B1. 2: $2 \to 1$; 1 step; degree 1; 1 lerp. 3: $3 \to 2 \to 1$; 2 steps; degree 2; $2+1=3$ lerps. 4: $4 \to 3 \to 2 \to 1$; 3 steps; degree 3; $3+2+1=6$ lerps. 7: $7 \to 6 \to \cdots \to 1$; 6 steps; degree 6; $6+5+4+3+2+1=21$ lerps.

B2. $(n-1)+(n-2)+\cdots+1 = \dfrac{n(n-1)}{2}$. It grows **quadratically** in the number of control points — another reason real work chains many cubics (6 lerps each) rather than raising the degree.

B3. **Zero** steps — `len(pts) > 1` is already false — and the single point comes straight back out. That is the degree-0 "curve": a stationary dot. An **empty** list is a caller error, not a curve: raise. The A2 checker tests degrees 0 through 4 precisely to see whether your loop handles the ends of that range.

B4. 41 points (`steps` + 1 — Wednesday's fence-post rule, since `steps` counts gaps and both ends are included) at 6 lerps each, so 246 lerps for the whole curve. Two different countings are in play and it is worth keeping them apart: 40 sampler *gaps* along the curve, and 3 de Casteljau *steps* inside every one of the 41 points.

B5. Re-sampling every frame: $246 \times 8 \times 150 = \mathbf{295{,}200}$ lerps. Sampling once: $246 \times 8 = \mathbf{1{,}968}$ — a 150× saving for a one-line change. It is legal exactly when the *control points* do not change; the curve is a function of its handles alone, so a fixed handle set means a fixed point list. A moving object riding a fixed path is fine — you re-evaluate one point per frame, not the whole path.

C1. $\mathbf{A} = (2,1) + 0.25(4,8) = (3,3)$; $\mathbf{B} = (6,9) + 0.25(8,-4) = (8,8)$; $\mathbf{p}(0.25) = (3,3) + 0.25(5,5) = (4.25,\ 4.25)$.

C2. $\mathbf{A} = (2,1) + 0.75(4,8) = (5,7)$; $\mathbf{B} = (6,9) + 0.75(8,-4) = (12,6)$; $\mathbf{p}(0.75) = (5,7) + 0.75(7,-1) = (10.25,\ 6.25)$.

C3. $\mathbf{p}(0) = (2,1) = \mathbf{P}_0$ and $\mathbf{p}(1) = (14,5) = \mathbf{P}_2$. At $t=0$ every lerp returns its *left* input, so each step simply drops the last entry: $[\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2] \to [\mathbf{P}_0, \mathbf{P}_1] \to [\mathbf{P}_0]$. At $t=1$ every lerp returns its *right* input and the first entry is dropped instead.

C4. Weights at $t=0.25$: 0.5625, 0.375, 0.0625. Then $x = 0.5625(2) + 0.375(6) + 0.0625(14) = 1.125 + 2.25 + 0.875 = 4.25$ and $y = 0.5625(1) + 0.375(9) + 0.0625(5) = 0.5625 + 3.375 + 0.3125 = 4.25$. Same point. It proves the loop and the formula are the *same curve* — the formula is what you get by substituting the first-step lerps into the second. It does **not** make the formula preferable: it is frozen at degree 2, while the loop you just ran is degree-agnostic.

C5. $\mathbf{P}_1$ is used **twice** in step one — once as the right end of $\mathbf{A}$ and once as the left end of $\mathbf{B}$. Each appearance carries weight $(1-t)t$ into the final answer, and the two copies add. $\mathbf{P}_0$ and $\mathbf{P}_2$ each appear once, so they get no multiplier.

D1. $\mathbf{A} = (0,0) + 0.25(4,12) = (1,3)$; $\mathbf{B} = (4,12) + 0.25(12,0) = (7,12)$; $\mathbf{C} = (16,12) + 0.25(4,-12) = (17,9)$.

D2. $\mathbf{D} = (1,3) + 0.25(6,9) = (2.5,\ 5.25)$; $\mathbf{E} = (7,12) + 0.25(10,-3) = (9.5,\ 11.25)$; $\mathbf{p}(0.25) = (2.5, 5.25) + 0.25(7,6) = (4.25,\ 6.75)$.

D3. $3+2+1=6$, and $n(n-1)/2 = 4 \cdot 3/2 = 6$. Yes.

D4. Yes, and it must be. Every step produces weighted averages of points that were already inside the hull, and an average of points inside a convex region stays inside it. Property 2 is not a picture you check afterwards — it is forced by the fact that every weight lies in $[0,1]$ and the weights sum to 1.

