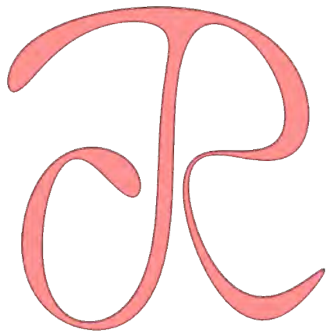
Curves

Jarek Rossignac



Curves are the building block of shape modeling, graphics, and animation. Artists, animators, architects, scientists, engineers, physicians, and industrial designers often draw (by hand) or create (using an interactive design software) a curve that represents the boundary of a regions or the trajectory of a point. These curves may be planar (2D) or not (3D or higher). Although they may contain sharp inflections at desired locations, they are usually smooth elsewhere. Combinations of such curves may be used to define surfaces, solids, and their animations.

As a potential user and developer of shape and animation design and visualization tools, you need to be familiar with the formulations, properties, limitations, and control techniques for such curves. Hence, in this chapter, we discuss a variety of curve representations and their use to create surfaces and animations.

We start by distinguish between implicit, parametric, and procedural representations of curves. Each representation has advantages and drawbacks. You need to understand and remember these, since they may impact the difficulty of developing—or the cost of performing—various functionalities that applications may require. For example, some representations may increase the running cost and reduce the accuracy of algorithms that compute intersections between a curve and a surface. Intersection computation is important for design (computing the vertex at the intersection of a curved edge with a surface) and for animation (computing the place where the trajectory of a point collides with a target surface). Some curves have all three representations. Others do not. Some representations of some curves may exist, but be impractical due to their complexity or numeric instabilities. Hence, we focus primarily on curves that have simple and practical representations of one or more kinds.

In particular, we discuss two popular schemes: Bezier and B-spline curves, which have simple parametric and procedural representations and different degree of smoothness. Both are defined by an ordered set of control points, which together are referred to as “the control polygon”.

Because Bezier curves, B-spline curves, and curves of other types may be conveniently defined and edited using control polygons, we investigate a variety of approaches for analyzing and processing polygonal curves, either open strokes or closed loops. In particular, we investigate several important questions, such as how to estimate the tangent and curvature at a vertex, how to test whether a point lies in the region defined by a polygonal loop, and how to trim out undesired portions of a self-intersecting polygonal loop.

Then, we discuss smoothing, re-sampling, and subdivision schemes for polygonal curves, which attempt to increase their smoothness, sometimes converging (in the limit) to known parametric curves. In particular, we study the new J-spline curves, which include the popular cubic and the quintic B-splines, and offer many interesting options for trading higher-order continuity for increase fidelity with respect to the control polygon. Finally, we study new ringing techniques for evaluating J-spline curves and the surfaces and animations derived from them.

You need to be able to remember, practice, and teach all concepts and techniques presented in this chapter. Many techniques discussed here are illustration of generic principles and approaches that you will be using in other circumstances, hence it is essential that you understand, remember, and practice them. Read the chapter slowly and make sure that you understand all the derivations and all the remarks. After each section, hide the text and try to write a short summary of what you have learned. Check the section and correct your summary.

Important formulae are shown in red. Programming assignments for both graduate and undergraduate students are highlighted in cyan. These must be posted as online applets. Additional theoretical assignments for graduate students are highlighted in grey. These must be posted as PDF files. Both are due before the fourth lecture. Please email the URLs with your full name to the TA.

1 - Functions

Before diving into curves, let us review functions. We do so, because several parametric and procedural schemes that define curves are in fact defining the curve in terms of functions that constrain or parameterize the coordinates of the points on the curves.

For a computer scientist, a function is a piece of code that may be invoked to compute a value (of a certain type, numeric or not). The function may take zero, one, or more arguments (parameter values) of different types. Although, complex numbers may be essential in some curve formulations, we focus primarily on functions that return real numbers, points or vectors. Points and vectors are vectors of two or three real numbers, each representing a coordinate (x, y, or z). A real number returned by a function may represent an angle or other geometric measures, such as the length of the projection (dot product) of a vector onto a direction.

Sometimes, the function is given. It may be the result of a clever geometric construction. Other times, the function is not known, and it is your job to find it, given a set of constraints that must either be approximated or interpolated exactly.

Functions may be very complex. Practitioners like to distinguish several types of popular functions, because the type of a function provides important information on the mathematical intricacy, computational cost, and numeric accuracy of its implementation.

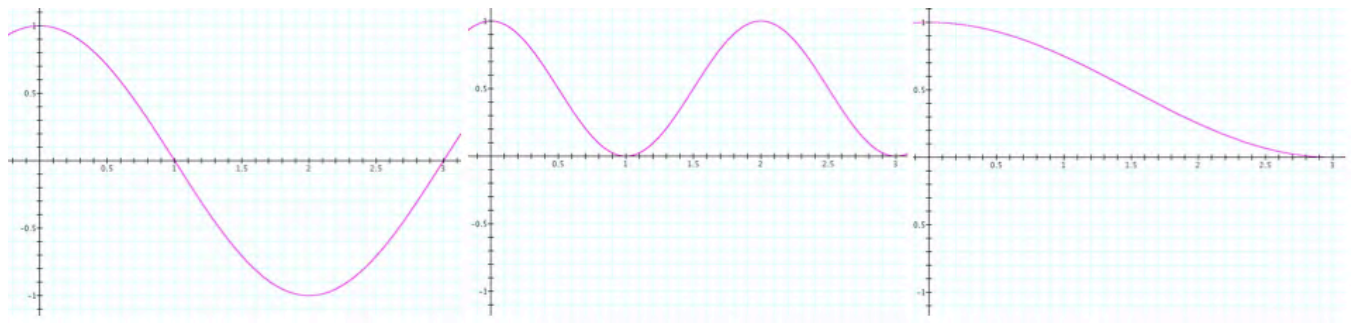
The simples and most popular are the **linear** functions in one or more variables. They are of the form $f(x)=ax+b$ or $f(x,y)=ax+by+c$. Surprisingly, many of the geometric constructions (including Bezier and B-spline curves, surfaces, and animations) are simple compositions of such linear functions. Although the result of such a compositions may not be linear, it may be computed by cascading linear functions. For example, $f(x)=a(x)x+b(x)$, where $a(x)=cx+d$ and $b(x)=ex+f$ is $f(x)=(cx+d)x+ex+f$, which is the quadratic function $f(x)=cx^2+(d+e)x+f$.

The next most popular set of functions are **polynomial** functions, where $f(x)=\sum a_i x^i$ for $i=0,1\ldots$ The coordinates of points on the Bezier curve and on a span of a B-spine curve are each a polynomial function of a parameter. For example the point P(t) on the curve, defined by parameter t, has coordinates x(t) and y(t), where $x(t)=\sum a_i t^i$ and $y(t)=\sum b_i t^i$. It is often important to distinguish the degree (highest power) of the polynomial, since quadratic functions are easier to compute upon than cubic or quartic ones. The vague term "easier" encapsulates issues of the shear cost of evaluating the function, the relative magnitude of the round-off errors that may creep during such a computation, and the difficulty of solving problems that involve such functions (for example, finding their maxima or solving systems of equations defined by several functions.

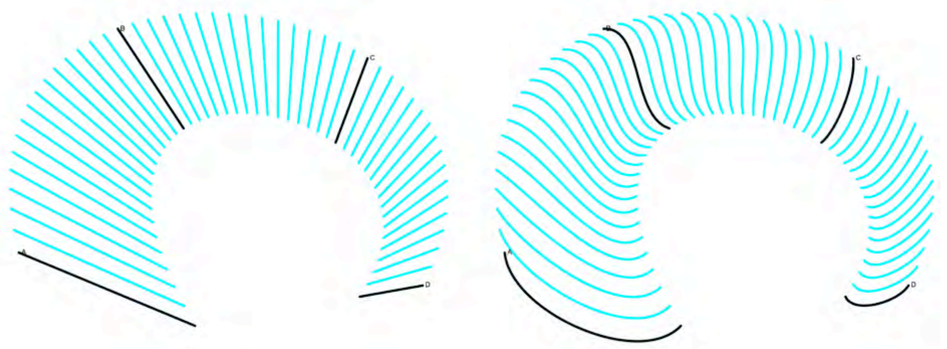
Many geometric constructions are defined as the solution of an equation. When the parameters of the equation vary, the solution evolves as a function of these parameters. More precisely, **algebraic** functions are defined as the solutions y(x) of a polynomial equation $\sum a_i(x)y^i=0$, or more generally, as the function y that solves the polynomial equation in n variables: $p(y,x_1, x_2,\ldots,x_n)=0$. When p is irreducible (i.e., cannot be expressed as the product of two simpler polynomials), then, by the implicit function theorem the solution $y(x_1, x_2,\ldots,x_n)$ exists, even though it may require composing different functions, each one valid over a specific domain. Often, $y(x_1, x_2,\ldots,x_n)$ is given as a polynomial.

Sometimes, the solution of a problem may be formulated as the set of variable combinations that simultaneously satisfy a system of such polynomial equalities (for example one must simultaneously satisfy and equality and its derivative to compute the extrema of a function) and also a set of polynomial inequalities (which constrain the desired solution to lie within a portion of the domain). Functions y that satisfy such systems of polynomial equalities and inequalities are called **semi-algebraic**. They are the building blocks of solid modeling, where 3D shapes are defined in Constructive Solid Geometry (CSG) as unions, intersections, and differences of half-spaces, each defined by a polynomial inequality.

Trigonometric functions are also popular, since they may be used to represent periodic mappings. For example, the parametric curve $x = r \cos(t)$ and $y = r \sin(t)$ is a circle. They are also used for numerous other purposes. For example, a smooth decay function may be defined as follows. Start with $y=\cos(x\pi/2)$ shown left below. Square it, shown center. Then, stretch it by 3 to obtain $y=\cos^2(x\pi/6)$, which decays y smoothly as x varies from 0 to 3, with horizontal slopes at both ends, as shown on the right. It may be used to smoothly decrease the effect of a surface deformation as one moves away from the point to which it was applied, canceling it at a distance of 3.



For example, we start with the image on the left (below). At the endpoint of each black edge (keyframe), the designer imposes a twist (rotation around the endpoint) of a certain angle. The twist is propagated to other points of the black edge, but the twist angle decays with the function described above, where instead of the value 3, we use the length of the edge. Hence, each point of a black edge is the subject of two displacements, each one is a rotation around on of the edge end-points. Each displacement defines a vector from the point to its desired position. We displace the point by the sum of these two vectors. As you move along the edge, the angle of rotation of one of the displacements diminishes as the other one grows. The result will be different from the image shown on the right (which is constructed as explained later in this chapter). The intermediate frames (cyan curves) are obtained by using a Bezier interpolation of the end-point positions and also of their twist angles.



Let (A_0,A_1), (B_0,B_1), (C_0,C_1), and (D_0,D_1) be the endpoint pairs of the control edges. Let (a_0,a_1), (b_0,b_1), (c_0,c_1), and (d_0,d_1) be the corresponding twist angles. An intermediate frame may be identified by the value of a time parameter t in [0,1]. A point along the straight-line edge of that frame that joins its two endpoints may be identified by a position parameter s also in [0,1]. Let P(t,s) define position of that point after the intermediate frame edge has been distorted by the two twists, as explained above. After studying the rest of this chapter, write down the precise expression of P(t,s) in terms of the 8 control points and angles defined above and of the parameters s and t. In your expression of P(s,t), you may use constructions, such as rot(A,f,C), which rotates point A by angle f around a fixed point C, but you need to provide expressions or detailed constructions for each scalar, vector, or point used in your solution.

For extra credit, you may try to **implement** your solution and compare it to the one shown above by changing the applet posted on http://www.gvu.gatech.edu/~jarek/demos/strokeMorphs/

Your solutions for this problem and for the other problems highlighted in grey in this chapter must be combined into a single PDF file and posted online. The file should include the title ("Assignment 1 for CS6491") the date, your full name (including an optional nick name between quotes), and a picture showing your face. The different problems must be clearly stated before your solution. If you found web or printed resources useful, you are free to use them, but you must acknowledge them explicitly, providing a URL and a brief explanation of what you found there and how you used or modified it for the purpose of this assignment.

One may plot a function (x) as the set of points with coordinates (x,y(x)). Such a formulation is both implicit and parametric. It is implicit, because we can say that the (x,y) coordinates of points on the plot satisfy the implicit equation y=y(x). It is parametric, because we can say that the coordinates (x,y) are both functions of a parameter t, where x=t and y=y(t). Finally, when computing y requires solving an algebraic polynomial, the formulation is procedural, as discussed below.

Hence plots of polynomial or algebraic functions are curves. They offer important advantages and are use in many settings. They may be easily extended to represent surfaces. For example, a terrain may be defined by the height h(x,y) of a point with horizontal coordinates (x,y).

Still, such function-based formulations are only a subset of representations of curves and surfaces. For example, a single polynomial function f(x) cannot be used to represent a circle whose implicit equation is $x^2+y^2=r^2$. The circle must be broken into, for example, two arcs: $y=\pm\sqrt{(r^2-x^2)}$.

The more general formulations are discussed next.

2 - Implicit, parametric, and procedural curves

2.1 Implicit formulations

The **implicit** representation of a curve is an equation f(x,y)=0 that constraints the values of the coordinates. Specifically, only points with coordinates (x,y) that satisfy equation f(x,y)=0 are on the curve. Hence, the equation f(x,y)=0 acts as a **constraint**. Often, f() is a polynomial expression.

