15

Curve Fitting

Why Study Curve Fitting, Splines and Bézier curves?

- **They produce smooth, flexible curves from simple data**.
- **They're numerically stable and easy to compute**. Bézier curves use the de Casteljau algorithm, which is robust and works with linear interpolation. Splines provide local control and avoid oscillations.
- **They reduce complexity and improve performance**. A spline can approximate a complicated curve or dataset using just a few parameters, avoiding the high degree and instability of a single polynomial.

15.1 Introduction

The general problem discussed in this chapter is how to fit a curve to a given set of (2D) data points $(x_0, y_0), (x_1, y_1), \cdots (x_n, y_n)$.

If the number of points is small (say, 7 or less) then interpolation is a good way to determine the coefficients a_i of a polynomial that fits the data:

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n$$

such that

$$P(x_i) = y_i, \quad i = 0, 1, \cdots, n$$

There are many methods that do this – Lagrange, Aitken-Neville, and Hermite – but we'll utilize *Newton's divided differences interpolation*, which is both simple mathematically and has a straightforward, fast implementation.

However, all of these approaches are prone to becoming numerically unstable when applied to more data, with the resulting polynomial oscillating wildly to accommodate every point. One solution is to switch to approximation – fit a lower-order polynomial to the data which doesn't pass through all of the supplied points but still offers a satisfactory close fit. We've already used the most common technique of this type, *least-squares*, back in Section 6.8, so we'll only briefly discuss it here.

Another choice is to switch to piece-wise polynomials of small degree, such as *cubic splines*, which fit multiple curve segments through point intervals while ensuring that the curves run together smoothly to give the appearance of a single equation.

A radically different approach is treat most of the data as *control points* which affect the curve's curvature. This lends itself to an interactive definition of the curve by moving the control points. We'll examine *Bézier curves* here, which offer a pleasing geometric and algebraic interpretation for how control points influence the curve.

Polynomial interpolation, least squares approximation, and cubic splines are usually discussed in textbooks on numerical analysis, such as Burden [**BFB16**] and Kiusalaas [**Kiu13**]. Bézier curves, and extensions such as B-splines and NURBS, are more likely to be described in analytic geometry textbooks, such as Marsh [**Mar05**] and Mortenson [**Mor99**].

15.2 Newton Interpolation

Newton's method manipulates a polynomial of the form

$$P_n(x) = a_0 + (x-x_0)a_1 + (x-x_0)(x-x_1)a_2 + \cdots + (x-x_0)(x-x_1)\cdots(x-x_{n-1})a_n$$

For example, four data points will be modeled by

$$\begin{aligned} P_3(x) &= a_0 + (x-x_0)a_1 + (x-x_0)(x-x_1)a_2 + (x-x_0)(x-x_1)(x-x_2)a_3 \\ &= a_0 + (x-x_0)\{a_1 + (x-x_1)[a_2 + (x-x_2)a_3]\} \end{aligned}$$

There's a natural recursive way to evaluate $P_n(x)$ for arbitrary n:

$$P_0(x) = a_n \quad P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, ..., n$$

but, in practice, it's more efficient to implement the nested multiplications using a loop as in evalPoly() in newtonInterp.py:

```
def evalPoly(a, xData, x):
  n = len(xData) - 1
  poly = a[n]
  for k in range(1, n + 1):
    poly = a[n - k] + (x - xData[n - k]) * poly
  return poly
```

P_n's coefficients are determined by making the polynomial pass through every data point: $y_i = P_n(x_i),\ i = 0, 1, \ldots, n$. This yields:

$$\begin{aligned} y_0 &= a_0 \\ y_1 &= a_0 + (x_1 - x_0)a_1 \\ y_2 &= a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2 \\ &\vdots \\ y_n &= a_0 + (x_n - x_0)a_1 + \cdots + (x_n - x_0)(x_n - x_1)\cdots(x_n - x_{n-1})a_n \end{aligned}$$

The values for a_i are obtained by calculating increasing levels of *divided difference* equations (∇^i):

$$\begin{aligned} \nabla y_i &= \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \ldots, n \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, 3, \ldots, n \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, 4, \ldots, n \\ &\vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}} \end{aligned}$$

After the level 1 equations (∇y_i) have been evaluated, their values are used to obtain the level 2 results ($\nabla^2 y_i$), and this process continues until a single level n equation is reached. This procedure can be represented by Table 15.1.

x_0	y_0				
x_1	y_1	∇y_1			
x_2	y_2	∇y_2	$\nabla^2 y_2$		
x_3	y_3	∇y_3	$\nabla^2 y_3$	$\nabla^3 y_3$	
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	$\nabla^4 y_4$

Table 15.1. Divided Difference Table for $n = 4$.