D5. At $t=0$ the ruler is exactly the segment $\mathbf{P}_0\mathbf{P}_1$ (every lerp returns its left input, so $\mathbf{D} = \mathbf{P}_0$ and $\mathbf{E} = \mathbf{P}_1$). The curve point rides that ruler, so it departs **straight at $\mathbf{P}_1$** — which is Property 3. (In general the ruler $\mathbf{DE}$ is tangent to the curve at $\mathbf{p}(t)$; the endpoint case is just the one you can read off without calculus.)

E1. $(0.8)^3 = 0.512$; $3(0.8)^2(0.2) = 0.384$; $3(0.8)(0.2)^2 = 0.096$; $(0.2)^3 = 0.008$. Sum = 1.000 exactly, since $[(1-t)+t]^3 = 1$ for every t.

E2. $\mathbf{P}_0$, at 0.512 — early in the sweep the start point dominates. At $t=0.8$ the list is the mirror image: 0.008, 0.096, 0.384, 0.512, so $\mathbf{P}_3$ dominates. The weight set is symmetric under $t \leftrightarrow 1-t$ together with reversing the points.

E3. $\frac{1}{8},\ \frac{3}{8},\ \frac{3}{8},\ \frac{1}{8}$, so $\mathbf{p}(0.5) = \dfrac{\mathbf{P}_0 + 3\mathbf{P}_1 + 3\mathbf{P}_2 + \mathbf{P}_3}{8}$. For Part D: $x = (0 + 12 + 48 + 20)/8 = 10$ and $y = (0 + 36 + 36 + 0)/8 = 9$, so $\mathbf{p}(0.5) = (10, 9)$. Worth memorising — it is the fastest hand check there is that an implementation works.

E4. $3 \cdot \frac{1}{3} \cdot (\frac{2}{3})^2 = \frac{4}{9} \approx 0.444$. Neither interior weight ever reaches 1, or even $\frac{1}{2}$ in the cubic case — so at every t the other handles still hold a majority of the vote and the curve cannot arrive at $\mathbf{P}_1$. That is the precise version of "handles, not stops".

E5. The whole curve shifts by exactly $(+60, -25)$ — same shape, new place. Because the weights sum to 1, adding a constant $\mathbf{v}$ to every control point adds $(\sum w_i)\mathbf{v} = \mathbf{v}$ to the result. The same argument works for scaling and rotation, which is why editors transform *the four handles* and never touch the sampled points. Monday's matrices are about to make that the default move.

F1. Both endpoints stay exactly put — $\mathbf{P}_1$'s weight is 0 at $t = 0$ and $t = 1$. *Every* interior point moves, because $3(1-t)^2 t > 0$ for all t strictly between 0 and 1; the shift is largest around $t = \frac{1}{3}$, where that weight peaks. So a handle is a global pull with a local emphasis, not a local edit.

F2. They leave and arrive along the *same* directions — direction is set by where the handle points, and that is unchanged. The long-handled one obeys those directions for *longer*, so it swings much wider before turning back: the same launch angle, a far more dramatic swoop.

F3. A straight segment. A lerp of two points on a line lands on that line, so step one stays on it, and so does every step after. The curve never leaves the line — a degenerate but perfectly legal Bezier. (It is also the tightest possible convex hull: the line itself.)

F4. Here it does — all three points are collinear, so the curve is the straight segment $\mathbf{P}_0\mathbf{P}_2$, and the midpoint lies on it. Check: $\mathbf{p}(0.5) = \frac{1}{4}\mathbf{P}_0 + \frac{1}{2}\mathbf{P}_1 + \frac{1}{4}\mathbf{P}_2$, and with $\mathbf{P}_1 = \frac{1}{2}(\mathbf{P}_0 + \mathbf{P}_2)$ that is $\mathbf{P}_1$. This is exactly why the slides say *usually* does not pass through: the curve can only touch an interior handle when the handles are degenerate enough that the point was already a weighted average of the others.

F5. The very same curve, traced backwards: $\mathbf{p}_{\text{rev}}(t) = \mathbf{p}(1-t)$. Reversing the list swaps the two ends of every lerp, and $\text{lerp}(\mathbf{Q}, \mathbf{P}, t) = \text{lerp}(\mathbf{P}, \mathbf{Q}, 1-t)$. The pixels are identical; only the direction of travel changes — which matters the moment something is *moving* along it.

G1. Skip it, without touching a pixel. Every handle has $x \geq 820$, so the whole hull lies right of the canvas, and by Property 2 the curve lies inside the hull. A curve that cannot leave a region entirely off-screen cannot be on-screen. The rejection is exact.