For example, f(x,y)=0 may be defined as $ax+by+c=0$, which of course implies a line in the plane. When a and b are not zero, the line passes though points (0,–c/b) and (c/a,0), which may be conveniently located along the axis of the global coordinate system. The vector N=<a,b> is orthogonal to the line. In fact, one may say that the line is the set of points P=(x,y) such that OP•N = –c. You may verify that expressing the dot-product as (x–0)a+(y–0)b yields the implicit equation above. A tangent T to the line may be obtained by rotating N by 90 degrees in one or the other direction. Hence, one such tangent is T = <b,–a>. You may verify that N and T are orthogonal to each other, since T•N=0. Often, one prefers to use unit normal's and tangents, hence dividing them by their norm $\sqrt{(a^2+b^2)}$.

A **circle** of radius r centered at the origin corresponds to the implicit equation $x^2+y^2-r^2=0$. Note that both sides of the equation may be multiplied by a non-zero scalar, hence yielding an equivalent equation (when the radius $r\neq0$): $x^2/r^2+y^2/r^2-1=0$. One may generalize this to $x^2/a^2+y^2/b^2-1=0$, which is an **ellipse**.

Another important implicit equation that we will see is the **parabola**: $ax^2-y+b=0$. Which may be written $y=ax^2+b$. Remember that the elegance of this formulation hinges on a judicious choice of coordinate systems.

Implicit curves that can be expressed as a polynomial of degree two ($ax^2+bxy+cy^2+dx+ey+f=0$) are called **conic sections**. The are also called **quadrics**, even though the term is more often used when discussing such functions of 3 variables (x,y,z), see below.

Indeed, implicit equations generalize trivially to surfaces in three dimensions. For example, the sphere of radius r centered at the origin may be defined as the set of points with coordinates (x,y,z) that satisfy the implicit equation $x^2+y^2+z^2-r^2=0$, or equivalently, $x^2+y^2+z^2=r^2$. Implicit surfaces that are defined by a polynomial equation of degree 2 (such as a sphere, a cylinder, a paraboloid) are called **quadrics**. Note that, implicit surfaces in 3D may be represented by a single implicit equation, which may have 3 variables: x, y, and z.

A torus may not be represented by a quadric implicit equation, but may be represented by a fourth order polynomial equation. Such equations are called **quartics**. Hence the torus is a quartic.

Note that a line that has a finite number of intersection points with a polynomial implicit surface of degree n intersects it at n points or less. For example, a line may only stab a plane once (assuming that the line is not in the plane). A line may only stab a quadric twice. A line may only stab a torus four times.

The main **advantage** of the implicit formulation is the simplicity of **testing** whether a given point (x,y) lies on the curve: we simply plug the values of x and y in the implicit equation and test whether it is verified. Another **advantage** is that the implicit formulation defines an **equation**. Hence, intersections of two curves may be formulated as the sets satisfying simultaneously their two implicit equations. Computing intersections then turns into solving (i.e., finding the roots of) a system of two simultaneous equations.

The main **drawback** of the implicit formulation is the **difficulty of generating** an ordered set of **points** along the curve, which is often needed for drawing the curve on modern graphic systems.

2.2 Parametric formulations

The **parametric** formulation of a curve is defined by a function that takes a parameter, t, as input and returns a point P(t). Typically, the coordinates (x,y), in 2D or (x,y,z) in 3D, of P are each a function of t. Again, we distinguish different types of functions, each associated with a different level of evaluation cost and difficulty: linear, quadratic, cubic, polynomial, rational…

To clarify notation, let (P.x,P.y) denote the coordinates of point P. A curve is linear if P.x(t) and P.y(t) are both **linear** functions of parameter t. For example, let P.x(t)=at+b and P.y(t)=ct+d. It may be more convenient to write this as P(t)=A+tT, where point A has coordinates (b,d) and vector T has components <a,c>. If parameter t is representing time, then vector T is the constant velocity of the point P(t) and A is its initial position (at t=0).

A **quadratic** curve may be formulated as $x(t)=at^2+bt+c$ and $y(t)=dt^2+et+f$. Linear functions are subsumed by quadratic functions (set a=d=0). Quadratic functions are useful, but they have **limitations**: In 2D, they do not have **inflection points**. In 3D they are **planar**.

Note that a circle, or even a circular arc, cannot be represented as a parametric curve by a quadratic function.

Similarly, the parametric form of a **cubic** curve is $x(t)=at^3+bt^2+ct+d$ and $y(t)=et^3+ft^2+gt+h$. These are more powerful than quadrics, because they include inflection points and may be stitched together to offer spline curves with second derivative continuity, as discussed below.

A **rational** formulation represents each coordinate as a division of one polynomial by another. For example, the simplest, **linear rational** version is x(t)=(at+b)/(ct+d) and y(t)=(et+f)/(gt+h). It is popular because perspective projections, which are fundamental in 3D graphics produce rational mappings, such as these. The **quadratic rational** is also popular, because may be used to model a **circle** exactly, something that polynomial and linear rational parametric formulations cannot do.

A fundamental **advantage** of parametric formulations is that they provide a simple recipe for generating points along the curve, which is convenient for drawing and animation: iteratively increment the parameter value t and compute the corresponding point P(t) on the curve.

A **drawback** of parametric formulations, is that they do not directly offer an equation or procedure for testing whether a point (a,b) lies on the curve. This task may in fact be rather difficult, because the test involves searching for solutions (i.e., t values) of the simultaneous system x(t)=a and y(t)=b, which may not have closed-form solutions and may require numerically unstable iterative methods when x(t) and y(t) are higher degree functions of t. For example, it may be rather difficult to compute all the intersection points between two curves, when each one is represented in a parametric form as a cubic.

Note that some curves have both implicit and parametric formulations. For example, a line that is not axis-aligned may be written in implicit form as ax+by+c=0 (or in vector notation as OP•N=d) and in parametric form as (x,y)=(0,–c/b) +t(c/a,c/b) (or in vector notation as P(t)=A+tT). Hence they combine the advantages of both formulations.

2.3 Procedural formulations

Many interesting shapes may be produced by an algorithm or through a series of transformations of some simple initial shape. For example, a snowflake may be drawn using a recursive transformation of the edges of a triangle. Some such procedures converge, in the limit, to a smooth curve. Even though we may not be able to generate the limit curve (because we are not able to execute an infinite number of iterations), we may usually approximate it quickly to a sufficient level of accuracy and may be able to talk about important properties of the limit curve. Unfortunately, such limit curves may not have implicit or parametric formulations. Some of them may be expressed as the solution to a differential equation. Others are the solution of an iterative smoothing or subdivision process.

For example, start with a polygonal curve and repeat the following process: **insert a new vertex in the middle of each edge and then take the dual**. (The dual inserts edge midpoints and deletes the old vertices.) The limit curve produced by this iterative process is the **quadratic B-spline**. If instead, the process inserts the edge midpoints and performs **two duals**, it converges to a **cubic B-spline**. Note that in this case, the resulting curves also have convenient (quadratic and cubic) parametric formulations and we will learn how to derive them from the input polygon. Hence, we can show that the iterative process converges, in the limit, to a smooth curve.

Other recursive subdivision scheme may **not** have useful (low degree polynomial or rational) parametric formulations. The popular **four-point** scheme is an example and many curves defined using the **J-spline** formulations derived from it are good examples.

The **advantage** of procedural formulations is their generality and the fact that they produce points along the curve.

The **drawback** is that it may be difficult to test whether a point lines on the limit curve (no equation is available) and that the points along a curve are generated by a procedure which for example doubles them at each step. Hence additional work may be needed when a specific number of points regularly distributed along the curve is desired (as it may be the case for animation, when a point must move along the curve at uniform speed in precisely k frames).

3 - Bezier curves

We are interested in curves that are each a continuous mapping from a real interval to some Euclidean space (typically the 2D plane or the 3D space). Such a mapping associates a point P(t) with each parameter value t. We often refer to t as time, since increasing t continuously will result in the motion of point P(t) along the curve. Often, we restrict our attention to the curve **span** that is the image of the t-interval [0,1]. Although other spans may be trivially re-parameterized to be the image of [0,1], for convenience, we sometimes also look at the image of [0,n]. Sometimes, as it is the common practice in the literature, we will use the term "curve" when referring to a span.

P'(t), P''(t), P''''(t), and P''''(t) respectively denote the first, second, third, and fourth **derivatives** of P(t) with respect to t.

We investigate here the construction of spans that either interpolate or approximate a control polygon (a series of control points). We discuss linear, quadratic, and cubic spans and then their generalization to higher degree curves. We also investigate procedural subdivision schemes that iteratively refine the control polygon, converging to the desired span. Finally, we discuss their use for constructing surfaces and animations.

Because we will discuss parametric curves and their derivatives with respect to parameterization, we often talk about them as motions and use terms such as velocity and acceleration. Note however that often, the designer is only interested in the plot of the curve, not in its parameterization. In such situations, derivatives in terms of an arc-length parameterization may be more appropriate.

3.1 Linear Bezier curves

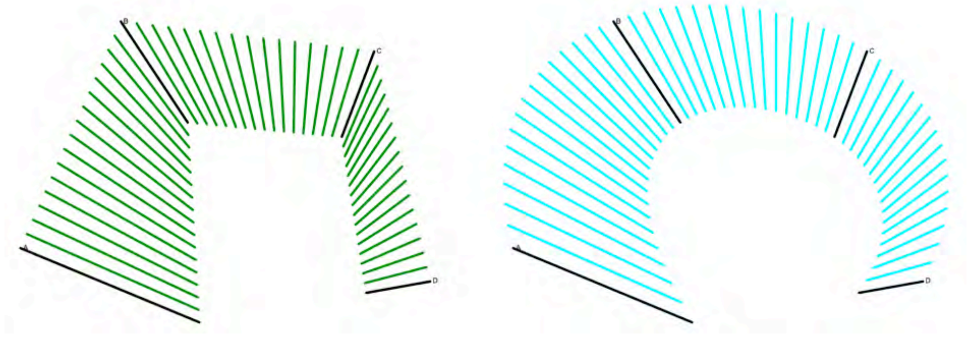
The simplest motion is linear. Assume that we want a constant velocity motion. The constant velocity constraint yields: P'(t)=V. Standard integration yields: $P(t)=tV+P_0$, which defines the motion or curve by some starting point, P_0, and by its constant velocity, V. We say that $tV+P_0$ is the **parametric** representation of the curve swept by P(t) as t varies in the prescribed interval. As stated earlier, this parametric formulation is useful for instantiating points along the curve (rendering) and for moving a point or coordinate system (animation).

If we let the parameter t vary from –∞ to +∞, the swept region is a line. An **implicit** representation of the line is P_0P•N=0. Here the symbol • denotes the dot product and N is the vector normal to the line, i.e. orthogonal to vector P_0P. Hence P_0P•N=0. (Remember that U•V=||U|| ||V|| cos(angle(U,V), which vanishes when one of the vectors is null or when their angle is ±90 degree.) We usually **normalize** N by dividing it by its length so that it has unit length. We call this normalized vector the **normal** of the line.

The implicit formulation is useful for computing collisions and intersections, because it provides a constraint that must be met by all points P on the line that leads to an equation used to solve for collision.

Instead of specifying the curve (or linear motion) in terms of the starting point P_0 and the constant velocity vector V, we may need the motion to start at point A and finish at point B. Hence, we must solve the problem of computing the parametric formulation of our curve subject to two **interpolatory position constraints**: P(0)=A and P(1)=B. Note here that we have decided to start the clock (t=0) when the pint is at A and to choose the time unit so that it arrives at B when t=1. As said earlier, other choices of origin of time and unit may be trivially converted to this one.

Substituting these constraints in $P(t)=tV+P_0$ yields: $A=P_0$ and B=A+V. Hence, of course, the velocity over this time trip is the constant V=B–A=AB. It is constant, because, by definition, we have decided to compute a linear interpolation. Note that such linear (constant velocity) motions are avoided by animation artists, who prefer gentle **arcs** and more interesting kinematic behaviors, offered by higher degree interpolations, as discussed later. For example, compare the linear interpolation of 4 edges (left below) with a curved interpolation (right). On the left, we used a linear interpolation between the corresponding endpoints of consecutive edge pairs. On the right, we consider the top endpoint of each edge and **fit a cubic Bezier curve** through them, as explained below. We do this for the bottom endpoints as well. Then, we join the corresponding points on these two parametric curves by cyan edges.



Since several of the constructions discussed below for higher degree curves are based on such linear interpolations, we will use a **function** L that computes P(t) given two constraint points A and B and a value for the t parameter:
`pt L(A,t,B) {return A+tAB ;}`

Note that we will often write A+tAB as (1–t)A+tB. Although this notation makes no sense when thinking of points as locations (you cannot scale or add locations), it is standard in linear algebra and makes sense if one associates point A with vector OA from some agreed upon origin O and then overloads the meaning of point addition and scaling with performing such operation on the associated vectors. Indeed, one can easily verify that (1–t)A+tB=A–tA+tB=A+t(B–A)=A+tAB. This notation, (1–t)A+tB, defines P(t) as a **weighted sum** of points A and B. Furthermore, because the two coefficients (weights) add to 1, the sum is a **convex combination** and the result, P(t), is **independent of the choice of the origin O** (an important property to preserve in your future constructions and formulations).

3.2 Quadratic Bezier spans

As said above, linear motions may not need the aesthetic or functional requirements of an application. Here we look at quadratic curves and motions.

3.2.1 Motivation and definition

In some applications, aesthetic or functional concerns call for curves where the first derivative, P'(t) is continuous everywhere (except at the first and last control point of a stroke). One way of building such curves is to paste together a series of quadratic spans. Each one, may be defined by a quadratic function, where the third derivative P'''(t) is zero, implying that the second derivative (acceleration) is constant: P"(t)=G.

3.2.2 Integration and parabola

To produce a parametric formulation of a quadratic span, we start with P"(t)=G and integrate it to obtain P'(t)=tG+V, which indicates that the **velocity vector varies linearly with time**. V is the **initial velocity** (at t=0). Integrating again yields: $P(t)=t^2G/2+tV+P_0$, where P_0 is some initial position.

You may recognize the equation of a **parabola**, which for example models free fall motion without friction.

