14. Parametric curves

Reading

Required:

- Watt, 3 (intro), 3.1, 3.2 (intro), 3.2.1, 3.2.2

Optional

- Bartels, Beatty, and Barsky. *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*, 1987.
- Farin. *Curves and Surfaces for CAGD: A Practical Guide*, 4th ed., 1997.

Curves before computers

The "loftsman's spline":

- long, narrow strip of wood or metal
- shaped by lead weights called "ducks"
- gives curves with second-order continuity, usually

Used for designing cars, ships, airplanes, etc.

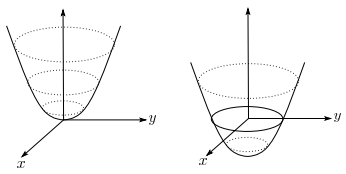
But curves based on physical artifacts can't be replicated well, since there's no exact definition of what the curve is.

Around 1960, a lot of industrial designers were working on this problem.

Today, curves are easy to manipulate on a computer and are used for CAD, art, animation, …

Mathematical curve representation

- Explicit $y=f(x)$
  - what if the curve isn't a function, e.g., a circle?
- Implicit $g(x,y) = 0$



- Parametric $(x(u),y(u))$
  - For the circle:
    $x(u) = \cos 2\pi u$
    $y(u) = \sin 2\pi u$

Parametric polynomial curves

We'll use parametric curves, *Q(u)=(x(u),y(u))*, where the functions are all polynomials in the parameter.

$$x(u) = \sum_{k=0}^{n} a_k u^k$$

$$y(u) = \sum_{k=0}^{n} b_k u^k$$

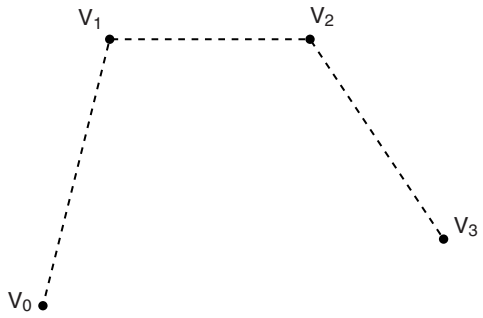
Advantages:

- easy (and efficient) to compute
- infinitely differentiable

We'll also assume that *u* varies from 0 to 1.

de Casteljau's algorithm

Recursive interpolation:

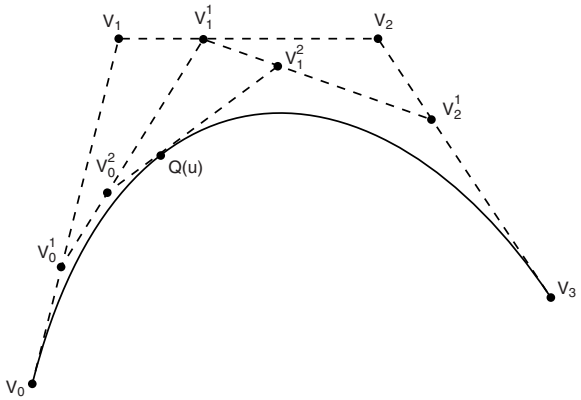


What if *u*=0?

What if *u*=1?

de Casteljau's algorithm, cont'd

Recursive notation:



What is the equation for $\mathbf{V}_0^1$?

Finding Q(u)

Let's solve for **Q**(u):

$$\mathbf{V}_0^1 = (1-u)\mathbf{V}_0 + u\mathbf{V}_1$$
$$\mathbf{V}_1^1 = (1-u)\mathbf{V}_1 + u\mathbf{V}_2$$
$$\mathbf{V}_2^1 = (1-u)\mathbf{V}_2 + u\mathbf{V}_3$$

$$\mathbf{V}_0^2 = (1-u)\mathbf{V}_0^1 + u\mathbf{V}_1^1$$
$$\mathbf{V}_1^2 = (1-u)\mathbf{V}_1^1 + u\mathbf{V}_2^1$$