Once the table is completed, the a_i coefficients can be extracted from the main diagonal:

$$a_0 = y_0, \quad a_1 = \nabla y_1, \quad a_2 = \nabla^2 y_2, \ldots, \quad a_n = \nabla^n y_n$$

However, there's no need to maintain the entire table if each column is evaluated in a bottom-up order, since the new data can be stored by overwriting the old column, thereby allowing the use of a single list to hold all the values. This is implemented by coefs() in newtonInterp.py:

```
def coefs(xData, yData):
  m = len(xData)
  a = yData[:]  # make a copy
```

```
  for k in range(1, m):
    for j in range(m-1, k-1, -1):
      a[j] = (a[j] - a[j-1]) / (xData[j] - xData[j-k])
  return a
```

Initially, a contains the y-coordinates corresponding to the second column of Table 15.1. Each pass through the outer loop generates the entries in the next column, which are stored by overwriting a.

Fig. 15.1 shows newtonInterp.py's plot of the interpolated curve for the data:

```
xData = [0.15, 2.3, 3.15, 4.85, 6.25, 7.95]
yData = [4.79867, 4.49013, 4.2243, 3.47313, 2.66674, 1.51909]
```

against the actual curve $y = 4.8\cos(\pi x/20)$.

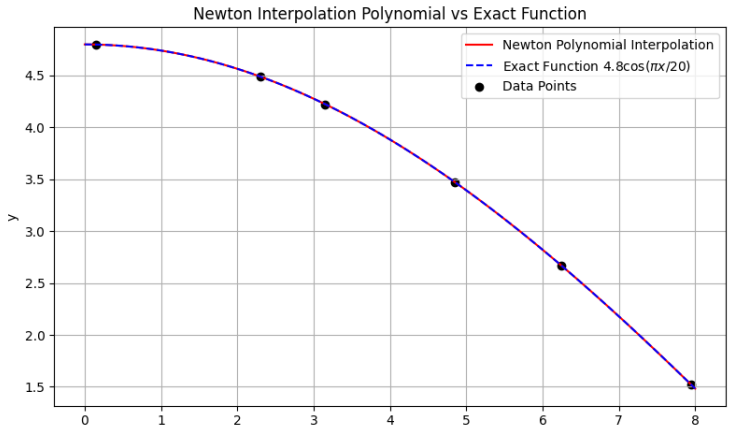


Figure 15.1. Newton Interpolation

The polynomial equation is:

```
P(x) = 4.79867 - 0.144(x-0.15) - 0.056(x-0.15)(x-2.30) +
          0.001(x-0.15)(x-2.30)(x-3.15) +
          0.000(x-0.15)(x-2.30)(x-3.15)(x-4.85) -
          0.000(x-0.15)(x-2.30)(x-3.15)(x-4.85)(x-6.25)
```

which illustrates a drawback of this approach – the equation isn't in a standard power form (e.g. $c_0 + c_1x + c_2x^2 + ...$). Also note that in this case a_4 and a_5 are 0 to 3 d.p., so the curve could be simplified.

15.2.1 The Runge Phenomenon. The Runge phenomenon arises when a curve is approximated by a high-order polynomial that passes through many points, especially points that are evenly spaced out. The polynomial will tend to increasingly oscillate as the number of coordinates grows, as illustrated in Fig.

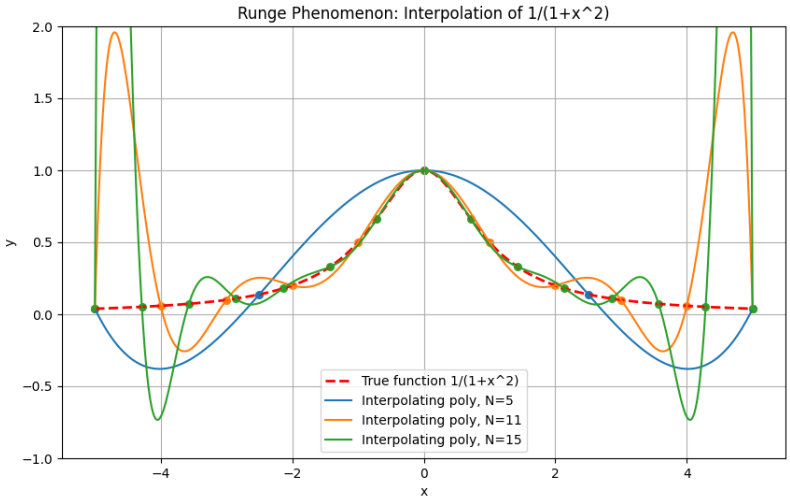


Figure 15.2. The Runge Phenomenon

15.2 which plots the equation $f(x) = 1/(1+x^2)$ against polynomial interpolations using 5, 11, and 15 equally spaced points.

The code that generates Fig. 15.2 is in `runge.py`, and uses the same Newton interpolation as `newtonInterp.py`.