3.2.3 Degrees of freedom and design paradigms

As t varies in [0,1], P(t) traces a parabolic span. Extending t past this unit interval traces the remaining branches of the parabola. To define the span, we must compute G, V, and P_0. In two dimensions, the triplet of two vectors, G and V, and the point P_0, is defined by 6 variables (their 3 coordinate pairs). Hence, we say that this parametric definition of a parabolic span has 6 **degrees of freedom** (DoF).

This statement implies that a designer will have to specify 6 values in order to completely define a parabolic span. For example, if one mouse click specifies two parameters (x and y coordinates of the mouse with respect to a screen coordinate system), them 3 mouse clicks will be necessary to completely define a parabolic span. Of course, you may invent the semantics and the conversion algorithm that defines how the three mouse positions are to be interpreted to define the parabola. For example, in some military or shooting game application, the user may find it natural to specify three points, P_0, V_0, and G_0, from which you would compute $V= V_0–P_0$ and $G= G_0–P_0$. Below, we discuss two other design paradigms, which are more symmetric and possibly more intuitive for design and fitting.

3.2.4 Solving for an interpolating parabolic span

One way to specify the degrees of freedom of a parabola is to force the parabola to pass through 3 given points, A, M, and C, at chosen time values. In other words, we specify the parabola by providing three position-interpolating constraints, for example: P(0)=A, P(½)=M, P(1)=C. Hence, evaluating the quadratic expression for the given values of t, we obtain a system of three point-equations: P_0=A, G/8+V/2+A=M, and G/2+V+A=C. In 2D, each one corresponds to two equations, one per coordinate.

The system may be solved using standard linear algebra techniques, or plain substitution as follows. The first equation gives us P_0 directly. We can subtract A from both sides of the other two equations to obtain a linear system of two vector equations: G/8+V/2=AM and G/2+V=AC. Multiplying the first one by 4 and subtracting the second one yields V=4AM–AC and hence V=3AM+CM. Substituting V in the second equation yields G/2+3AM+CM=AC, which becomes G/2=AC+3MA+MC, G/2=AC+MA+2MA+MC, and finally G=4(MA+MC).

3.2.5 Program the drawing of the span and of vectors V and G

To memorize these formulae and develop an intuitive understanding of their meaning, write a short applet that lets the designer move the three points A, M and C. Draw the points and their names. Draw the initial velocity V as a vector (arrow) starting at A. Then, draw the constant acceleration G as a vector (arrow) starting at M. You may want to start with the applet posted at http://www.gvu.gatech.edu/~jarek/examples/applet06/ and modify it to suit your needs.

Then write a program that plots the parabola using: $P(t)=t^2G/2+tV+P_0$, where P_0=A, V=3AM+CM, and G=4(MA+MC). Do this first by incrementing t and by evaluating $t^2G/2+tV+P_0$.

Then change colors and use the following incremental tracing approach, which is based on the Tayor series expansion of this polynomial form. To trace a parabola, assume that you know the current position P(t) and current velocity V(t). (You do know them at t=0.) You may start with the pseudo-code below (you should test it carefully and correct mistakes, if any):

```
pt P=A; vec V=2AB; vec G=2(BA+BC); // start, velocity, and acceleration
D=eV+e2G/2; // position increment
W=eG; // velocity increment
beginShape(); for (s=0; s<=1; s+=e) { P.vertex();  P+=D; V+=W; }; endShape();
```

This code must of course be converted into function or method calls that manipulate points and vectors. Again, let the user select and drag points A, B, and C. Draw the hat and the parabola. Compare your drawing with the curve provided by the graphic API.

3.2.6 More general time constraints

Note that we picked special constraints. The time values associated with the points A, M and C were evenly spaced and spanning the [0,1] interval. Other choices of evenly spaced time values may be trivially converted to this formulation by a linear mapping between a different parameter and time t. Say for example that you want Q(a)=A, Q((a+b)/2)=M, Q(b)=C. We compute the parabola P(t) for P(0)=A, P(½)=M, P(1)=C, and use the mapping Q(s)=P((s–a)/(b–a)).

However, when the constraint parameters are not evenly spaced, we can still compute the parabola parameters by solving the system of two vector equations, but the system will depend on the chosen time values and hence will yield a different solution from the one above. Such a more general solution may be preferred in some applications, for example when one of the distances ||AM|| and ||MC|| is much larger than the other or to achieve interesting animation effects.

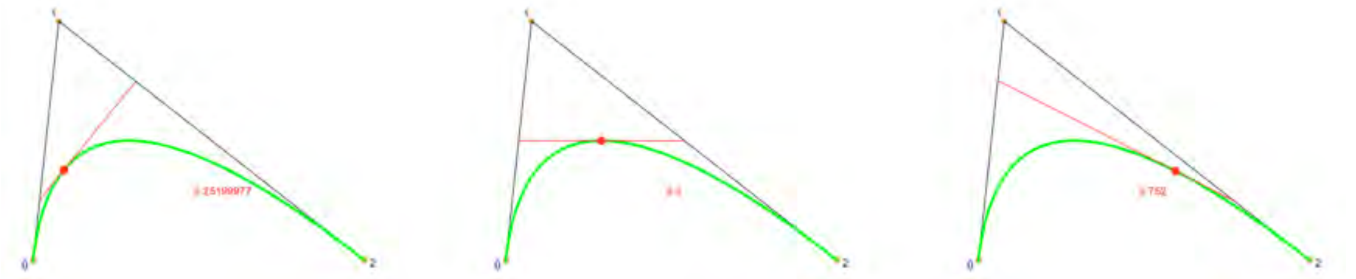
3.2.7 Control hat of a quadratic Bezier span

The parabolic span constructed above interpolates the three constraint points, A, M, and C. Now, we will consider another way of defining a parabola using a slightly different triplet of control points, A, B, and C. Here, the parabola interpolates A and C as before using P(0)=A and P(1)=C. However, it does not interpolate B. The desired parabola is contained in the convex hull of points A, B, and C, i.e. in the triangle (A,B,C), which I call the **hat**. Furthermore, at t=0, the parabola is tangent to the edge AB and at t=1 it is tangent to the edge BC. Hence, the parabolic span snuggly fits its hat. This alternate definition is important for several reasons: (1) It is the popular **quadratic Bezier curve** of the control polygon (A, B, C). (2) It may be evaluated using a simple combination of the linear interpolations using L() defined above. (3) It provides an intuitive tool for rounding the sharp corners of polygons.

3.2.8 Evaluation of the quadratic Bezier span from its control hat

A point P(t) of the parabolic span that is defined by the control polygon or hat (A,B,C) may be computed using a point-valued function: pt Q(A,B,C,t) {return L(L(A,t,B) ,t, L(B,t,C)) }, which performs a **synchronized linear interpolation of linear interpolations**. Remember this principle, as it will help you invent constructions for higher-order (smoother) spans and splines and also achieve smooth morphs between shapes.

To keep a mental image, think of one train T going at constant speed from A to B. Another train U is simultaneously going from B to C. An elastic rope is stretched between T and U. You are simultaneously walking along that rope so that at each time t, the length in front of you and the length behind you have the ratio of (1–t) to t. You are walking at constant speed, but in the local coordinates of the moving and stretching rope. The construction is shown for different values of t (roughly ¼, ½, and ¾) in the figure below.



3.2.9 Computing V and G from a control hat

Developing and factoring the formulation P(t)=L(L(A,t,B) ,t, L(B,t,C)) yields $P(t)=(1–t)^2A+2t(1–t)B+t^2C$ and thus $P(t)=(BA+BC)t^2+(2AB)t+A$. Substituting various values of t, we can verify that the span interpolates A and C: P(0)=A, P(1)=C. Furthermore, by identification, we conclude that the initial velocity at A is V=2AB and the constant acceleration is G=2(BA+BC).

3.2.10 Computing the mid-course point M from a control hat

We can compute the mid-course point P(½)=((A+B)/2+(B+C)/2)/2=B+(BA+BC)/4. Note that this is the mid-course interpolated point M used above to define the parabola through 3 interpolated point constraints. Hence M=P(½)=B+(BA+BC)/4. In other words, one may construct M as the midpoint of an edge between the midpoints of segments (A,B) and (A,C).

3.2.11 Retrofitting the hat from A, M, C

We can use this observation to compute the hat of a parabola defined by 3 interpolated points (A,M,C). This process of computing B from A, M, and C is called **retrofitting**. M=B+(BA+BC)/4 yields 4MB=AB+CB, 4MB=AM+MB+CM+MB, MB=(AM+CM)/2. Hence B=M+(AM+CM)/2.

To understand and memorize this formula, use it first by hand to construct B from A, M, and C. Write (and post on your submission web page) an English explanation of how B can be constructed. Then, implement it in your program, so that the points A, M, and B controlled by the user serve to compute B and then the hat (A,B,C) is used by your code to display the parabola (in a different color) to ensure that it is still the same curve. To show that several curves match, the best is to render the first one using a thick stroke and then, for subsequent ones, reduce the stroke weight and change colors.

3.2.12 Matrix formulation

Matrix multiplications provide an elegant formulation for Bezier curves. From $P(t)=(BA+BC)t^2+(2AB)t+A$, we can write $P(t)=(A–2B+C)t^2+(2B–2A)t+A$, which may be written in matrix form as

$$P(t)=\begin{pmatrix}A & B & C\end{pmatrix}\begin{pmatrix}1 & -2 & 1\\ -2 & 2 & 0\\ 1 & 0 & 0\end{pmatrix}\begin{pmatrix}t^2\\ t\\ 1\end{pmatrix}$$

You do not need to know this matrix by heart, but need to be able to compute it from the formulation of P(t).

3.2.13 Formulation of derivative

Taking the derivatives of $P(t)=(BA+BC)t^2+(2AB)t+A$, one obtains $P'(t)=2(t-1)A+2(1-2t)B+2tC$ and $P''(t)=G=2(BA+BC)$. One may verify that the derivative of the matrix formulation yields the same result:

$$P'(t)=\begin{pmatrix}A & B & C\end{pmatrix}\begin{pmatrix}1 & -2 & 1\\ -2 & 2 & 0\\ 1 & 0 & 0\end{pmatrix}\begin{pmatrix}2t\\ 1\\ 0\end{pmatrix}$$

For example: $P'(0)=2AB$, $P'(½)=AC$, $P'(1)=2BC$.

Note (and remember) that the **initial velocity is twice the vector AB** and that the **mid-course velocity is the vector AC**, which is the average velocity over the time interval [0,1].

3.2.14 Convex hull and properties

The **convex hull** is an important concept in graphics, modeling, and animation. We illustrate its use here. The convex hull $H(P_i)$ of a set of points P_i is the set of points that may be written as a convex combination of P_i, which is a weighted sum $\Sigma\alpha_i P_i$, where $\Sigma\alpha_i=1$ and $\forall i$, $0\le\alpha_i$. Intuitively, the bounding loop of the convex hull is the rubber band stretched around the points.

Note that the convex hull of a set contains the set. A set is **convex** if it contains its convex hull. Convex sets have an important property for graphics. They are used in numerous solutions for collision and visibility. For example, if a set S is included in a convex set C, then the convex hull H(S) is also included in C.

In particular, the convex hull, H(A,B,C), of 3 points A, B, C is the set of points aA+bB+cC such that a+b+c=1 with $0\le a$, $0\le b$, $0\le c$. In other words, it is the triangle(A,B,C).

Explain how you would test whether a point P is contained in triangle (A,B,C).

3.3 The span is contained in the convex hull of its control points

The quadratic Bezier span is included in the triangle formed by its control points.

An important property of the quadratic Bezier span is that $P(t) \in H(A,B,C)$ when $0\le t\le 1$. To prove this formula, we note that P(t)=L(L(A,t,B) ,t, L(B,t,C)) computes P(t) by a series of convex combinations of points in H(A,B,C). Indeed the initial points, A, B, and C, are in H(A,B,C). Furthermore, L(Q,t,R) for $0\le t\le 1$ produces points in the convex hull H(Q,R). The property stated above ensures that all constructions stay in H(A,B,C).

More generally, the Bezier span of any degree is included in the convex hull of its control polygons.

How would you use this to quickly test whether a span is guaranteed not to intersect the screen?

3.3.1 Intersection of the span with a line

Intersections between geometric primitives and lines are fundamental to computer graphics, since photons travel in straight lines (rays) and we must often be able to compute where they first hit the primitives. We illustrate such a computation here by computing the intersection between a line and a parabolic span P(t) defined by hat(A,B,C).

We assume that the line Line(Q,N) is passing through Q and is orthogonal to the vector N, which is called the normal of the line N. We identify the intersection points P(t) by the corresponding time values computed as the roots of equation $QP(t)\bullet N=0$, which states that either Q=P(t) or that the vector QP(t) is orthogonal to the normal N, and hence parallel to the line.

Hence, we must find values of t in [0,1] for which $((BA+BC)t^2+(2AB)t+QA)\bullet N=0$. **Distributing the dot product over vector addition** and scaling yields: $((BA+BC)\bullet N)\ t^2 + (2AB)\bullet N\ t + QA\bullet N=0$, which is a quadratic equation in t. You know how to solve it. But do not forget to test whether the roots lie in [0,1]! You must remember this derivation process and be able to perform it to re-derive the formula.

Implement this solution. And verify it by having your program draw the hat, the parabola, the line, and the intersection point(s). You may use a fixed oblique line, but it would be more fun to let the user move the line by dragging two points that define it. You may increase the number of vertices in the polygon P[], and use some of them as control points for a curve and some of them as control points for the line segment. This way, you need not modify the user interface for moving points.

3.3.2 Recursive subdivision

We have seen above how to produce points along the quadratic Bezier span. Such points may be used to compute a polygonal approximation of the parabola. In turn, this polygonal approximation may be used for estimating intersections of the parabola with another parabola. Note however that these polygon intersections are incorrect and may miss true intersections or report false positives.