$$\begin{aligned}\mathbf{Q}(u) &= (1-u)\mathbf{V}_0^2 + u\mathbf{V}_1^2 \\ &= (1-u)[(1-u)\mathbf{V}_0^1 + u\mathbf{V}_1^1] + u[(1-u)\mathbf{V}_1^1 + u\mathbf{V}_2^1] \\ &= (1-u)[(1-u)\{(1-u)\mathbf{V}_0 + u\mathbf{V}_1\} + u\{(1-u)\mathbf{V}_1 + u\mathbf{V}_2\}] + \ldots \\ &= (1-u)^3\mathbf{V}_0 + 3u(1-u)^2\mathbf{V}_1 + 3u^2(1-u)\mathbf{V}_2 + u^3\mathbf{V}_3\end{aligned}$$

Finding Q(u) (cont'd)

In general,

$$\mathbf{Q}(u) = \sum_{i=0}^{n} \binom{n}{i} u^i (1-u)^{n-i} \mathbf{V}_i$$

where "*n* choose *i*" is:

$$\binom{n}{i} = \frac{n!}{(n-i)!i!}$$

This defines a class of curves called **Bézier curves**.

What's the relationship between the number of control points and the degree of the polynomials?

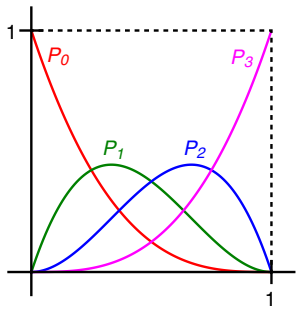
Bernstein polynomials

The coefficients of the control points are a set of functions called the **Bernstein polynomials:**

$$\mathbf{Q}(u) = \sum_{i=0}^{n} P_i(u) \mathbf{V}_i$$

For degree 3, we have:

$$P_0(u) = (1-u)^3$$
$$P_1(u) = 3u(1-u)^2$$
$$P_2(u) = 3u^2(1-u)$$
$$P_3(u) = u^3$$



Useful properties on the interval [0,1]:

- each is between 0 and 1
- sum of all four is exactly 1 (a.k.a., a "partition of unity")

These together imply that the curve lies within the **convex hull** of its control points.

Matrix form of Bezier curves

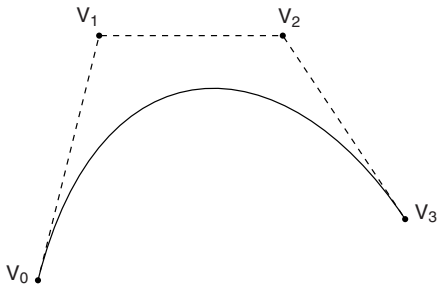
We can write out the equation of a Bezier curve in matrix form:

$$\mathbf{Q}(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_0^T \\ \mathbf{V}_1^T \\ \mathbf{V}_2^T \\ \mathbf{V}_3^T \end{bmatrix}$$

$$= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \mathbf{M}_{\text{Bezier}} \mathbf{V}$$

Displaying Bézier curves

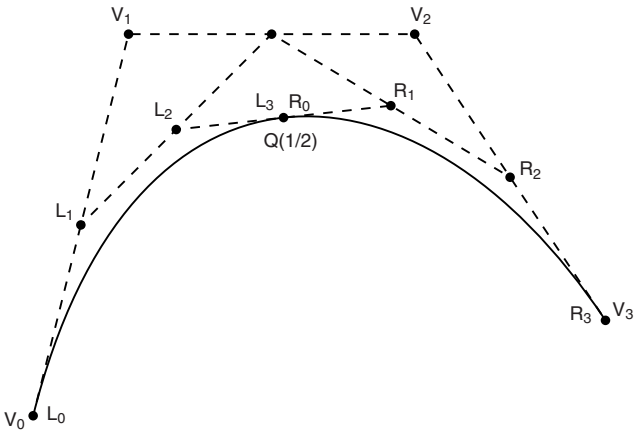
How could we draw one of these things?