One solution is to vary the distribution of the sampled coordinates so they're more densely concentrated at the edges of the interpolation interval. However, in the next two sections we'll employ least squares fitting and cubic splines instead. The former gives us the freedom to choose lower-degree polynomial approximations to the data, while the latter fits multiple small order polynomials over sub-intervals of the data.

15.3 Least Squares Curve Fitting

Least Squares curve fitting was the subject of section 6.8, so won't be discussed again in much depth. One implementation issue is that the code in that section applied Gaussian elimination to row lists (see `quadReg.py`) whereas we'll use the matrix module, `Mat.py`, from Appendix I here. For example, the coefficients for the quadratic polynomial $a_0 + a_1x + a_2x^2$) approximating n data points will be represented by $\mathbf{Aa} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix}$$

and

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \end{bmatrix}$$

polyLS.py uses the supplied n data points to create matrices for **A** and **b**, and then multiplies the inversion of **A** to **b** to find **a** (which is called `X` in the code snippet below):

```
# Data
xs = [0.1, 0.4, 0.5, 0.7, 0.7, 0.9]
ys = [0.61, 0.92, 0.99, 1.52, 1.47, 2.03]
nPts = len(xs)
order = int(input(f"Order of polynomial (n < {nPts-1}): "))
m = order + 1

A, B = createMats(xs, ys, m)
# Solve system of linear equations A*X = B
try:
  AInv = A.inverse()
  X = AInv * B
  coefs = X.transpose().toList()[0]
except ValueError as e:
  print(f"Error: {e}")
  sys.exit(1)
xsFit, ysFit = fitCurve(xs, ys, coefs)
```

Note that the code asks the user to supply a polynomial order which determines the number of simultaneous equations to solve. The program plots the resulting polynomial against the original data and reports the goodness of fit.

Fig. 15.3 shows four separate runs of polyLS.py with polynomial orders from 1 (linear) to 4.

Naturally, the goodness of fit improves as the order increases, but this measure doesn't reflect our desire for curve simplicity. Probably, the best approximation is of order two.

15.4 Cubic Spline Interpolation

Spline interpolation divides the data range into intervals and constructs a different polynomial for each piece. The simplest type is piece-wise linear interpolation, which joins the data points with straight lines, with the obvious drawback of a lack of "smoothness" between the pieces. We really want the two curve segments that meet at a common point to have the same slope (i.e. the same tangent) and curvature (the same rate of change for their tangents). These smoothness requirements can be defined geometrically for a given function:

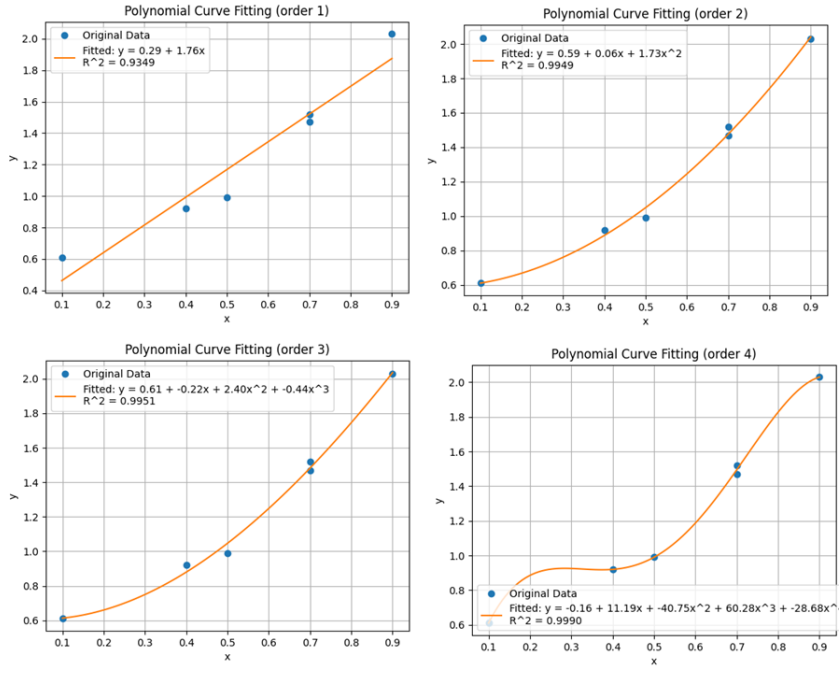


Figure 15.3. Polynomial Curve Fittings with Different Orders

- C^0: the function is continuous;
- C^1: the function is continuously differentiable (i.e. it's first derivative is continuous);
- C^2: the function has a continuous second derivative.

The simplest piece-wise approximation that satisfies these properties is the cubic spline, and Fig. 15.4 shows how these requirements are expressed. The spline S is made up of a series of cubic polynomials, denoted $S_i(x)$, on the intervals $[x_i, x_{i+1}]$ for $i = 0, 1, \ldots, n-1$.