A safer way of producing an approximation is to use recursive subdivision. Instead of generating a parabola P from its hat we generate two smaller hats, each one defining a parabola P_L and P_R, such that P_L is the first portion of P (for t in [0, ½]), and that P_R is the remaining portion. A recursive subdivision of P_L and P_R yields a series of hats that are a polygonal approximation of P.

This process has a **drawback** over the parametric sampling discussed earlier: it roughly doubles the number of vertices at each level of recursion, hence it is not suited for generating an arbitrary number of evenly spaced points along the span.

An **adaptive subdivision** may be devised to control this growth: we only split a hat if it is sufficiently large and not flat. Various formulae may be used for such tests. One of them computes the area of triangle (A,B,C) and stops subdividing when that area becomes smaller than a given threshold. You may want to think of other criteria, which would for example guarantee that the approximation is within one pixel from the true curve.

Here is the pseudo code for a recursive formulation. Study the construction of the intermediate points. Verify it by hand on an example figure. Convert this pseudo-code to Processing, implement it, and test it (fix what needs fixing). Let the user adjust the number of subdivisions (pressing ‘,’ and ‘.’ keys).

```
void V(A,B,C,r) {if (r==0) {B.vertex(); return;}
        ML=(A+B)/2; MR=(B+C)/2; M=(ML+MR)/2; V(A,ML,M,r-1); V(M,MR,C,r-1); }
```

Suggest how the recursive subdivision, combined with the convex hull property, could be used to compute intersections between pairs of parabolic spans, each defined by its hat. Write down the high level algorithm and explain clearly what each module does. Identify functions that you do not know how to implement. You do not need to implement this part.

3.3.3 Practice exercise

Problem: Given the constraints P(0)=A and P(1)=C compute the equation of a parabolic trajectory with constant acceleration P”(t)=G. Then, compute P’(0), P’(1), P’(½), and P(½).

Solution (*with detailed explanations of each step*):

Successive integration of P”(t)=G yields $P'(t)=tG+V_0$, and then $P(t)= ½t^2G+tV_0+P_0$.

Substituting 0 for t in this expression and using the constraint P(0)=A yields $P_0=A$.

Substituting 1 for t in this expression and using the constraint P(1)=C yields $½G+V_0+P_0=C$.

Substituting A for P_0 in the second equation yields $½G+V_0+A=C$, which means that moving by $½G+V_0$ from A you arrive at C. Hence, the vector $½G+V_0$ is the displacement from A to C. Therefore, $½G+V_0+A=C$ may be written $½G+V_0=AC$. (Avoid writing C–A, which is the difference between two locations and makes no sense.) Hence, by subtracting ½G on both sides, we obtain: $V_0=AC–½G$.

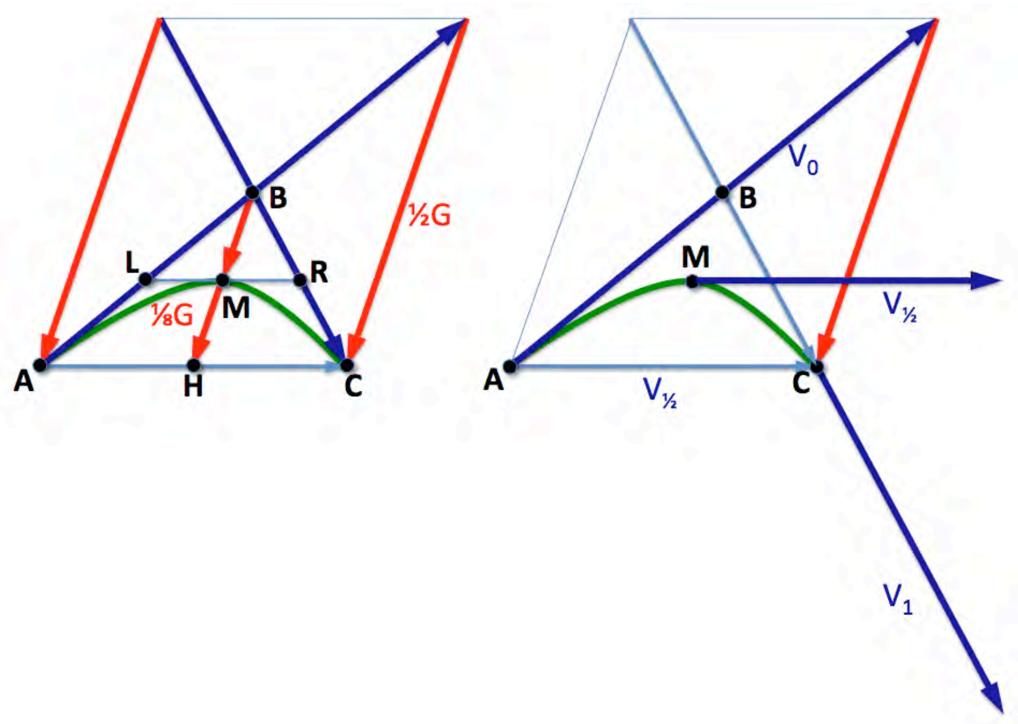
Substituting AC–½G for V_0 and 0 or 1 for t in $P'(t)=tG+V_0$, we obtain P’(0)=AC–½G and P’(1)=AC+½G.

Substituting A for P_0, AC–½G for V_0 and ½ for t in $P(t)= ½t^2G+tV_0+P_0$ yields P(½)=⅛G+½(AC–½G)+A, which simplifies to P(½)=A+½AC–⅛G. Note that A+½AC is the midpoint of segment (A,C) and may be written ½(A+C). Hence , we prefer the more symmetric formula: P(½)=½(A+C)–⅛G.

Substituting AC–½G for V_0 and ½ for t in $P'(t)=tG+V_0$, yields P’(½)=½G+(AC–½G), which simplifies to P’(½)=AC.

3.3.4 Summary of parabolic span

The figure below summarizes the geometric relations between the various vectors and the hat. You should be able to draw it from memory. Start with A and C as the base of the parallelogram. Then, draw the sides displacing A and C by –½G. Then, draw the diagonals. They give you the initial and final velocities, V_0 and V_1. B is the intersection of the diagonals. H is the midpoint of (A,C). M is the midpoint of (B,H).



We can represent the parabola using various combinations of points and vectors mentioned in this figure. Furthermore, we can easily convert from one representation to another. Instead of discussing all such conversions, we will consider the hat as the primary representation and discuss conversions to and from other representations.

Given the hat (A,B,C) we obtain $H=½(A+C)$, $M=½(B+H)$, $G=4BH$, $V_0=2AB$, $V_1=2BC$, and $V_{½}=AC$.

We also obtain the parametric expression $P(t)=A+2tAB+t^2(BA+BC)$ and its alternate form as a composition of linear interpolants expressed as convex combination of control points: $P(t)=(1–t)((1–t)A+tB)+t((1–t)B+tC)$. Note that computing vectors from B to points expressed on both sides of this equality yields, $BP(t)=(1–t)^2BA+t^2BC$.

Also note that the same green parabolic span may be represented as the union of two half-spans, one defined by hat (A,L,M) and the other defined by hat(M,R,C). Hence, you should be able to reinvent the construction that subdivides the hat (A,B,C) into these two hats.

The two half-spans join smoothly with position and first derivative continuity. Hence we say that they for a C^1 curve. In general, two spans (A,L,M) and (N,R,C) do not necessarily form a continuous curve. To obtain a continuous, i.e., C^0 curve, we must have M=N. To obtain a C^1 curve, we must have L+AL=R+CR, which is equivalent to LR+LA+CR=0, and to LR+LA+AC+CR=AC, which yields 2LR=AC.

Now, we discuss how to recover (A,B,C) from other representations. You should be able to reinvent these formulae by looking at the figure above.
Given (A,M,C), $H=½(A+C)$ and $B=M+HM$.
Given (A,G,C), $H=½(A+C)$ and $B=H+¼G$.
Given (A,V_0,C), $B=A+½V_0$.
Given (A,V_0,M), $B=A+½V_0$, $C=A+V_0+4BM$.
Given (A,V_0,V_1), $B=A+½V_0$, $C=B+½V_1$.
Given $(M,V_{½},G)$, $B=M–⅛G$, $H=M+⅛G$, $A=H–½V_{½}$, $C=H+½V_{½}$.

3.4 Cubic Bezier spans

3.4.1 Motivation and definition

Cubic Bezier spans are more versatile than quadratic spans, because they may represent curves with inflections and because, when the four control points are not coplanar, the resulting curve is also not coplanar. Furthermore, them may be pasted together to define curves with a smoothly varying first and second derivative, which is important in many aesthetic and analysis applications, as the human visual system may interpret abrupt changes of curvature as discontinuities and as curvature discontinuities in boundary conditions may produce turbulences in fluid flows. Hence, cubic Bezier curves are popular tool for modeling smooth curves and motions.

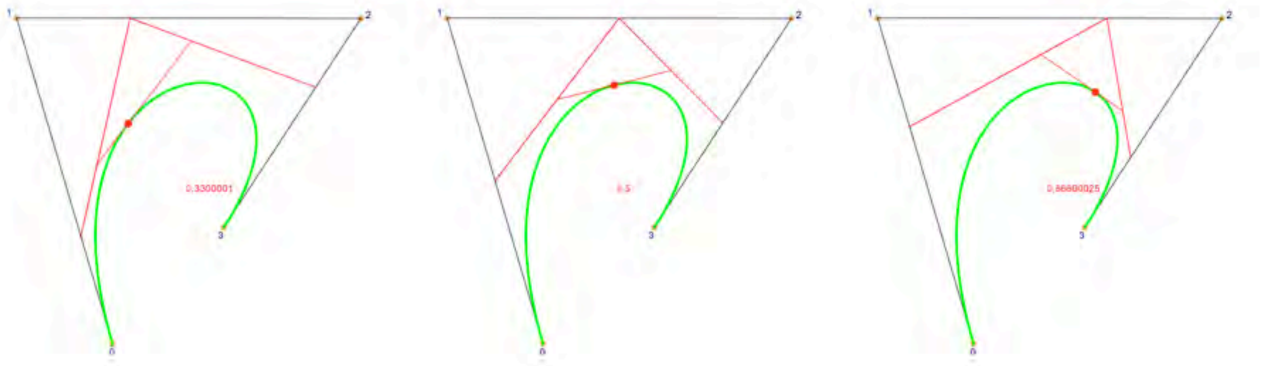
3.4.2 Integration

Since this is a cubic curve, we require that the fourth derivative P''''(t) be zero, implying (by integration) that the third derivative (jerk) be constant: P''' (t)=J.

To produce a parametric formulation of a cubic span, we start with P'''(t)=J and through successive integration obtain $P''(t)=tJ+G$, $P'(t)=t^2J/2+tG+V$, and $P(t)=t^3J/6+t^2G/2+tV+P_0$, where J is the constant jerk, G the initial acceleration, V the initial velocity, and P_0 the initial position. This is the general equation of a cubic curve in parametric form. Again, we restrict our attention to the span produced when t varies in [0,1].

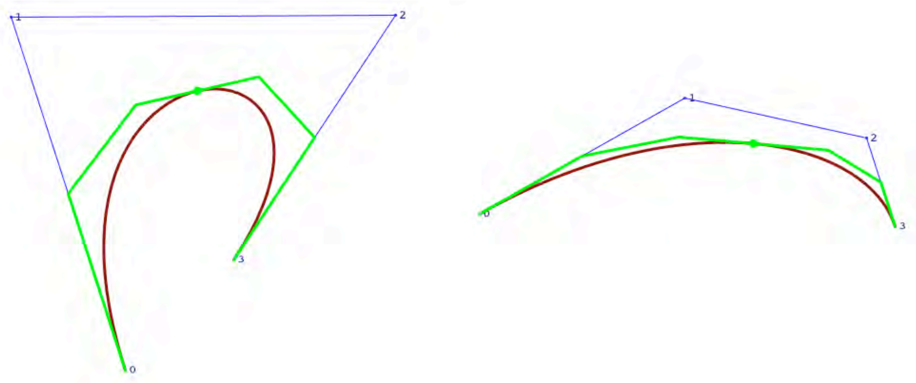
3.4.3 Evaluation of the cubic Bezier span from its control polygon

A point P(t) of the cubic Bezier span defined by the four control points (A,B,C,D) may be computed using a point-valued function: `pt Q(A,B,C,D,t) {return L( Q(A,B,C,t) ,t, Q(B,C,D,t) ) }`, formulated as a linear interpolation of two quadratic Bezier evaluations, or equivalently using the explicit formulation, as `pt Q(A,B,C,D,t) {return L( L(L(A,t,B),t,L(B,t,C)) ,t, L(L(B,t,C),t,L(C,t,D)))}`, performing three levels of a synchronized linear interpolation. The construction is shown below for three different values of t.



3.4.4 Subdivision

As for the quadratic Bezier curve, a cubic Bezier curve (in dark red below) with control polygon A, B, C, and D (in blue below) may be easily split into two sub-spans and the control polygons (in green below) of these sub-spans may be easily computed from A, B, C, and D, as the subset of the intermediate points computed during the evaluation above. **Try writing the subdivision algorithm**. The construction is shown for t=½, but may be carried out for other values. This subdivision may be performed recursively. Note that at each stage, the control polygon becomes significantly flatter (right).



3.4.5 Polynomial coefficients in terms of control points

By developing the explicit formulation above, one obtains $P(t)=(1-t)^3A+3(1-t)^2tB+3(1-t)t^2C+t^3D$.