It would be nice if we had an *adaptive* algorithm, that would take into account flatness.

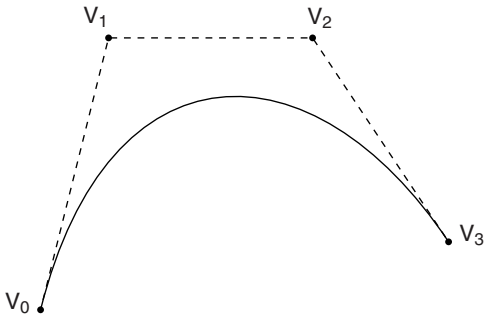
```
DisplayBezier( V0, V1, V2, V3 )
begin
    if ( FlatEnough( V0, V1, V2, V3 ) )
        Line( V0, V3 );
    else
        something;
end;
```

Subdivide and conquer




```
DisplayBezier( V0, V1, V2, V3 )
begin
    if ( FlatEnough( V0, V1, V2, V3 ) )
        Line( V0, V3 );
    else
        Subdivide(V[]) ⇒ L[], R[]
        DisplayBezier( L0, L1, L2, L3 );
        DisplayBezier( R0, R1, R2, R3 );
end;
```


Testing for flatness



Compare total length of control polygon to length of line connecting endpoints:

$$\frac{|\mathbf{V}_0-\mathbf{V}_1|+|\mathbf{V}_1-\mathbf{V}_2|+|\mathbf{V}_2-\mathbf{V}_3|}{|\mathbf{V}_0-\mathbf{V}_3|}<1+\varepsilon$$

More complex curves

Suppose we want to draw a more complex curve.

Why not use a high-order Bézier?

Instead, we'll splice together a curve from individual segments that are cubic Béziers.

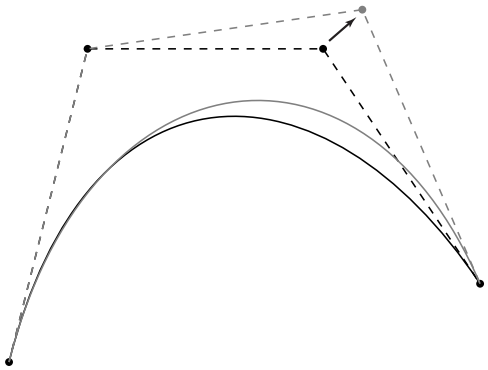
Why cubic?

There are three properties we'd like to have in our newly constructed splines…

Local control

One problem with Béziers is that every control point affects every point on the curve (except the endpoints).

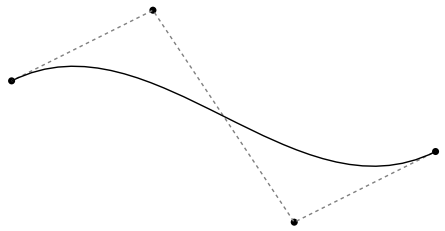
Moving a single control point affects the whole curve!



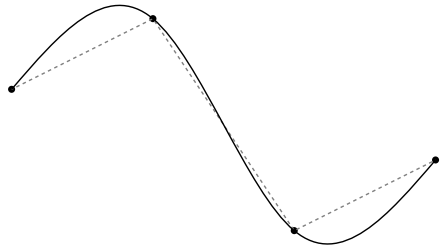
We'd like our spline to have **local control**, that is, have each control point affect some well-defined neighborhood around that point.

Interpolation

Bézier curves are **approximating**. The curve does not (necessarily) pass through all the control points. Each point pulls the curve toward it, but other points are pulling as well.



We'd like to have a spline that is **interpolating**, that is, that always passes through every control point.