We'll assume that the end points at x_0 and x_n have no curvature (i.e. $S''(x_0) = S''(x_n) = 0$. This creates a *natural* spline, so named because it mimics the natural shape a flexible strip assumes if forced to pass through specified points (as in Fig. 15.5)

The polynomial for each interval $[x_i, x_{i+1}]$ is written as:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3. \tag{15.1}$$

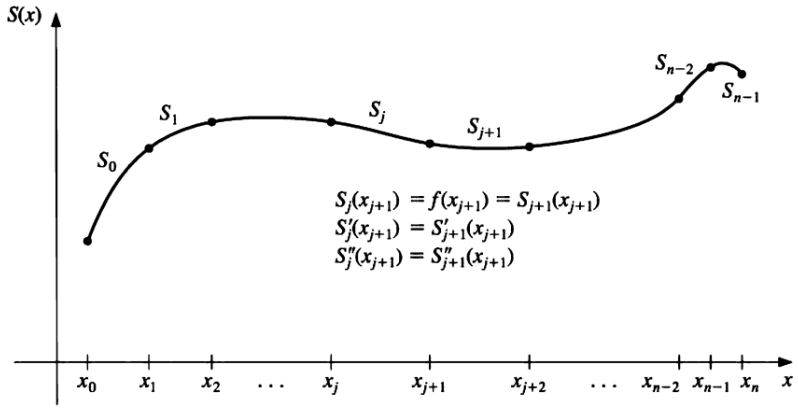


Figure 15.4. Smoothness Requirements for a Cubic Spline

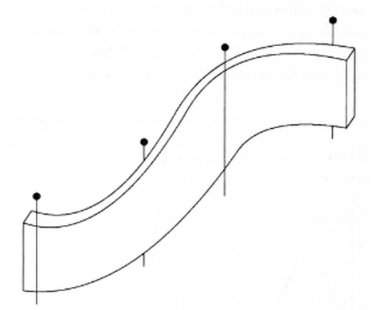


Figure 15.5. A Real-world Spline

Note that $S_i(x_i) = a_i = y_i$, so the unknown coefficients are b_i, c_i, and d_i. Our goal is to obtain expressions for b_i and d_i in terms of c_i, and to formulate c_i as simultaneous linear equations that can be solved.

We start by formalizing the smoothness requirements for each $S_i(x)$. Each interior point on the spline, x_i $(1 \leq i \leq n-1)$, exhibits:

- First derivative continuity (i.e. slope equality):

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}). \tag{15.2}$$

- Second derivative continuity (i.e. slope change (curvature) equality):

$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}). \tag{15.3}$$

Also, the natural spline boundary conditions mean that:

$$S''(x_0) = 0, \quad S''(x_n) = 0. \tag{15.4}$$

The derivatives of $S_i()$ are:

$$S_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2, \tag{15.5}$$
$$S_i''(x) = 2c_i + 6d_i(x - x_i). \tag{15.6}$$

Let $h_i = x_{i+1} - x_i$, and then at x_{i+1}:

$$S_i(x_{i+1}) = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3, \tag{15.7}$$
$$S_i'(x_{i+1}) = b_i + 2c_i h_i + 3d_i h_i^2, \tag{15.8}$$
$$S_i''(x_{i+1}) = 2c_i + 6d_i h_i. \tag{15.9}$$

Apply Equation (15.5) at x_{i+1}:

$$S_{i+1}'(x_{i+1}) = b_{i+1}$$

Substitute this equation and Equation (15.2) into Equation (15.8):

$$b_{i+1} = b_i + 2c_i h_i + 3d_i h_i^2 \tag{15.10}$$

Apply Equation (15.6) at x_{i+1}:

$$S_{i+1}''(x_{i+1}) = 2c_{i+1}$$

Substitute this equation and Equation (15.3) into Equation (15.9):

$$2c_{i+1} = 2c_i + 6d_i h_i$$
$$c_{i+1} = c_i + 3d_i h_i$$

Rearranging:

$$d_i = (c_{i+1} - c_i)/3h_i \tag{15.11}$$

We'll use this equation later to obtain d_i values once we've calculated c_i's.

Substitute $S_i(x_{i+1}) = a_{i+1}$ into Equation (15.7):

$$a_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 \tag{15.12}$$

Substitute Equation (15.11) into Equation (15.12):

$$a_{i+1} = a_i + b_i h_i + \frac{h_i^2}{3}(2c_i + c_{i+1})$$

Rearrange for b_i:

$$b_i = \frac{1}{h_i}(a_{i+1} - a_i) - \frac{h_i}{3}(2c_i + c_{i+1}) \tag{15.13}$$

This equation will be used later to obtain values for b_i once we have the c_i's.