3.4.6 Matrix formulation

By developing the polynomial in t of $P(t)=(1-t)^3A+3(1-t)^2tB+3(1-t)t^2C+t^3D$ and rearranging the terms, one obtains $P(t)=A(-t^3+3t^2-3t+1)+B(3t^3-6t^2+3t)+C(-3t^3+3t^2)+D(t^3)$, which in matrix form becomes:

$$\mathbf{P}(t) = \begin{pmatrix}\mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D}\end{pmatrix}\begin{pmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 3 & 0 & 0\\ 1 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}t^3\\ t^2\\ t\\ 1\end{pmatrix}$$

3.4.7 Derivatives

One can obtain P'(t) by taking the derivative of $P(t)=(1-t)^3A+3(1-t)^2tB+3(1-t)t^2C+t^3D$, but it is more convenient to take the derivative of the matrix formulation, which yields.

$$\mathbf{P}'(t) = \begin{pmatrix}\mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D}\end{pmatrix}\begin{pmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 3 & 0 & 0\\ 1 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}3t^2\\ 2t\\ 1\\ 0\end{pmatrix}$$

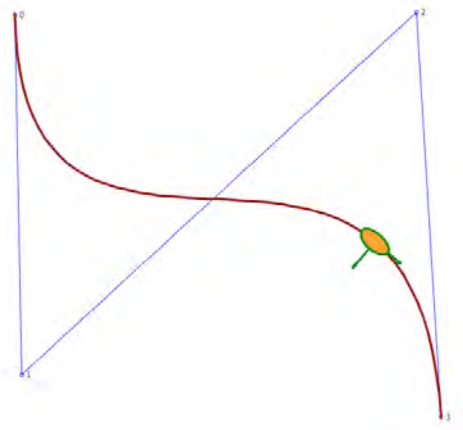
Which yields $P'(t)=(-3t^2+6t-3)A+(9t^2-12t+3)B+(-9t^2+6t)C+(3t^2)D$. In particular, P'(0)=3AB and P'(1)=3CD.

More generally, P'(t)=3(Q(B,C,D,t)–Q(A,B,C,t)). In other words, the instantaneous velocity (and hence the tangent to the curve) at point P(t) is three times the vector joining the two quadratic Bezier curves: the one with hat (A,B,C) and the one with hat (B,C,D). Knowing how to compute the derivative is important for numerous applications.

One may wish to form a more complex trajectory by smoothly joining two Bezier curves (A_1,B_1,C_1,D_1) and (A_2,B_2,C_2,D_2). If they are used to model a trajectory, then you may require that the arriving and departing velocities at their joint be equal. Hence, the common point (D_1=A_2) should be the middle of segment (C_1,B_2).

One may also want to slide an object along the curve so that its center of mass tracks P(t) and so that its axis remains aligned with P'(t).

An applet showing this behavior is posted at http://www.gvu.gatech.edu/~jarek/demos/bezier/ . A screenshot is shown below.



A second derivative yields:

$$\mathbf{P}''(t) = \begin{pmatrix}\mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D}\end{pmatrix}\begin{pmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 3 & 0 & 0\\ 1 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}6t\\ 2\\ 0\\ 0\end{pmatrix}$$

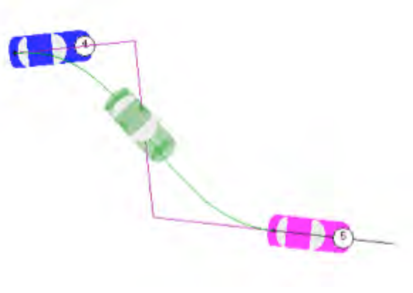
Which may also be written as P"(t)=6(1–t)(BA+BC)+6t(CB+CD). In particular, P"(0)=6(BA+BC) and P"(1)=6(CB+CD).

Knowing the second derivative is also important if one wishes to arrange two cubic spans so that at their junction they have the same first and second derivatives, which requires: $A_2=D_1$, $A_2B_2=A_2C_1$, and $C_1+B_1C_1=B_2+C_2B_2$.

Furthermore, having the closed-form expressions of the first and second derivative can be used to compute parameter values of inflection points, where the first and second derivative vectors are parallel (this provides us with an equation in t).

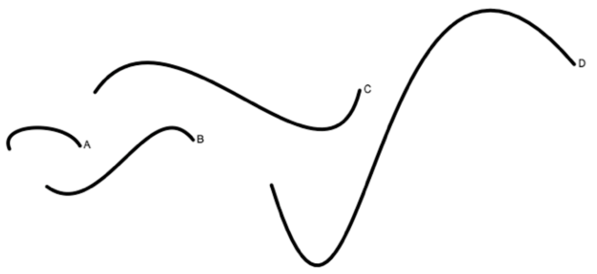
3.4.8 Hermite interpolation

Assume that we are given two positions and velocities: P(0)=P_0, P(1)=P_1, P'(0)=P_0', and P'(1)=P_1'. How would we compute the cubic function that interpolates them? This is an example of Hermite (1822-1901) interpolation. An applet showing its use for the animation of a car is posted at http://www.gvu.gatech.edu/~jarek/demos/frames/ . Press '1' to set this mode. A screenshot is shown below.



Explain precisely how the Bezier control polygon {A,B,C,D} is computed from the Hermite data, {P_0,P_1,P_0',P_1'}.

Sometimes, you are given only the tangent directions (and do not know the magnitudes of the derivatives vectors). You may have to invent their magnitude. A simple and reasonable choice is to give them the magnitude of the distance between P_0 and P_1. But this choice may flatten the curve when the two tangents are far from being aligned with the vector P_0P_1. A different choice of magnitudes is illustrated in the applet posted at http://www.gvu.gatech.edu/~jarek/demos/strokeMorphs/ for which a screen short is shown below.



Specifically, each edge is drawn as a cubic Bezier curve. Given control points A and D and tangents A' and D', we construct B=A+rA' and C=D–rD', such that ||BC||=2r.

```
void drawCurve() {
    if(a==0 && b==0) {A.to(B); return;}
    float s=biArcSolve(A,U(V(A,B)).makeRotatedBy(a),U(V(B,A)).makeRotatedBy(b),B);
    float d=d(A,B)*3;
    float r=d*s/(d+s);

drawCubicBezier(A,S(A,r,U(V(A,B)).makeRotatedBy(a)),S(B,r,U(V(B,A)).makeRotatedBy(b)
),B);
   }
```

using `biArcSolve`, detailed below, to compute r:

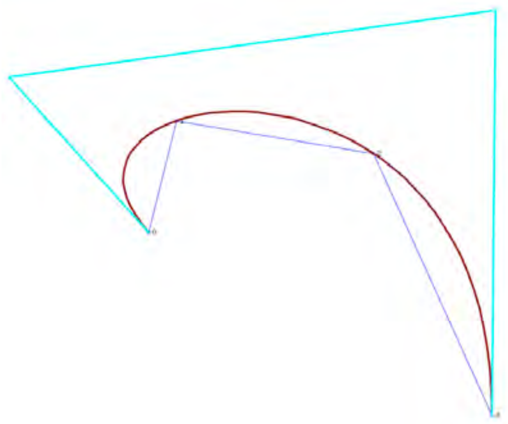
```
float biArcSolve(pt A, vec U, vec T, pt B) {
vec BA=V(B,A), UT=A(U,-1,T);
   float BA2=d(BA,BA), BAUT=d(BA,UT), UT2=d(UT,UT);
   float d=sq(BAUT)-BA2*(UT2-4);
   return (BAUT+sqrt(d))/(4-UT2);
   };
```

3.4.9 Images of B and C

We may think of the process of converting the control polygon into its cubic Bezier curve as a bending. The image B' of control point B would be P(⅓) and similarly, the image C' of control point C would be P(⅔). Plugging these parameter values into $P(t)=(1-t)^3A+3(1-t)^2tB+3(1-t)t^2C+t^3D$, we observe that B'=(8A+12B+6C+D)/27 and C'=(A+6B+12C+8D)/27, which may also be written as B'=B+(8BA+6BC+BD)/27 and C'=C+(CA+6CB+8CD)/27.

3.4.10 Retrofitting

Suppose that you want to compute the control polygon (A,B,C,D) of a Bezier cubic curve that satisfies the following four point-constraints: P(0)=A, P(⅓)=B', P(⅔)=C', and P(1)=D. To solve this interpolation problem, we solve the system of the two equations defined above and obtain: B= (–15A+54B'–27C'+6D) / 18 and C= (–15D+54C'–27B'+6A) / 18. An applet illustrating this retrofitting construction is available at http://www.gvu.gatech.edu/~jarek/demos/retrofitBezier/ . A screenshot is shown below, where the cyan control polygon is computed from the interpolated blue one.

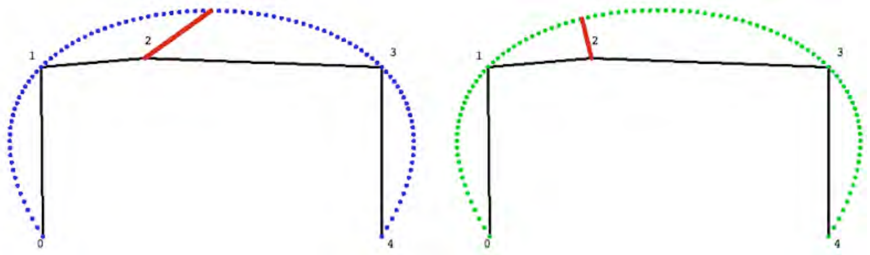


Note that we have solved above the general problem of fitting a cubic parametric curve through four points (A, B', C', and D') for uniformly spaced parameter values. The Bezier formulation is only a convenient representation of this general result.

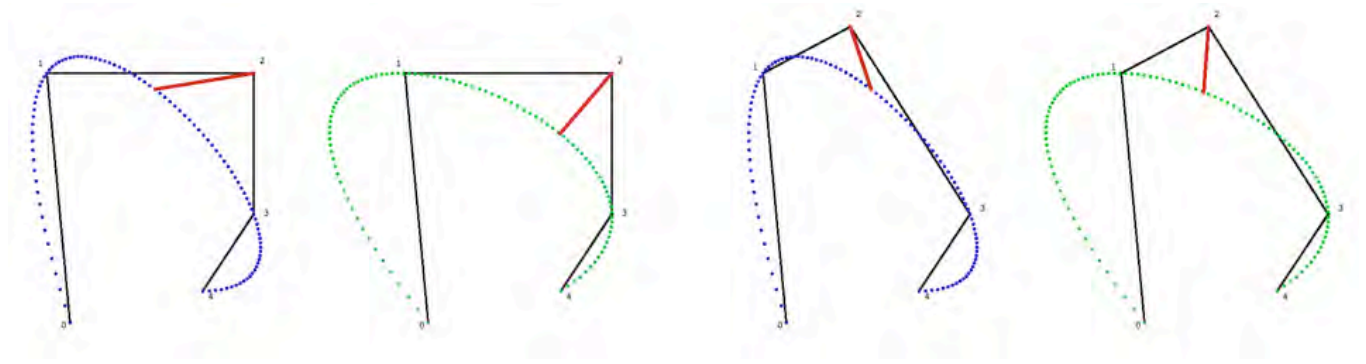
3.4.11 Midcourse point of a cubic span that interpolates four points

Now, assume that you are given four points, (A,B',C',D) as before, you fit an interpolating cubic span through them, as above, and now you want the mid-course point M=P(½). This is important for subdivision. Working through the formula and substituting t, one obtains: M=(B'+C')/2+(AB'+DC')/16. You should practice drawing this construction by hand and must remember it. Another way of explaining it is to say, start at the midpoint of B' and C' and move by ⅛ away from the midpoint of AD. Verify on an example that the two constructions are equivalent.

For example, consider five control points P_0, P_1, P_2, P_3, and P_4. We wish to adjust the position of P_2 by a correction vector (red in the figure below) that places it as the midpoint of a cubic span that interpolates the other points. The blue solution (left) computes the cubic fit using retrofitting so that $P(-2)=P_0$, $P(-1)=P_1$, $P(1)=P_3$, and $P(2)=P_4$ and then sets the new P_2 to P(0). Note that the new point is independent of the original position of P_2. The green solution (right) sets the new P_2 to P(t), where the value of t is computed from the ratios of the edge lengths.



Furthermore, the cubic fit in the green solution uses constraints $P(0)=P_0$, $P(s/w)=P_1$, $P((s+t)/w)=P_3$, and $P(1)=P_4$, where $s=\|P_0P_1\|$, $t=\|P_1P_3\|$, $u=\|P_3P_4\|$, and w=s+t+u and hence takes into account the edge lengths.



3.4.12 Convex hull

The cubic Bezier span is contained in the convex hull of its four control points.

Prove that it is not always contained in the union of the two triangles (A,B,C) and (A,C,D), but that it must always be contained in the union of the three triangles (A,B,C), (A,C,D), (B,C,D).

3.5 Bezier curves of arbitrary degree

3.5.1 De Casteljau's algorithm

The definition and construction of the Bezier curve in terms of its control points was patented by Pierre Bézier (1910-1999), while developing the UNISURF CAD/CAM system at the French car company Renault (1933-1975). In 1985, he received the ACM SIGGRAPH "Steven Coons" award for his lifetime contribution to computer graphics and interactive techniques.

The construction has also been discussed in 1959 by Paul de Casteljau (1930-) as he was working at another French car company, Citroën. De Casteljau proposed a numerically stable (recursive subdivision) algorithm for evaluating Bezier curves, or more generally, polynomials in Berstein form.

A polynomial B(t) of degree n may be written in Berstein form as

$$B(t) = \sum_{i=0}^{n} \beta_i b_{i,n}(t)$$

where b is the Bernstein (1880-1968) basis polynomial

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

It can be evaluated for a given value of t using the recurrence relation

$$\beta_i^0 = \beta_i$$
$$\beta_i^j = (1-t)\beta_i^{j-1} + t\beta_{i+1}^{j-1}$$
$$B(t) = \beta_0^n$$

Applying this evaluation technique to curves, we start with a series of n+1 control points P_i. Then, a point B(t) on the Bezier curve of degree n defined by these control points may be computed as

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{P}_i b_{i,n}(t)$$

using the same recurrence, but now the β_i^j represent points.