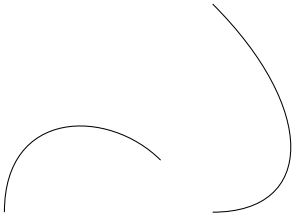


Continuity

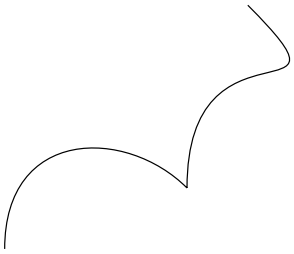
We want our curve to have **continuity**. There shouldn't be an abrupt change when we move from one segment to the next.

There are nested degrees of continuity:

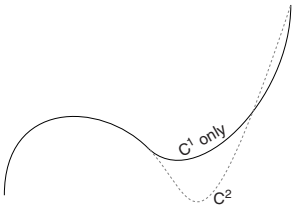
C^{-1}:



C^0:



C^1, C^2:



C^3, C^4, …:

Ensuring continuity

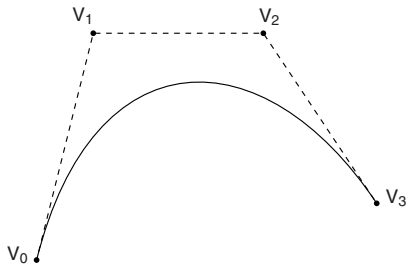
Let's look at continuity first.

Since the functions defining a Bézier curve are polynomial, all their derivatives exist and are continuous.

Therefore, we only need to worry about the derivatives at the endpoints of the curve.

Derivatives at the endpoints

$$\mathbf{Q}'(0) = 3(\mathbf{V}_1 - \mathbf{V}_0)$$
$$\mathbf{Q}'(1) = 3(\mathbf{V}_3 - \mathbf{V}_2)$$
$$\mathbf{Q}''(0) = 6(\mathbf{V}_0 - 2\mathbf{V}_1 + \mathbf{V}_2)$$
$$= -6[(\mathbf{V}_1 - \mathbf{V}_0) + (\mathbf{V}_1 - \mathbf{V}_2)]$$
$$\mathbf{Q}''(1) = 6(\mathbf{V}_1 - 2\mathbf{V}_2 + \mathbf{V}_3)$$
$$= -6[(\mathbf{V}_2 - \mathbf{V}_3) + (\mathbf{V}_2 - \mathbf{V}_1)]$$

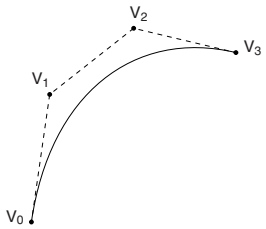


In general, the *n*th derivative at an endpoint depends only on the *n*+1 points nearest that endpoint.

Ensuring C^2 continuity

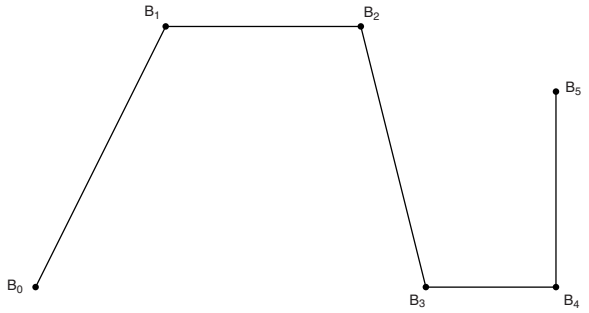
Suppose we have a cubic Bézier defined by ($\mathbf{V}_0$,$\mathbf{V}_1$,$\mathbf{V}_2$,$\mathbf{V}_3$), and we want to attach another curve ($\mathbf{W}_0$,$\mathbf{W}_1$,$\mathbf{W}_2$,$\mathbf{W}_3$) to it, so that there is C^2 continuity at the joint.

$$C^0 : \mathbf{Q}_V(1) = \mathbf{Q}_W(0)$$
$$C^1 : \mathbf{Q}_V'(1) = \mathbf{Q}_W'(0)$$
$$C^2 : \mathbf{Q}_V''(1) = \mathbf{Q}_W''(0)$$