G2. Two handles sit at $y = 700$, below the canvas, so the hull crosses the bottom edge and the test can only say "may be partly off-screen — draw it". But $\mathbf{p}(0.5) = \big(150 + 3(320) + 3(560) + 720,\ 100 + 3(700) + 3(700) + 100\big)/8 = (3510/8,\ 4400/8) = (438.75,\ 550)$. The four y values read $100, 700, 700, 100$ — symmetric — so $t = 0.5$ is the curve's lowest point. $550 < 600$: the curve is entirely on-screen. The hull was **conservative** — it over-reported, because the curve never reaches its handles.

G3. Yes. A rectangle is convex, so it contains the hull of any points inside it; the curve is inside its hull; therefore the curve is inside the rectangle. This direction is exact too — and it is the cheap test worth running first.

G4. The box is looser — it is the smallest *rectangle* containing the points, and the hull is the smallest *convex set*, so the hull is inside the box. But the box costs four `min`/`max` scans and no geometry at all. Looser only ever costs you a wasted draw; it never hides a visible curve, which is the one error that would be unacceptable.

G5. The test may say **draw** for something invisible — that only wastes time. It must **never** say **skip** for something visible — that corrupts the picture. So every bound must *contain* the true object. This is exactly the contract bounding volumes will sign in the ray-tracing weeks.

H1. $\mathbf{Q}_0 = \mathbf{P}_3 = (300, 200)$. Property 1 says curve 1 ends *exactly* at $\mathbf{P}_3$ and curve 2 starts *exactly* at $\mathbf{Q}_0$ — these are the two parameter values where the weights are $(1, 0, 0, 0)$ and $(0, 0, 0, 1)$, so no arithmetic error can creep in. Share one point and the seam is invisible at any zoom.

H2. On the straight line through $\mathbf{P}_2$ and $\mathbf{P}_3$, on the far side of the join, so that $\mathbf{P}_2$, $\mathbf{Q}_0$, $\mathbf{Q}_1$ are collinear in that order. Curve 1 arrives along $\mathbf{P}_2 \to \mathbf{P}_3$ and curve 2 departs along $\mathbf{Q}_0 \to \mathbf{Q}_1$, so lining those up makes the directions agree. With $\mathbf{P}_3 - \mathbf{P}_2 = (90, 80)$: $\mathbf{Q}_1 = (300, 200) + (90, 80) = (390, 280)$.

H3. $(345, 240) = \mathbf{P}_3 + 0.5(90, 80)$, still on the same ray, so the *shape* is still smooth — no visible corner. But the handle is half as long, so curve 2 leaves more slowly: an object driven at a steady t would lurch down in speed at the seam. Matching *direction* fixes the drawing; matching *length* as well is what fixes the motion.

H4. Anything off that line — $\mathbf{Q}_1 = (300, 380)$, say, which sends curve 2 straight down while curve 1 arrived heading up and to the right. Wanted corners are everywhere in type: the apex of an 'A', the point of a serif, the join between the bowl and the stem of a 'b'. Smoothness is a choice you make per join, not a law.

H5. (1) **No local control**: every weight is nonzero for all interior t, so on a degree-39 curve moving one handle nudges the entire letter — unusable for a designer. Chained cubics confine an edit to one or two pieces. (2) **Cost and looseness**: $n(n-1)/2$ lerps per point grows quadratically, and a 40-point hull is enormous, so culling stops helping. Short pieces are cheap, tightly bounded and independently editable.

I1. `pts = control` does not copy — it is a second name for the caller's list, so the in-place writes and `pop` destroy the argument. One call "works", then the handles are gone. Fix: `pts = list(control)`. The in-place arithmetic itself is fine, because each step reads `pts[i+1]` before that slot is written.

I2. `zip(pts, pts)` pairs every point with *itself*, so each lerp returns `p` unchanged and the list never shrinks — an infinite loop. Fix: `zip(pts, pts[1:])`, which pairs *consecutive* points and is exactly what turns n into $n-1$.

I3. The guard runs one step too many. With one point left, `pts[1:]` is empty, so the comprehension produces `[]`, the loop then stops on length 0, and `pts[0]` has nothing to return. Fix: `while len(pts) > 1` — stop at the survivor, do not consume it.