Substitute Equation (15.11) into Equation (15.10):

$$\begin{aligned} b_{i+1} &= b_i + 2c_i h_i + 3(\frac{c_{i+1} - c_i}{3h_i})h_i^2 \\ &= b_i + h_i(c_i + c_{i+1}) \end{aligned}$$

So

$$h_i(c_i + c_{i+1}) = b_{i+1} - b_i$$

Reducing the subscripts:

$$h_{i-1}(c_{i-1} + c_i) = b_i - b_{i-1}$$

Replace the right hand side of this equation by the right hand sides of Equation (15.13) for b_i and b_{i-1}:

$$h_{i-1}(c_{i-1}+c_i) = \frac{1}{h_i}(a_{i+1}-a_i) - \frac{1}{h_{i-1}}(a_i - a_{i-1}) - \frac{h_i}{3}(2c_i + 2c_{i+1}) - \frac{h_{i-1}}{3}(2c_{i-1} + c_i)$$

Simplify:

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1}) \quad (15.14)$$

Note that this equation only applies in the range $i = 1, 2, \ldots n-1$ due to its use of c_{i-1} and c_{i+1}, so we must determine c_0 and c_n separately. This is possible by using the boundary conditions in Equation (15.4) and Equation (15.6):

$$\begin{aligned} S''(x_0) &= 0 \\ S_i''(x) &= 2c_i + 6d_i(x - x_i). \end{aligned}$$

So

$$S_0''(x_0) = 0 = 2c_0 + 6d_0(x_0 - x_0)$$

and so $c_0 = 0$. Similarly:

$$S''(x_n) = 0.$$

So

$$S_n''(x_n) = 0 = 2c_n + 6d_n(x_n - x_n)$$

and so $c_n = 0$.

$c_0 = 0$, $c_n = 0$ and Equation (15.14) specify a linear system described by $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a $(n+1) \times (n+1)$ matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \\ h_0 & 2(h_0 + h_1) & h_1 & & 0 & \\ 0 & h_1 & 2(h_1 + h_2) & h_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & \cdots & 0 & 0 & 1 & \end{bmatrix},$$

and $\mathbf{x}$ and $\mathbf{b}$ are the vectors:

$$\mathbf{x} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{bmatrix}.$$

The matrix $\mathbf{A}$ is diagonally dominant; that is, in each row, the magnitude of the diagonal entry exceeds the sum of the magnitudes of all the other entries in the row. Such a *tridiagonal* matrix can be solved using the Thomas algorithm (`https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm`), which produces a result in $O(n)$ time rather than the $O(n^3)$ required by Gaussian elimination. However, we'll stick with the Gaussian so we can use our `Mat.py` class (see Appendix I), and also because the number of data points (n) is small. An optimization we will take is to note that there's no need to encode all of the matrices since we already have values for c_0 and c_n (both equal to 0). Its only necessary to consider the range $i = 1, 2, \ldots, n-1$, when building $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$.

`cubicSplineMat.py` fits a spline to the data points represented by the lists:

```
xs = [0, 1, 2, 3, 4, 5]
ys = [0, 0.5, 2, 1.5, 0.5, 0]
```

It produces the graph shown in Fig. 15.6.

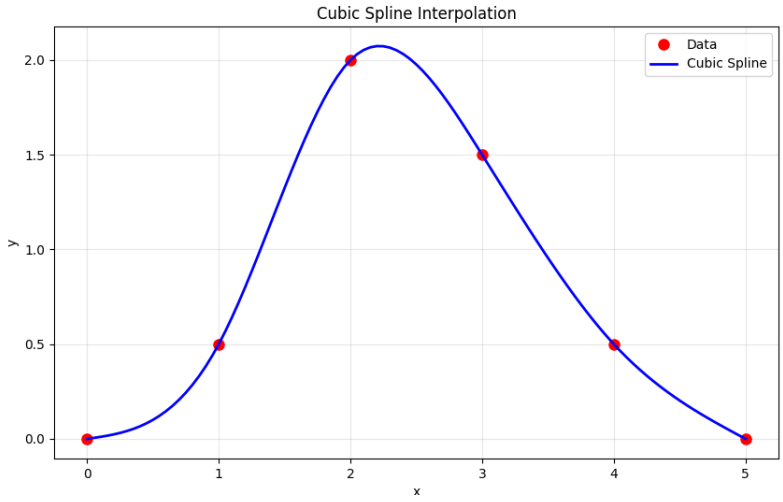


Figure 15.6. A Cubic Spline and Data

The coefficients (stored in the lists `ais`, `bs`, `cs`, and `ds`) are calculated by cubicSpline():

```
def cubicSpline(xs, ys):
  n = len(xs)
  if n < 3:
    raise ValueError("Need at least 3 points for cubic spline")

  ais = ys.copy()
  bs = [0]*n
  cs = [0]*n
  ds = [0]*n

  # Compute spacing
  hs = [xs[i+1]-xs[i] for i in range(n-1)]

  # Create matrices and solve the system
  A, b = buildMats(ais, hs, n)
  try:
    cSoln = A.inverse() * b
    cs[0] = 0
    for i in range(cSoln.nRows):
      cs[i+1] = cSoln[i][0]
    cs[n-1] = 0
  except ValueError as e:
    raise ValueError(f"Failed to solve spline system: {e}")

  # Compute other coefs using the cs values
  for j in range(n-1):
    bs[j] = (ais[j+1] - ais[j]) / hs[j] - \
            hs[j] * (cs[j+1] + 2 * cs[j]) / 3
    ds[j] = (cs[j+1] - cs[j]) / (3 * hs[j])

  return ais, bs, cs, ds
```