Building a complex spline

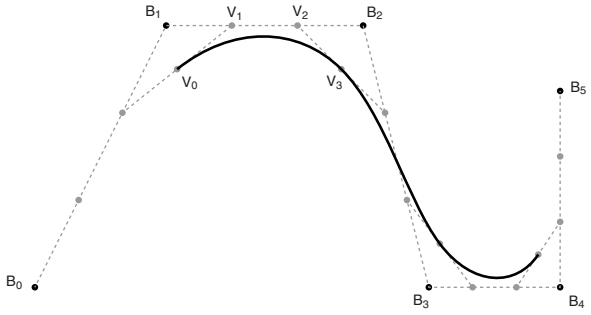
Instead of specifying the Bézier control points themselves, let's specify the corners of the A-frames in order to build a C^2 continuous spline.



These are called **B-splines**. The starting set of points are called **de Boor points**.

B-splines

Here is the completed B-spline.



What are the Bézier control points, in terms of the de Boor points?

$$\mathbf{V}_0 = \_\_\_\_[\_\_\_\_\mathbf{B}_0 + \_\_\_\_\mathbf{B}_1]$$
$$+\_\_\_\_[\_\_\_\_\mathbf{B}_1 + \_\_\_\_\mathbf{B}_2]$$
$$= \_\_\_\_\mathbf{B}_0 + \_\_\_\_\mathbf{B}_1 + \_\_\_\_\mathbf{B}_2$$
$$\mathbf{V}_1 = \_\_\_\_\mathbf{B}_1 + \_\_\_\_\mathbf{B}_2$$
$$\mathbf{V}_2 = \_\_\_\_\mathbf{B}_1 + \_\_\_\_\mathbf{B}_2$$
$$\mathbf{V}_3 = \_\_\_\_\mathbf{B}_1 + \_\_\_\_\mathbf{B}_2 + \_\_\_\_\mathbf{B}_3$$

B-splines

We can write the B-spline to Bezier transformation as:

$$\begin{bmatrix} \mathbf{V}_0^T \\ \mathbf{V}_1^T \\ \mathbf{V}_2^T \\ \mathbf{V}_3^T \end{bmatrix} = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}\begin{bmatrix} \mathbf{B}_0^T \\ \mathbf{B}_1^T \\ \mathbf{B}_2^T \\ \mathbf{B}_3^T \end{bmatrix}$$

$$\mathbf{V} = \mathbf{M}_{\text{B-spline}}\mathbf{B}$$

What is the matrix form for the curve $\mathbf{Q}(u)$?

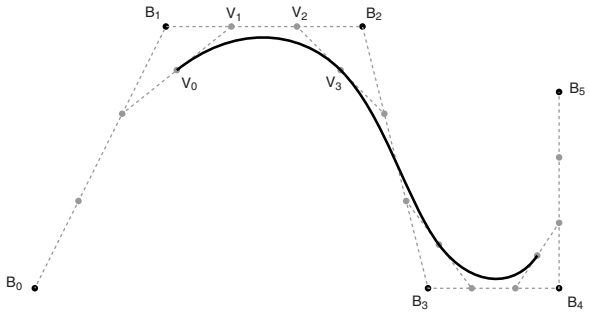
Endpoints of B-splines

We can see that B-splines don't interpolate the de Boor points.

It would be nice if we could at least control the *endpoints* of the splines explicitly.

There's a trick to make the spline begin and end at control points by repeating them.

In the example below, let's force interpolation of the last endpoint:



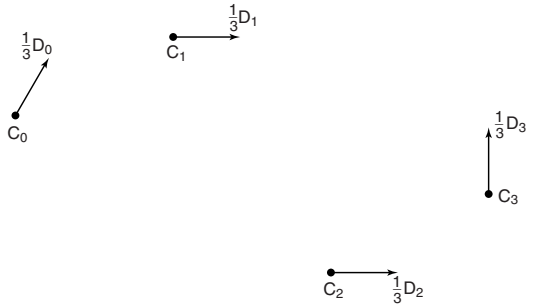
$\mathbf{B}_0\ \mathbf{B}_1\ \mathbf{B}_2\ \mathbf{B}_3\ \mathbf{B}_4\ \mathbf{B}_5$

C^2 interpolating splines

Interpolation is a really handy property to have.