3.5.2 Implementation

The formulation above leads to an elegant implementation (below) of the point B(t) on the Bezier curve of degree n defined by n+1 control points. B(t) is returned by the call `bez(0,t,n-1);` The implementation of `bez` uses a recursive call:

```
pt bez(int i, float t, int r)
```

```
        {if(r==0) return P[i];
         return L(bez(i,t,r-1),t,bez(i+1,t,r-1));}
```

It uses L(A,t,B) which returns point A+tAB and is implemented as

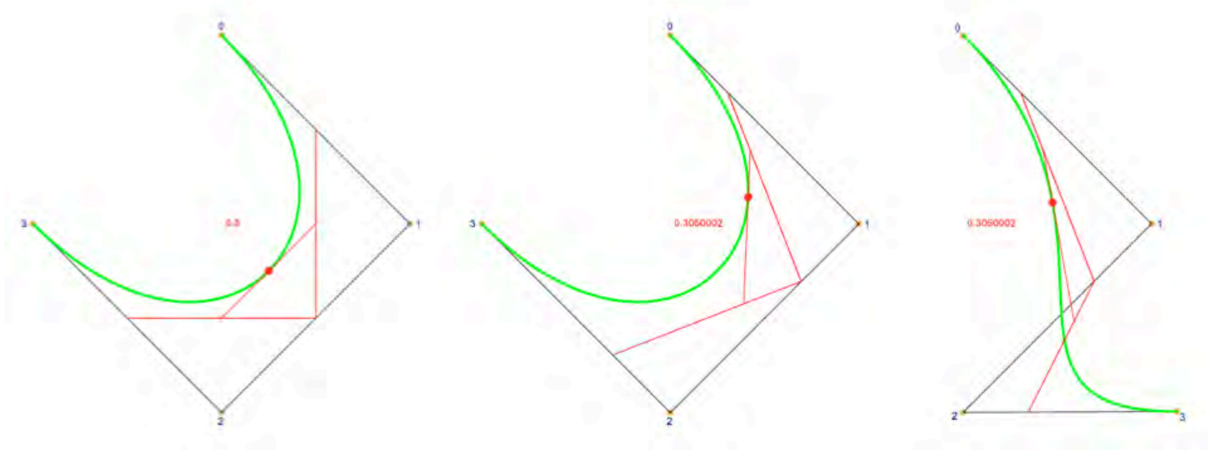
```
pt L(pt A, float s, pt B) {return P(A.x+s*(B.x-A.x),A.y+s*(B.y-A.y)); };
```

You need to be able to reinvent this recursive algorithm. Try it now.

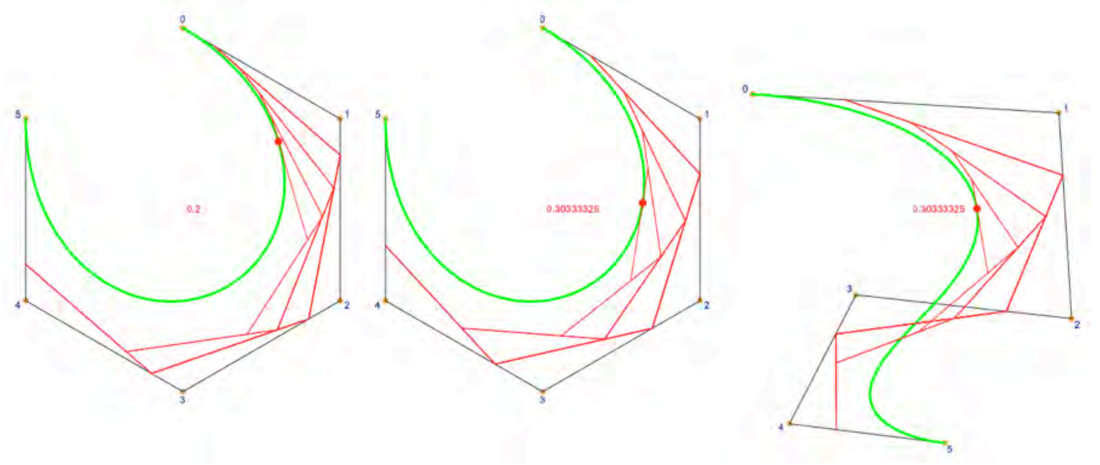
To plot the curve (green below), use the call in a loop that increments the parameter t, and send the coordinates of the produced points to the graphic system using the `.v()` method, which, when executed between `beginShape()` and `endShape()` starts the curve (when the first point is sent) or extends it by adding a line segment to each new point.

```
beginShape(); for(float s=0; s<=1; s+=0.01) bez(0,s,n-1).v(); endShape();}
```

The figure below illustrates a cubic Bezier curve (green), its control polygon (black), and the computation (linear combinations shown as red construction lines) of a particular point B(t) (red dot) for different values of parameter t (left and center). The image on the right shows the influence of the last control point on the shape of the curve, which now has in inflection point.



Using 6 control points, the same code yields a quintic (much smoother) Bezier curve, as shown below.



Note that higher order Bezier curves of tortuous control polygons tend to pass far from the intermediate control points (as shown above, right). The different curve formulations proposed in the literature (and discussed below) strive to reduce this gap, while preserving as much of the smoothness as possible.

4 - Polygonal curves

4.1 Definition and global parameterization of a polygon

A polygon (polygonal curve) is the set of edges that join consecutive **vertices** (what we have been calling above the control points). We can construct a global parameterization as follows. Given a series of control point P_0, P_1… P_n, we construct an interpolating curve P(t) as a piecewise linear curve, $P(t) = L(P_i,t–i,P_{i+1})$ for i such that i≤≤t<i+1.

4.1.1 Closed and open loops

The above formulation defines an open-loop polygon, which we will call **stroke**. For a closed-loop curve, which we will call **loop**, one may add a loop-closing edge $L(P_n,t–i,P_0)$ from the last to the first vertex.

4.2 Point-in-polyloop test

Assume that P is a loop that is not self-intersecting. Given a point Q, how could one test whether Q lies in the interior of P?

Several approaches exist. The most common one is to shot a ray from Q towards infinity and use the parity of the number of intersections. (We must pick a ray that avoids vertices of P.) If the number of intersections is even, then Q is out. To justify this claim, consider a point Q' at infinity along the ray. Initially Q' is out. As Q' approaches Q by sliding along the ray, each time it crosses P it changes status: out—in—out—in. Its status when it arrives at Q is the status of Q. Hence, when it crossed P an even number of times, Q' and hence Q is out.

Another equivalent approach picks an arbitrary point O, which does not have to be inside P. Then, it considers the triangles that O forms with each edge of P. We test Q for inclusion in all these triangles. If the number of triangles that include Q is even, then Q is out. Explain why this formulation is equivalent to the previous one.

The point-in-polygon test and the computation of the closest point on the boundary of the polygon are illustrated in the applet http://www.gvu.gatech.edu/~jarek/demos/loop/ where the point closest to the mouse on the loop is shown with a different color depending on whether the mouse lies in or out of the polygon.

4.2.1 Numeric issues

In the above algorithm, Q may lie on the common borders of two triangles. Counting them both will result in the wrong answer—not counting either of them, will also produce the wrong answer. Suggest a practical technique for ensuring that the point-in-triangle test returns results that are consistent with the parity test used above.

4.3 Local differential estimators

Consider a polyloop L defined by a cyclic sequence of vertices. Often, we will want to discuss a particular vertex in the context of its neighbors in the sequence. To simplify notation, we will use consecutive letters and for instance refer to a vertex C in a sub-sequence A,B,C,D,E of consecutive vertices of L. In particular, we would talk about an edge AB or BC of the polyloop. First, we discuss how to estimate the local differential properties of a smooth, although unknown, curve J passing through the vertices of L.

4.3.1 Velocity estimator at a vertex

Assume that, as you travel along J, it takes one unit of time to travel from any one vertex of L to the next one. As you travel along an arc of J between vertices A and B, the average velocity vector will be AB. Hence, we estimate the **velocity** vector at B by the average V=(AB+BC)/2. Note that V=(AB+BC)/2=(B–A+C–B)/2=(C–A)/2=AC/2.

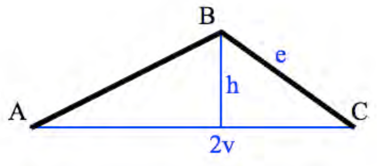
We therefore define the **tangent** vector to J at B to be AC.direction and the **normal** vector to J at B to be AC.direction.left.

4.3.2 Acceleration estimator at a vertex

The **acceleration** g at B is estimated as the change in velocities between the two edges incident upon B. Hence, the acceleration at B is defined as g(B) = BC–AB = BC+BA = 2((A+C)/2–B), which is twice the vector from B to the midpoint of A and C.

4.3.3 Curvature estimator at a vertex

The radius of curvature at a vertex B may be estimated as the radius of the circle passing through A, B, and C. Unfortunately, this circle-fit solution yields unsatisfactory results when the angle at B is acute. A better estimate is provided by a **parabolic-fit** solution, where the **radius r of curvature** at B is estimated by $r=V^2/(2h)$, where V is the velocity at B (defined above) and where h is the distance between B and the line passing through A and C. To better understand this formula, note that 2h is the norm of the normal component of acceleration g(B). Hence, h=AB•AC.left.direction which we would implement as `h=dot(V(A,B),U(R(V(A,C))))`, where `V(A,B)` returns the vector AB, `R(V)` returns the version of vector V rotated 90 degrees, and `U(V)` returns the unit vector obtained by **normalizing** V (i.e., dividing it by its norm). Furthermore, the **normal acceleration** of a point traveling on a circle of radius r with tangential velocity V is V^2/r. Hence, $r=V^2/(2h)$.



4.3.4 Jerk estimator at an edge

The **jerk** is the change of acceleration. We associate a jerk with each edge of L. The jerk of edge BC is the difference of accelerations j(B) = g(C)–g(B) = 2((B+D)/2–C)–2((A+C)/2–B) = –A+3B–3C+D. The jerk measures the change in the force felt by a person traveling along the curve. Below, we visualize the normal component of the jerk to compare the smoothness of several subdivision schemes.

4.4 Self-intersecting polygonal curves

A loop may self intersect. Does it, in that case, still provide an unambiguous representation of the boundary of a planar region? It does, but one may consider different interpretations.

4.4.1 XOR

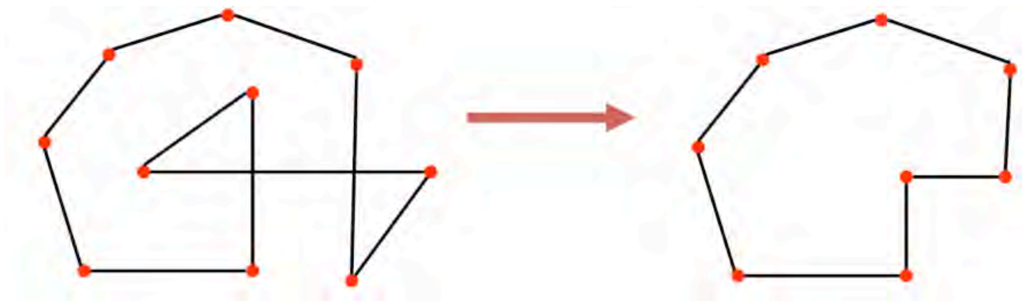
The point-in-polygon rules (based on parity of the number of ray intersections or of containing triangles) may be applied to self-intersecting loops. Note that at each point of the loop, except for the self-intersection and overlap points, the curve separates the interior from the exterior.

4.4.2 Winding number

At any point Q, shoot a ray to P(t) when t=0 and then watch that ray as t evolves and P(t) slides around the closed-loop curve back to the starting point. In particular keep track of the winding number $w_P(Q)$ of Q with respect to P, which is the signed count of the number of 360 degree turns that the ray has made. By convention, we pick the counter-clockwise direction as positive. When the loop is free from self-intersections, points outside have winding number zero and points inside have either all winding number 1 or all winding number –1, depending on the clockwise/counterclockwise orientation of the loop.

4.4.3 Trimming

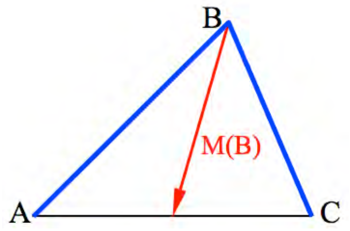
We can use winding numbers to devise more interesting schemes for defining what portions of the plane are inside the loop when it is self-intersecting. For example, propose a definition that would produce the trimming below and show another example where your definition fails to produce a result that most humans would expect. Find Heisserman's work in this area (maybe his PhD dissertation) and explain the definition he proposes.



5 - Smoothing polygonal curves

5.1 Local operators for polygonal curves

One way of **increasing the smoothness** of a polyloop L is to adjust the position of the vertices. The **Laplace** of B in the subsequence A,B,C is the vector M(B)=(BA+BC)/2. Note that M(B) is half the acceleration g(B)/2.



Let `tuck(s)` be a process which for each vertex B in the subsequence A, B, C computes M(B) and then moves each vertex B by a fraction s of M(B) to B+sM(B). A sequence of `tuck(s)` steps for a value of s in [0,2/3] will eliminate the sharp features of L, producing a fairer polyloop and in some sense filtering out the high frequency. For example, `tuck(2/3)` moves B to the **average** (A+B+C)/3. This averaging process is a low-pass filter.

The **dual** operator, `dual`, inserts a new vertex in the middle of each edge and then deletes the old vertices. The **dual-squared** operator, `dual2`, is the sequence `{dual; dual;}`. We can associate each vertex `dual2`(B) of the polyloop produced by applying `dual2` with the vertex B of the original polyloop so that `dual2`(B)=(A–6B+C)/4. The adjustment vector `dual2`(B)–B is (BA+BC)/4. Note that `dual2` is equivalent to `tuck(1/2)`.