`ais` and `hs` are passed to `buildMat()` to build the reduced size **A** and **b** matrices. **A** is inverted using the `Mat.inverse()` implementation of Gaussian elimination, and the `cs` coefficients are read from `A.inverse() * b`. Note that these values are only for the range $i = 1, 2, \ldots, n-1$. The `bs` and `ds` values are obtained via implementations of Equation (15.13) and (15.11).

`buildMats()` creates a tridiagonal matrix for **A** and the **b** right-hand side, but restricted to the range $i = 1, 2, \ldots, n-1$. The function also prints the matrices as a useful debugging check. For the supplied input, the output is:

```
A =                                                    b=
 4.00    1.00    0.00    0.00                           3.00
 1.00    4.00    1.00    0.00                          -6.00
```

```
 0.00     1.00     4.00     1.00                    -1.50
 0.00     0.00     1.00     4.00                     1.50
```

Note that since six data points were supplied, then **A** is 4x4 and **b** is a 4-element vector.

The program finishes by printing out the coefficients used in $S_0, S_1, \ldots S_4$:

```
Interval [0, 1]: a=0.0000, b=0.1005, c=0.0000, d=0.3995
Interval [1, 2]: a=0.5000, b=1.2990, c=1.1986, d=-0.9976
Interval [2, 3]: a=2.0000, b=0.7033, c=-1.7943, d=0.5909
Interval [3, 4]: a=1.5000, b=-1.1124, c=-0.0215, d=0.1340
Interval [4, 5]: a=0.5000, b=-0.7536, c=0.3804, d=-0.1268
```

This makes it quite clear that the spline is a piece-wise curve made up of five cubic polynomials.

15.5 The Bézier Curve

Cubic splines and Bézier curves are both used for modeling smooth curves, but while a spline passes through all the data points, a Bézier curve treats all but the first and last coordinates as *control* points which influence the curve's curvature. The result is a single polynomial rather than the piece-wise polynomials that make up a spline. A cubic Bézier (degree 3) requires two control points, a quadratic Bézier (degree 2) requires just one, but a curve may utilize any number of control points if necessary.

A Bézier curve can be contructed geometrically with the de Casteljau algorithm, or algebraically as a combination of Bernstein polynomials. We'll consider each in turn.

15.5.1 A Geometric Construction. A second-degree Bézier curve can be built with the de Casteljau's algorithm in the manner illustrated by Fig. 15.7.

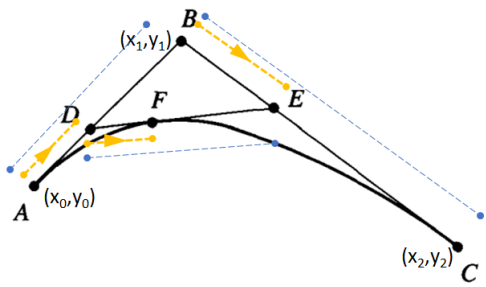


Figure 15.7. Geometric construction of a second-degree Bézier curve

The points A, B, C are selected so that the line AB is tangent to the curve at A, and BC is tangent to C. For some ratio u, where $0 \leq u \leq 1$, the points D and E are constructed so that

$$\frac{AD}{AB} = \frac{BE}{BC} = u$$

Crucially, F also utilizes the same ratio on DE so that $DF/DE = u$.

The curve is defined by how F moves from D to E while D and E are themselves moving (D from A to B, and E from B to C). This effect is best seen by running `animInterp.py` which animates the F point.

We can express this construction in terms of vectors with the following substitutions: Let the vector $\mathbf{a}$ represent point A, use $\mathbf{b}$ for point B, and $\mathbf{c}$ for C. The vectors $\mathbf{d}$ and $\mathbf{e}$ for D and E are:

$$\mathbf{d} = \mathbf{a} + u(\mathbf{b} - \mathbf{a})$$
$$\mathbf{e} = \mathbf{d} + u(\mathbf{c} - \mathbf{b})$$

which let's us write $\mathbf{f}$ for F as:

$$\mathbf{f} = \mathbf{d} + u(\mathbf{e} - \mathbf{d})$$

The equations for $\mathbf{d}$ and $\mathbf{e}$ can be substituted into this equation to give:

$$\mathbf{f} = (1 - u^2)\mathbf{a} + 2u(1 - u)\mathbf{b} + u^2\mathbf{c}$$

We generalize this equation for any point on the curve by parameterizing the point in terms of u:

$$\mathbf{p}(u) = (1 - u^2)\mathbf{a} + 2u(1 - u)\mathbf{b} + u^2\mathbf{c}$$

This second-degree equation utilizes $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ as control points, and the ratio u as the parametric variable. Similar constructions are employed for Bézier curves of any degree.