How can we keep the C^2 continuity we get with B-splines but get interpolation, too?

Here's the idea behind ***C*² interpolating splines**. Suppose we had cubic Béziers connecting our control points $\mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_m$ and that we somehow knew the first derivative of the spline at each point.



What are the **V** and **W** control points in terms of **C**s and **D**s?

Finding the derivatives

Now what we need to do is solve for the derivatives. To do this we'll use the C^2 continuity requirement.

$$\begin{aligned}
\mathbf{V}_0 &= \mathbf{C}_0 & \mathbf{W}_0 &= \mathbf{C}_1 \\
\mathbf{V}_1 &= \mathbf{C}_0 + \tfrac{1}{3}\mathbf{D}_0 & \mathbf{W}_1 &= \mathbf{C}_1 + \tfrac{1}{3}\mathbf{D}_1 \\
\mathbf{V}_2 &= \mathbf{C}_1 - \tfrac{1}{3}\mathbf{D}_1 & \mathbf{W}_2 &= \mathbf{C}_2 - \tfrac{1}{3}\mathbf{D}_2 \\
\mathbf{V}_3 &= \mathbf{C}_1 & \mathbf{W}_3 &= \mathbf{C}_2
\end{aligned}$$

C^1 continuty:

$\mathbf{V}_2$ and $\mathbf{W}_1$ both depend on $\mathbf{D}_1$

C^2 continuty:

$$\mathbf{Q}''(1) = \mathbf{P}''(0)$$

$$6(\mathbf{V}_1 - 2\mathbf{V}_2 + \mathbf{V}_3) = 6(\mathbf{W}_0 - 2\mathbf{W}_1 + \mathbf{W}_2)$$

Finding the derivatives, cont.

Here's what we've got so far:

$$\begin{aligned}
\mathbf{D}_0 + 4\mathbf{D}_1 + \mathbf{D}_2 &= 3(\mathbf{C}_2 - \mathbf{C}_0) \\
\mathbf{D}_1 + 4\mathbf{D}_2 + \mathbf{D}_3 &= 3(\mathbf{C}_3 - \mathbf{C}_1) \\
&\vdots \\
\mathbf{D}_{m-2} + 4\mathbf{D}_{m-1} + \mathbf{D}_m &= 3(\mathbf{C}_m - \mathbf{C}_{m-2})
\end{aligned}$$

How many equations is this?

How many unknowns are we solving for?

Not quite done yet

We have two additional degrees of freedom, which we can nail down by imposing more conditions on the curve.

There are various ways to do this. We'll use the variant called **natural C^2 interpolating splines**, which requires the second derivative to be zero at the endpoints.

This condition gives us the two additional equations we need. At the $\mathbf{C}_0$ endpoint, it is:

$$\mathbf{Q}''(0) = 6(\mathbf{V}_0 - 2\mathbf{V}_1 + \mathbf{V}_2) = 0$$

At the $\mathbf{C}_m$ endpoint, it is:

$$\mathbf{R}''(1) = 6(\mathbf{U}_1 - 2\mathbf{U}_2 + \mathbf{U}_3) = 0$$

Solving for the derivatives

Let's collect our m+1 equations into a single linear system:

$$\begin{bmatrix} 2 & 1 & & & & & \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & & \ddots & & & \\ & & & 1 & 4 & 1 & \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} 3(\mathbf{C}_1 - \mathbf{C}_0) \\ 3(\mathbf{C}_2 - \mathbf{C}_0) \\ 3(\mathbf{C}_3 - \mathbf{C}_1) \\ \vdots \\ 3(\mathbf{C}_m - \mathbf{C}_{m-2}) \\ 3(\mathbf{C}_m - \mathbf{C}_{m-1}) \end{bmatrix}$$

It's easier to solve than it looks.