Unfortunately, a sequence of `tuck(s)` steps changes the curve even in smooth parts and pulls it inside the turns. In fact, applying a large (maybe thousands) number of these steps will shrink the whole curve to a point. To avoid, or at least reduce the shrinking effect, one may use the **push/pull** approach, which repeats the sequence `{tuck(s); tuck(-t);}`. The `tuck(s)` step of the push/pull reduces the details and shrinks the polyloop. The `tuck(-t)` exaggerates the details of the smoother curve (hence does not fully reproduce the original details) and also tends to snap back the curve to its original position in smooth parts. The values of s and t parameters may be chosen to implement different filter behaviors.

For example, the sequence `{tuck(2/√6); tuck(-2/√6);}` moves each vertex C in the subsequence A,B,C,D,E to its position F(0)=(–A+4B+4D–E)/6 on the **parametric cubic curve** F(t) whose coefficients are computed so that it interpolates the neighbors: F(–2)=A, F(2)=E, F(–1)=B, and F(1)=D. Although the value 2/√6=0.82 is to large for practical use and will produce instabilities, it is interesting to note that `{tuck(s); tuck(-s);}` moves C towards that F(0) position on the cubic defined by its neighbors. Hence, the motion of vertices stops if they have converged to a uniformly sampled cubic curve.

The formula for F(0)=(–A+4B+4D–E)/6 which we used above was derived by setting F(u)=au^3+bu^2+cu+d. The four constraints invoked above yield for equations, which we want to solve for d: –8a+4b–2c+d=A, 8a+4b+2c+d=E, –a+b–c+d=B, and a+b+c+d=D. The first two yield A+E=8b+2d. The last two yield B+D=2b+2d. Multiplying the second result by 4 and subtracting the first yields 6d=4(B+D)–(A+E). You should try to re-derive this formula yourself.

To verify that `{adjust(2/√6); adjust(-2/√6);}` does indeed move C to (–A+4B+4D–E)/6, we compute the images by `adjust(2/√6)` of B', C' and D' of B, C, and D in the subsequence A, B, C, D, E and then compute the final position of C by applying `adjust(-2/√6)` to C' in the sub-sequence B', C', D'.

Different formulae for computing the cubic fit and the best location for C may be explored with the applet posted at http://www.gvu.gatech.edu/~jarek/demos/cubicfit/

6 - Subdivision and J-splines

The effects of the operators introduced here and their combinations that implement the various subdivision schemes may be studied using the applet posted on http://www.gvu.gatech.edu/~jarek/demos/uniform/

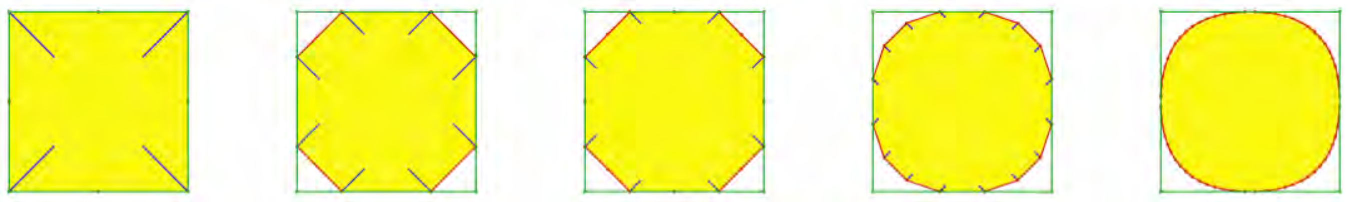
The reader can edit the red control curve by dragging vertices, deleting them by dragging them out of the window, or inserting new ones by clicking close to the middle of an edge. Then, to explore the steps of ca subdivision scheme, press 'c' to copy the new control polygon into the working copy. Then, perform the various operations by pressing 'r' to refine, 'd' to decimate, 't' to tuck, and 'u' to untuck. One may also execute combinations: 'f' for {t,u} and j for {r,t,u}, or keep 'y' pressed and edit the control loop. Try inventing new combinations (for example, repeating {r,u,t,t,u} that remove sharp features or exaggerate them. Write them down and write down their effects. Notice whether they interpolate the control vertices.

If the vertices of the polyloop L form a coarse sampling of some original curve J, then the smoothing technique discussed above will not make L smoother. In fact, it will quickly shrink L into a small circle. Clearly, additional vertices are needed for smoothing coarse polyloops.

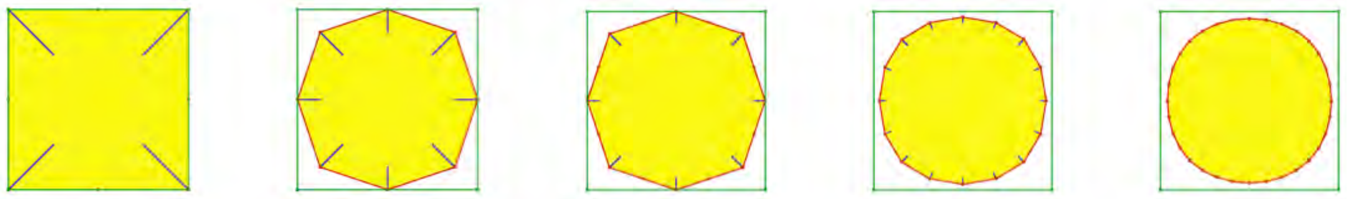
Let `refine` be an operator that inserts a new vertex in the middle of each edge. Applying a short sequence of `refine` operations and then applying a smoothing technique will produce a smoother looking polyloop. Unfortunately, this solution is slow and does not offer the flexibility of the smoothing schemes outlined below.

6.1 Subdivision schemes

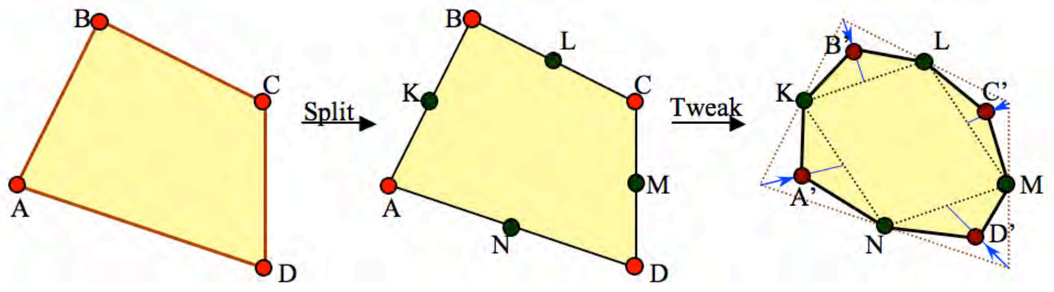
The simplest subdivision scheme is to repeat the sequence `{refine; dual;}`. As we do so, the polyloop converges to the **quadratic uniform B-spline curve** having L as the original **control polyloop**. Note (below) that the final curve J_{rd} interpolates the mid-edge points of L, but not the vertices of L. We show below, the initial control polyloop (green square) and the initial state of the subdivision curve (shaded yellow area). The blue lines show the vector M(B) for every vertex B. From left to right we show the result for r, rd, rdr, rdrd, rdrdrdrd, where r stands for `refine` and d for `dual`.



Repeating `{refine; dual2;}` converges to the **cubic uniform B-spline curve** having L as the original control polyloop. From left to right we show below the result for r, rd^2, rd^2r, rd^2rd^2, $rd^2rd^2rd^2$, where r stands for `refine` and d^2 for `dual2`.



Note that the mid-edge vertices are not affected by the `dual2` step. Hence, this scheme may be more efficiently implemented as a Split&Tweak sequence: Split inserts a new vertex in the middle of each edge. Tweak tucks in the old vertices by moving them halfway towards the average of their new neighbors.

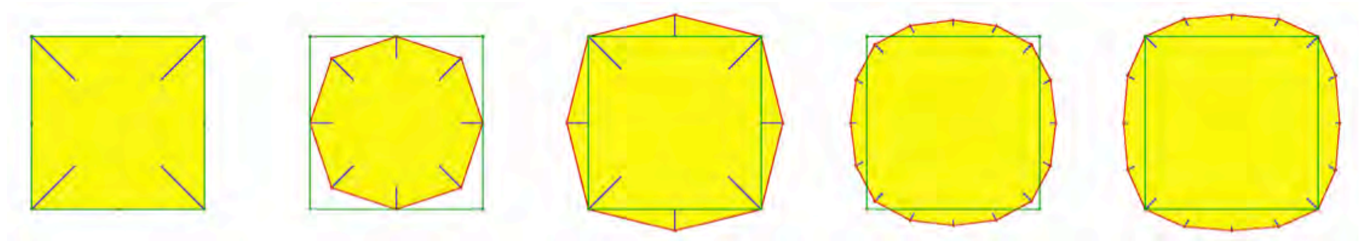


Note that the limit cubic B-spline may be easily represented by a series of smoothly connected cubic Bezier spans. Each edge of the original control polygon is bent to produce the control polygon of a Bezier span. To do so, simply **break each edge into three equal parts. Then, move each original vertex to the average of its new neighbors**. Verify that this construction meets the requirements that consecutive cubic Bezier spans meet with tangent and acceleration continuity. The conversion to Bezier has several advantages over the subdivision scheme: (1) Each span lies in its convex hull, hence, one can quickly decide which spans may intersect the window or another geometry and discard those which are guaranteed not to. (2) We can use various techniques discussed earlier for evaluating the Bezier curve efficiently.

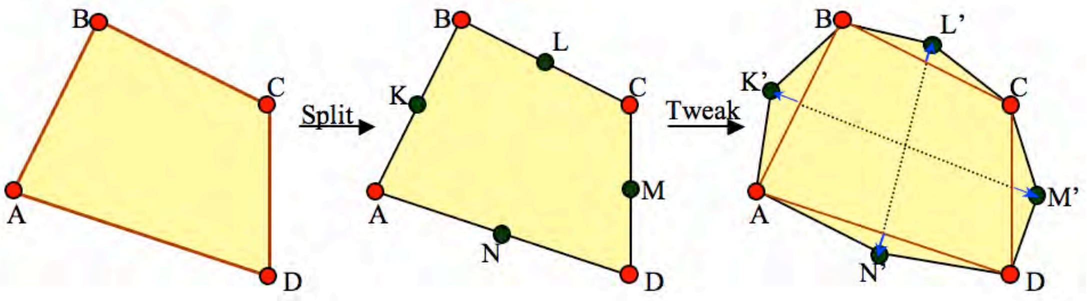
Also, remember that the Bezier curve interpolates the first and last of its control points. Hence, the tucked in versions of the original vertices are guaranteed to lie on the limit B-spine curve. For example, the image on a cubic B-spline of a control point B in the sequence (A, B, C) is B+(BA+BC)/6.

Note that the final curve J_1 is relatively far from sharp vertices of L and does not interpolate the edge midpoints. It tends to push the vertices inside the turns.

To overcome this tendency, we may use the **four-point scheme**. Remember that `dual2` is equivalent to `tuck(1/2)`. Hence, we may try to use the push/pull idea developed above to undo this push. To do so, we repeat the sequence `{refine; tuck(1/2); tuck(-1);}`. From left to right we show below the result for r, rt, rtu, rturt, and rtutu. where r stands for `refine` , t for `tuck(1/2)` and u ("untuck") for `tuck(-1)`.



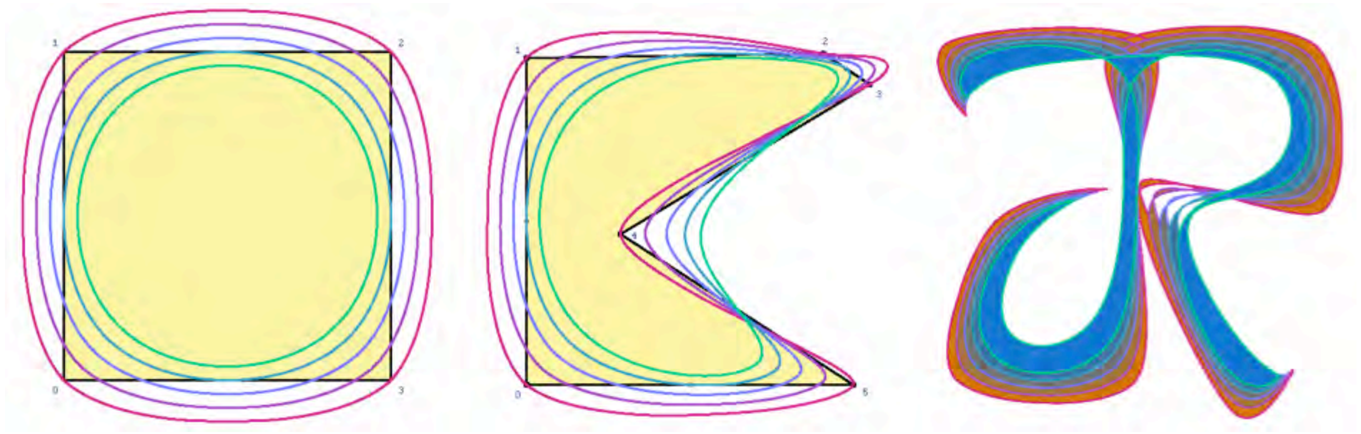
The four-point subdivision scheme may be implemented efficiently as a variation of the Split&Tweak process. However, here, the Tweak bulges out the new vertices by 1/8th away from the average of their third-degree neighbors. For example, the mid-edge point M (below) would by moved to M' by a vector that pushes it by 1/8 away from the midpoint of A and B.



Under the four-point subdivision, the polyloop converges to a smooth curve that interpolates the vertices of L.

We define the **Jarek** family J_s of subdivision curves that generalize the B-spline and Four-point schemes presented above. Specifically, let J_s be the curve towards which the polyloop L converges when subjected to the sequence of `J(s)= {refine; tuck(1/2); tuck(s-1);}`subdivisions.