Fig. 15.8 shows the construction of a cubic Bézier curve, which requires four control points A, B, C, and D. The curve begins at A as a tangent to line AB, and ends at D, as a tangent to CD.

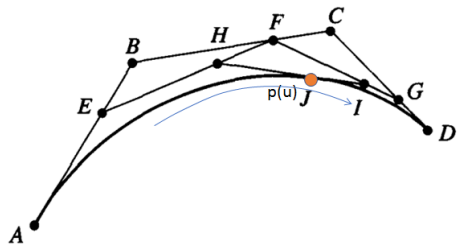


Figure 15.8. Geometric construction of a Cubic Bézier curve

We construct E, F, and G so that

$$\frac{AE}{AB} = \frac{BF}{BC} = \frac{CG}{CD} = u$$

On EF and FG, H and I are located at

$$\frac{EH}{EF} = \frac{FI}{FG} = u$$

Finally, on HI, J is positioned so that

$$\frac{HJ}{HI} = u$$

No further subdivisions are possible, which means that J moves along the curve.

If A, B, C, and D are represented by the vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$, then the intermediate points E, F, G, H, and I can be expressed as:

$$\mathbf{e} = \mathbf{a} + u(\mathbf{b} - \mathbf{a}) \quad \mathbf{f} = \mathbf{b} + u(\mathbf{c} - \mathbf{b}) \quad \mathbf{g} = \mathbf{d} + u(\mathbf{d} - \mathbf{c})$$
$$\mathbf{h} = \mathbf{e} + u(\mathbf{f} - \mathbf{e}) \quad \mathbf{i} = \mathbf{f} + u(\mathbf{g} - \mathbf{f})$$

J is then:

$$\mathbf{j} = \mathbf{h} + u(\mathbf{i} - \mathbf{h})$$

This expression can be expanded:

$$\begin{aligned}
\mathbf{p}(u) &= \mathbf{h} + u(\mathbf{i} - \mathbf{h}) \\
&= \mathbf{e} + u(\mathbf{f} - \mathbf{e}) + u(\mathbf{f} + u(\mathbf{g} - \mathbf{f}) - (\mathbf{e} + u(\mathbf{f} - \mathbf{e}))) \\
&= \mathbf{a} + u(\mathbf{b} - \mathbf{a}) + u(\mathbf{b} + u(\mathbf{c} - \mathbf{b}) - (\mathbf{a} + u(\mathbf{b} - \mathbf{a}))) + \\
&\quad u(\mathbf{b} + u(\mathbf{c} - \mathbf{b}) + u(\mathbf{d} + u(\mathbf{d} - \mathbf{c}) - (\mathbf{b} + u(\mathbf{c} - \mathbf{b})))) - \\
&\quad (\mathbf{a} + u(\mathbf{b} - \mathbf{a}) + u(\mathbf{b} + u(\mathbf{c} - \mathbf{b}) - (\mathbf{a} + u(\mathbf{b} - \mathbf{a}))))
\end{aligned}$$

This (thankfully) simplifies to:

$$\mathbf{p}(u) = (1-u)^3\mathbf{a} + 3u(1-u)^2\mathbf{b} + 3u^2(1-u)\mathbf{c} + u^3\mathbf{d},$$

a cubic curve with four control points.

15.5.2 Implementing De Casteljau's Algorithm.

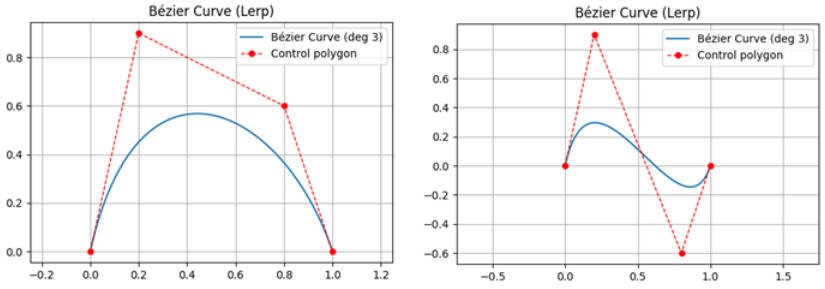
bezLerp.py plots a Bézier curve by passing a list of control points to `bezierCurveDC()` which implements de Casteljau's algorithm:

```
ctrlPts = [(0,0),(0.2,0.9),(0.8,0.6),(1,0)]
curveX, curveY = bezUtils.bezierCurveDC(ctrlPts, 200)
```

The graph is shown in Fig. 15.9(a). If the third control point is changed to `(0.8,-0.6)` then the plot on the right is produced.