We can use **forward elimination** to zero out everything below the diagonal, then **back substitution** to compute each **D** value.

Note: technically speaking, we need to put the transposes of **D** and **C** vectors in the matrices. We'll omit this for ease of reading.

Forward elimination

First, we eliminate the elements below the diagonal:

$$\begin{bmatrix} 2 & 1 & & & & & \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & & \ddots & & & \\ & & & 1 & 4 & 1 & \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} \mathbf{E}_0 \\ \mathbf{E}_1 \\ \mathbf{E}_2 \\ \vdots \\ \mathbf{E}_{m-1} \\ \mathbf{E}_m \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 & & & & & \\ 0 & 7/2 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & & \ddots & & & \\ & & & 1 & 4 & 1 & \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} \mathbf{F}_0 = \mathbf{E}_0 \\ \mathbf{F}_1 = \mathbf{E}_1 - (1/2)\mathbf{E}_0 \\ \mathbf{E}_2 \\ \vdots \\ \mathbf{E}_{m-1} \\ \mathbf{E}_m \end{bmatrix}$$

Back subsitution

The resulting matrix is **upper diagonal**:

$$\mathbf{UD} = \mathbf{F}$$

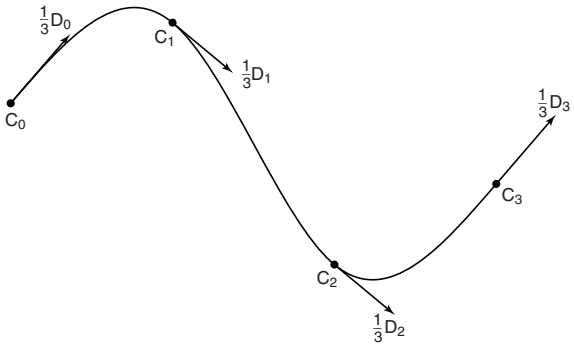
$$\begin{bmatrix} u_{11} & & \cdots & & u_{1m} \\ & & & & \\ & & & & \\ & & \ddots & & \vdots \\ & & & & \\ & & & & u_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} \mathbf{F}_0 \\ \mathbf{F}_1 \\ \mathbf{F}_2 \\ \vdots \\ \mathbf{F}_{m-1} \\ \mathbf{F}_m \end{bmatrix}$$

We can now solve for the unknowns by back substitution:

$$u_{mm}\mathbf{D}_m = \mathbf{F}_m$$
$$u_{m-1m-1}\mathbf{D}_{m-1} + u_{m-1m}\mathbf{D}_m = \mathbf{F}_{m-1}$$

C^2 interpolating spline

Once we've solved for the real D_is, we can plug them in to find our Bézier control points and draw the final spline:



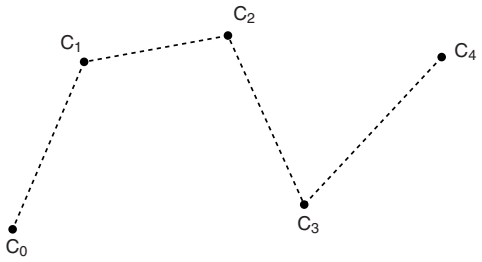
Have we lost anything?

A third option

If we're willing to sacrifice C^2 continuity, we can get interpolation *and* local control.

Instead of finding the derivatives by solving a system of continuity equations, we'll just pick something arbitrary but local.

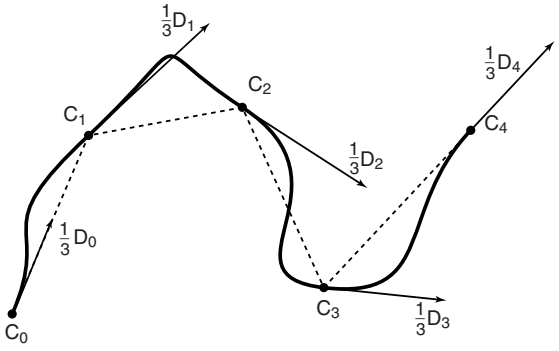
If we set each derivative to be a constant multiple of the vector between the previous and next controls, we get a **Catmull-Rom spline**.