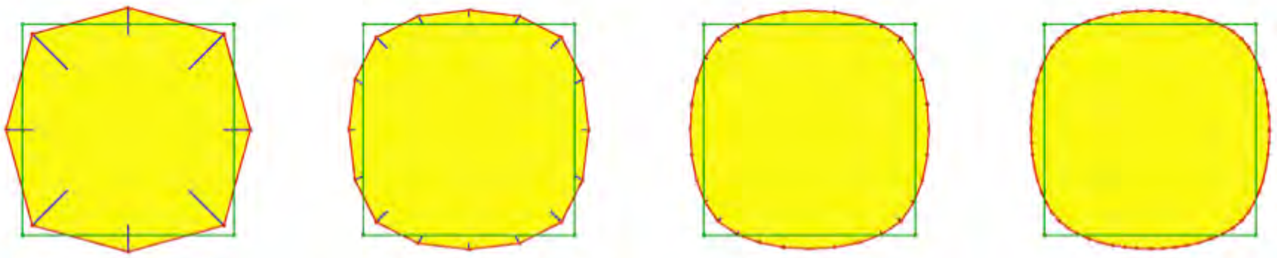
We show below, inwards out, J_0, $J_{¼}$, $J_{½}$, $J_{¾}$, and J_1 for a square control polyloop and for a non-convex one. Note that although $J_{¼}$ osculates the edges at their midpoints, it is different and smoother than J_{rd}. On the right, we show a superposition of the regions enclosed by these curves for a self-intersecting control polyloop.



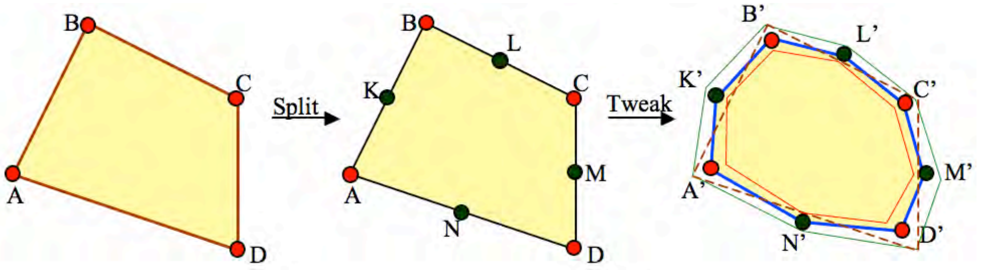
J_0 is the **cubic uniform B-spline** (in green above). Note that the first iteration of the push/pull pushes its original vertices far inwards the turn. As a consequence, the final curve remains far from the original vertices.

J_1 is the **Four-point** subdivision curve (in red above). Note that it interpolates the vertices of L, but appears less smooth (below right).

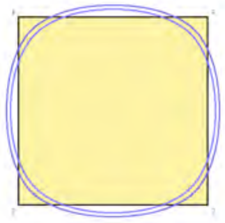
$J_{½}$ (the central curve in blue above) is a compromise between J_0 and J_1. Usually, L and $J_{½}$ have nearly equal areas.



$J_{½}$ can also be implemented as a Split&Tweak process, but here both the old and the new vertices are tweaked, each by half of their tweak vectors in the previous two schemes. (It is sometimes referred to as the Jarek curve.)

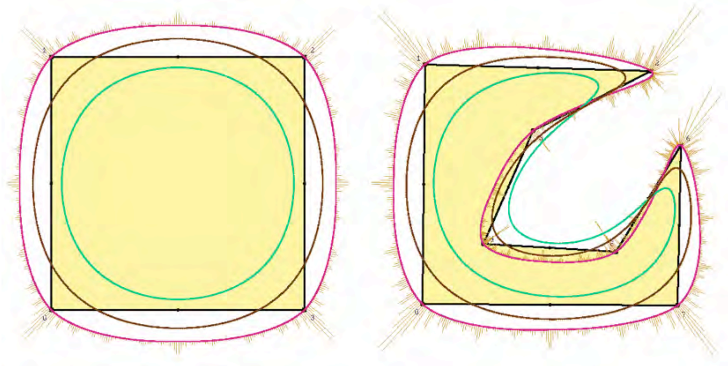


$J_{⅝}$ offers a different compromise between J_0 and J_1. It is a better approximation of L than $J_{½}$ when the quality of an approximation is measured by the Hausdorff distance (maximum discrepancy discussed in the next chapter). Hence, L is a better low-resolution approximation of $J_{⅝}$ than it is of other subdivision curves. Therefore, we suggest to use $J_{⅝}$ for the multi-resolution rendering, because the yields a less visible popping artifact that the other subdivision schemes discussed so far. We compare below $J_{½}$ (smaller blue) and $J_{⅝}$.



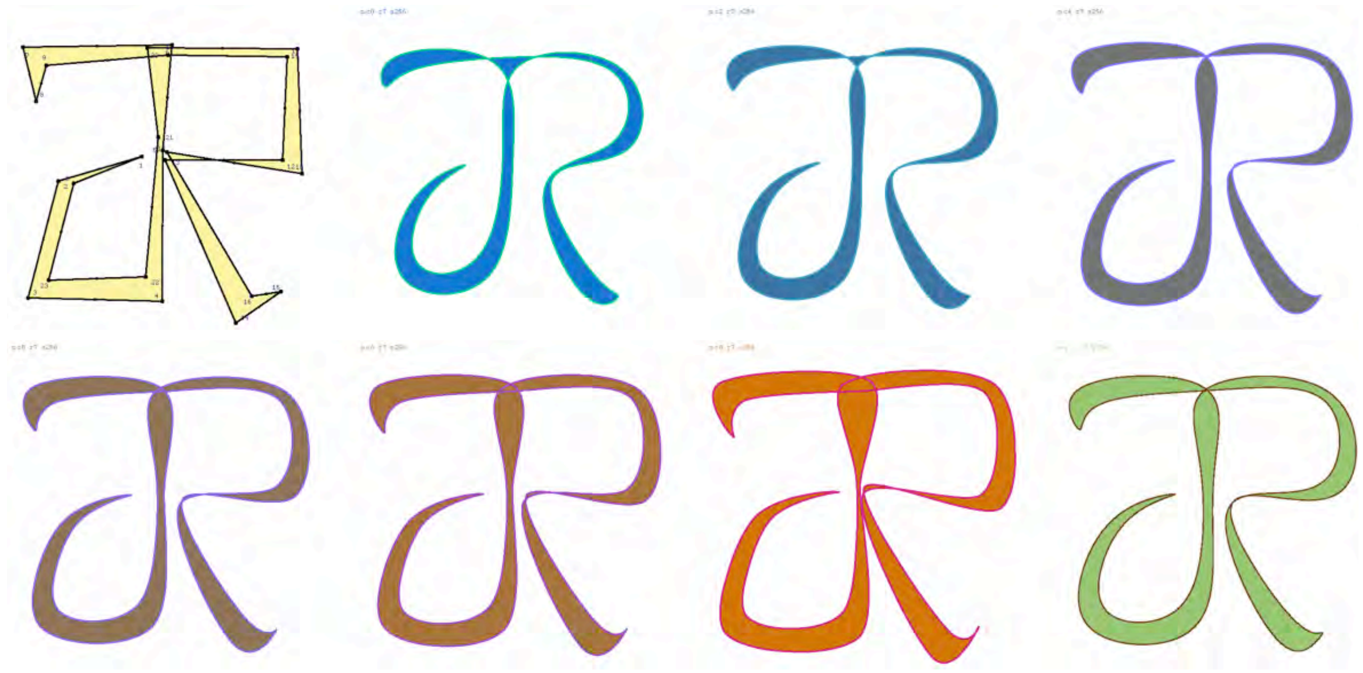
6.2 Continuity

To compare the smoothness of the various curves, we plot (below) their normal jerk, scaled by the same amount for all three curves in a figure. Note that the jerk of J_0 (green) is barely visible, while the jerk of J_1 (red) is significant and irregularly distributed.

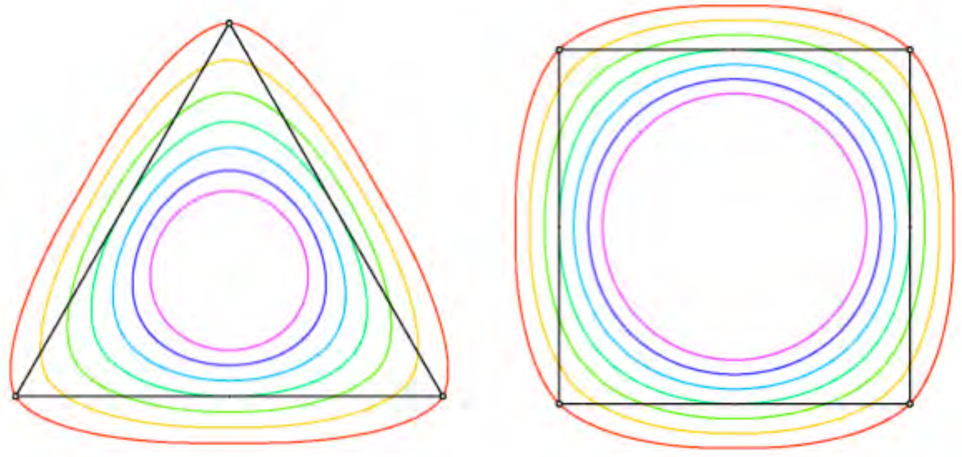


The intermediate curve **J_m** (shown in brown above between J_0 and J_1 has no noticeable jerk and produces more pleasant shapes than the other subdivision schemes. It is produced by a mixed scheme where the first two iterations use negative s.

We show below the shapes produced by the various schemes: L, J_0, $J_{1/4}$, $J_{1/2}$, $J_{5/8}$, $J_{3/4}$, J_1, J_m. Notice that each scheme has distinct characteristics You may explore these schemes at http://www.gvu.gatech.edu/~jarek/demos/jsplines/



More precisely, we have shown that the limit curve is C^4 when $s = 3/2$, C^3 when $1<s<3/2$ and $3/2<s\leq 2.8$, C^2 when $0<s\leq 1$ and $2.8<s<4$, and C^1 when $-1.7\leq s<0$ and $4\leq s\leq 5.8$. An example of a J-spline family is shown below, starting with J_0 (the outer interpolating four-point curve in red), incrementing s by ¼ each time, and finishing with the quintic B-spline (inner magenta curve), the first four derivatives of which are continuous.



6.3 Ringing J-splines and rendering surfaces and animations

Read the paper "Ringing subdivision of curves and surfaces" by Rossignac and Venkatesh, available at http://www.gvu.gatech.edu/~jarek/papers/ringing.pdf and briefly explain how and why the ringing approach works. Explain how many control points are needed to produce a span (image of an edge of the control polygon)?

Resources

Education-Driven Research in CAD. J Rossignac. Computer-Aided Design Journal (CAD), Vol 36/14 pp 1461-1469, 2004. GVU Tech Report GIT-GVU-03-26. PDF

J-splines. J. Rossignac, S. Schaefer. Journal of Computer Aided-Design (JCAD), 40(10-11):1024-1032, Oct-Nov 2008. DOI. PDF. Accompanying applet: http://www.gvu.gatech.edu/~jarek/demos/ring/

Ringing subdivision of curves and surfaces. J. Rossignac, A. Venkatesh. IEEE Computer Graphics & Applications 2010. PDF. Accompanying applet: http://www.gvu.gatech.edu/~jarek/demos/ring4/

Unary shape operations. J. Heisserman, R. Woodbury. In Geometric Modeling for Product Realization. IFIP Transactions B-8 North-Holland. Peter R. Wilson, Michael J. Wozny, Michael J. Pratt (Eds.): pp. 63-80, 1992, ISBN 0-444-81662-3.

Practice

Exercises

1) Explain what is a quadratic Bezier span and how to trace it, given the 3 control points.
2) Explain how to compute the hat of the quadratic Bezier span that interpolates three given points.
3) Explain how to compute a point for a given parameter t on a cubic Bezier span from its control points.
4) Discuss the advantages of a cubic Bezier span over a quadratic Bezier span.
5) Explain how to refine a control polygon so P as to obtain a series of control polygons for cubic Bezier spans whose union is the cubic B-spline curve of P.
6) Explain how to draw a narrow band over a cubic B-spline curve by exploiting the convex hulls of the corresponding series of cubic Bezier spans.
7) Explain how to subdivide the control polygon of a cubic Bezier span in two control polygons that yield two spans whose union is the same as the original span.
8) Explain how to compute the normal at a vertex of a polyloop. Provide an intuitive justification.
9) Explain the difference between the acceleration and the curvature at a vertex of a polyloop. Provide formulae for both.
10) Provide an intuitive definition of the jerk of a curve. Give the formula for estimating it at an edge of a polyloop.
11) Provide the overall algorithm and the geometric construction for smoothing a polyloop.
12) Express tuck(1/2) in terms of duals.
13) Explain why repeating a tuck is not a great strategy for smoothing a polyloop. Propose a better strategy.
14) What is the difference between smoothing and subdivision?
15) Provide a simple polyloop subdivision scheme and give the name of the curve it produces. Illustrate its first step on a square.
16) Provide the formula for a subdivision that converges at a cubic B-spline curve.
17) Provide the construction details and implementation code of the 4-point subdivision process.
18) Compare the curves produced by the 4-point scheme and the cubic Bezier scheme. Discuss their advantages and drawbacks.
19) Explain the subdivision scheme the converges to the Jarek curve and its advantages.
20) Explain how to compute a series of control polygons for cubic Bezier spans whose union is a smooth curve that interpolates the initial vertices.
21) Explain how to produce smooth animations of smooth curves from a user-defined grid of control vertices.
22) Explain how to produce a smooth surface from a user-defined grid of control vertices in 3D.