I4. The dial is squared between steps, so the steps run at t, t^2, t^4 instead of one shared t. Both endpoint tests still pass because $0^2 = 0$ and $1^2 = 1$ — the two values the bug cannot touch — which is why the asserts are silent. Everything in between is dragged toward the left input of each lerp, i.e. toward $\mathbf{P}_0$. Fix: delete the line. One evaluation uses **one** t, and a test at an interior value such as $t = 0.5$ against $(\mathbf{P}_0 + 3\mathbf{P}_1 + 3\mathbf{P}_2 + \mathbf{P}_3)/8$ would have caught it instantly.

I5. The second coefficient is 2 where the cubic needs 3 (the quadratic's number, typed from memory). The weights no longer sum to 1 — at $t = 0.5$ they give $0.125 + 0.25 + 0.375 + 0.125 = 0.875$ — so the result is only 87.5% of a proper average and every interior point is pulled toward the origin, which is the top-left of the canvas. Endpoints survive because the broken term is 0 at both ends. Fix: use 3 — or better, delete the whole expression and `return de_casteljau([p0,p1,p2,p3], t)`, which has no coefficient to mistype at any degree.

J1. One line each: `return de_casteljau([p0,p1,p2], t)` and `return de_casteljau([p0,p1,p2,p3], t)`. The algorithm lives in exactly one place, so there is exactly one place for a bug to live — and exactly one function to fix when you find it.

J2. A **single** control point (degree 0): zero steps run and that point comes back — `de_casteljau([p], t)` must equal `p` for every t. And a **five**-point list (degree 4): four steps, 10 lerps, no special case. A function built from four hardcoded terms handles neither.

J3. `u = i / (FRAMES - 1)` and `p = curves.cubic_bezier(a, b, c, d, u)`. Both belong *inside* the loop — they depend on i. What belongs *outside* is the path's point list: `pts = sample_parametric(kite, 0.0, 1.0, 64)`, sampled once because the handles never move.

J4. One point per frame at 6 lerps each: 900 lerps for the whole clip, against 246 paid once for the path — about 1,150 lerps in total. Motion along a curve is essentially free; what costs is re-sampling paths that never changed.

J5. `heading = (p(u + du) - p(u)).normalized()` for a small `du`, then orient the object along it — the difference of two nearby curve points *is* the direction of travel. At $u = 1$ (and anywhere past it) $u + du$ leaves the curve, so look *backwards* there instead: `p(u) - p(u - du)`. Same direction, no clamping artefact on the last frame.

K1. $\mathbf{A} = (0,4)$, $\mathbf{B} = (4,8)$, $\mathbf{C} = (8,4)$; $\mathbf{D} = (2,6)$, $\mathbf{E} = (6,6)$; $\mathbf{F} = (4,6) = \mathbf{p}(0.5)$. (Check against E3: $(0+0+24+8)/8 = 4$ and $(0+24+24+0)/8 = 6$.)

K2. Left chain: $[(0,0),\ (0,4),\ (2,6),\ (4,6)]$. Right chain: $[(4,6),\ (6,6),\ (8,4),\ (8,0)]$. Each is four points — a cubic's worth.

K3. Left chain at $t = 0.5$: step one $(0,2)$, $(1,5)$, $(3,6)$; step two $(0.5,3.5)$, $(2,5.5)$; survivor $(\mathbf{1.25},\ \mathbf{4.5})$.
Original at $t = 0.25$: step one $(0,2)$, $(2,8)$, $(8,6)$; step two $(0.5,3.5)$, $(3.5,7.5)$; survivor $(\mathbf{1.25},\ \mathbf{4.5})$.
Identical. Halfway along the first half is quarter of the way along the whole — exactly what "this is the first half" has to mean.

K4. They are the two outer edges of the triangle of intermediate points: $\mathbf{P}_0 \to \mathbf{A} \to \mathbf{D} \to \mathbf{F}$ down the left side and $\mathbf{F} \to \mathbf{E} \to \mathbf{C} \to \mathbf{P}_3$ back up the right. One evaluation at t therefore yields not just $\mathbf{p}(t)$ but the control points of **both pieces** of the curve, split at that t. This is *subdivision*, and it is free.

K5. **Culling.** Original box: $x \in [0,8]$, $y \in [0,8]$, area 64. Left chain's box: $x \in [0,4]$, $y \in [0,6]$, area 24. Each split tightens the bound sharply, so subdivide-then-cull discards off-screen work far faster than one loose test on the whole curve.
Flattening. Keep splitting and each control polygon gets straighter, because the curve stays inside a hull that keeps shrinking. Once a chain is straight to within half a pixel, *draw the chain* — no sampling, no chosen `steps`, and the accuracy is guaranteed rather than guessed. That is how real renderers turn font outlines into line segments, and it is the honest version of the "curves get tessellated" line from the end of the lecture.