Catmull-Rom splines

The math for Catmull-Rom splines is pretty simple:

$$\mathbf{V}_0 = \mathbf{C}_1$$
$$\mathbf{V}_1 = \mathbf{C}_1 + \tfrac{\tau}{3}(\mathbf{C}_2 - \mathbf{C}_0)$$
$$\mathbf{V}_2 = \mathbf{C}_2 - \tfrac{\tau}{3}(\mathbf{C}_3 - \mathbf{C}_1)$$
$$\mathbf{V}_3 = \mathbf{C}_2$$

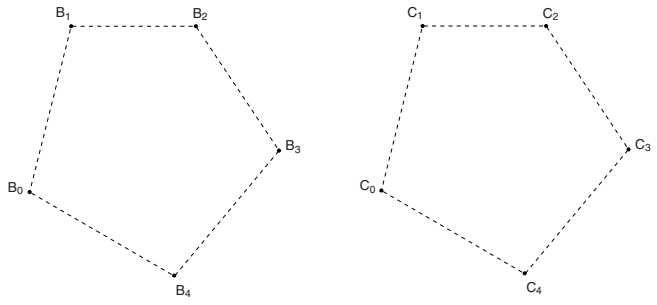


The parameter τ controls the tension.

Closing the loop

What if we want a closed curve, i.e., a loop?

With B-splines and Catmull-Rom curves, this is easy:



Closing the loop, cont'd

With C^2 interpolating splines, we have to modify the matrix:

$$\begin{bmatrix} 4 & 1 & & & & 1 \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & & \ddots & & \\ & & & 1 & 4 & 1 \\ 1 & & & & 1 & 4 \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} 3(\mathbf{C}_1 - \mathbf{C}_m) \\ 3(\mathbf{C}_2 - \mathbf{C}_0) \\ 3(\mathbf{C}_3 - \mathbf{C}_1) \\ \vdots \\ 3(\mathbf{C}_m - \mathbf{C}_{m-2}) \\ 3(\mathbf{C}_0 - \mathbf{C}_{m-1}) \end{bmatrix}$$

We can use a *modified* forward elimination to zero out everything below the diagonal, then back substitution to compute each *D* value.

Curves in the animator project

In the animator project, you will draw a curve on the screen:

$$\mathbf{Q}(u) = \big(x(u), y(u)\big)$$

You will actually treat this curve as:

$$\theta(u) = y(u)$$
$$t(u) = x(u)$$

Where θ is a variable you want to animate. We can think of the result as a function:

$$\theta(t)$$

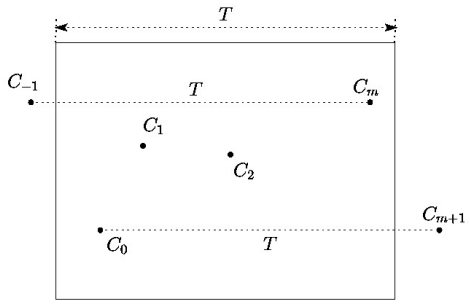
You have to apply some constraints to make sure that $\theta(t)$ actually is a *function*.

"Wrapping"

One of the requirements is to implement "wrapping" so that the animation restarts smoothly when looping back to the beginning.

This is a lot like making a closed curve: the calculations for the θ-coordinate are exactly the same.

The *t*-coordinate is a little trickier: you need to create "phantom" *t*-coordinates before and after the first and last coordinates.



Summary

What to take home from this lecture:

- Geometric and algebraic definitions of Bézier curves.
- Basic properties of Bézier curves.
- How to display Bézier curves with line segments.
- Meanings of C^k continuities.
- Geometric conditions for continuity of cubic splines.
- Properties of C^2 interpolating splines, B-splines, and Catmull-Rom splines.
- Geometric and algebraic construction of B-splines and Catmull-Rom splines.
- How to construct closed loop splines.