`bezierCurveDC()` in bezUtils.py:

```
def bezierCurveDC(ctrlPts, nSamps):
  xs = []   # points making up the curve
  ys = []
  if nSamps < 2:  # no. of samples
```



(a) Third Control Point: (0.8, 0.6) (b) Third Control Point: (0.8, -0.6)

Figure 15.9. Using De Casteljau to Draw Bézier Curves

```
    nSamps = 2
  for s in range(nSamps+1):
    u = s / nSamps
    qx, qy = bezLerp(ctrlPts, u)
    xs.append(qx)  # one point for each sample
    ys.append(qy)
  return xs, ys
```

The real work is done by `bezLerp()` which returns a curve coordinate for a given u parameter.

```
def bezLerp(ctrlPts, u):
  pts = list(ctrlPts)  # make a working copy
  n = len(pts)
  for r in range(1, n):   # r = interpolation depth
    for i in range(n - r):
      x = lerp(pts[i][0], pts[i+1][0], u)
      y = lerp(pts[i][1], pts[i+1][1], u)
      pts[i] = (x, y)
  return pts[0]

def lerp(a, b, u):  # Linear interpolation
  return a + (b-a)*u
```

To understand `bezLerp()` it helps to manually execute it on a small example, such as the curve in Fig. 15.7. In that case, three control points (A, B, C) are passed to the function along with u.

The `r` loop will execute twice: once with $r = 1$ and then with $r = 2$. For $r = 1$, the i loop will range over 0 and 1 which will interpolate the line AB and then BC, which 'moves' D and E (see Fig. 15.7). Note that these points are written over the existing data in `pts` before the next iteration of r begins.

When $r = 2$, the inner loop will only consider 0 which means that it interpolates the line DE (since D and E are in `pts`). This is equivalent to calculating the position of point F (see Fig. 15.7), the current coordinate of the curve. F is stored in `pts`, and returned when `bezLerp()` finishes.

15.5.3 An Algebraic Definition. Any point $\mathbf{p}(u)$ on a Bézier curve can be written as:

$$\mathbf{p}(u) = \mathbf{p_0} f_0(u) + \mathbf{p_1} f_1(u) + ... + \mathbf{p_n} f_n(u)$$

where $0 \leq u \leq 1$, the vectors $\mathbf{p_i}$ are the control points, and the $f_i(u)$ functions must produce a curve that has certain characteristics:

(1) The curve must start at the first control point, $\mathbf{p_0}$, and end at the last, $\mathbf{p_n}$.

(2) The curve must start as a tangent to the line $\mathbf{p_1} - \mathbf{p_0}$ at $\mathbf{p_0}$ and end as a tangent to $\mathbf{p_n} - \mathbf{p_{n-1}}$ at $\mathbf{p_n}$.

(3) The functions $f_i(u)$ must be symmetric with respect to u and $(1 - u)$. This lets us reverse the sequence of control points without changing the shape of the curve.

A family of functions called *Bernstein* polynomials satisfies these requirements, and $\mathbf{p}(u)$ can be rewritten using them:

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p_i} B_{i,n}(u) \tag{15.15}$$

where

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i} \tag{15.16}$$

and

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

A second-degree Bézier curve (where $n = 2$ and there are three control points) is obtained from Equation (15.15) as:

$$\mathbf{p}(u) = \mathbf{p_0} B_{0,2}(u) + \mathbf{p_1} B_{1,2}(u) + \mathbf{p_2} B_{2,2}(u)$$

From Equation (15.16) we find

$$B_{0,2}(u) = (1-u)^2$$
$$B_{1,2}(u) = 2u(1-u)$$
$$B_{2,2}(u) = u^2$$

Substituting them into $\mathbf{p}(u)$:

$$\mathbf{p}(u) = (1-u^2)\mathbf{p_0} + 2u(1-u)\mathbf{p_1} + u^2\mathbf{p_2}$$

This is the same as the expression we obtained from the geometric construction, albeit with **a**, **b**, and **c** replaced by the corresponding $\mathbf{p_i}$'s.

Now, let's expand Equation (15.15) for a cubic curve, where $n = 3$:

$$\mathbf{p}(u) = \mathbf{p_0}B_{0,3}(u) + \mathbf{p_1}B_{1,3}(u) + \mathbf{p_2}B_{2,3}(u) + \mathbf{p_3}B_{3,3}(u)$$

From Equation (15.16) we find

$$\begin{aligned} B_{0,3}(u) &= (1-u)^3 \\ B_{1,3}(u) &= 3u(1-u)^2 \\ B_{2,3}(u) &= 3u^2(1-u) \\ B_{3,3}(u) &= u^3 \end{aligned}$$

Substituting these into $\mathbf{p}(u)$:

$$\mathbf{p}(u) = (1-u)^3\mathbf{p_0} + 3u(1-u)^2\mathbf{p_1} + 3u^2(1-u)\mathbf{p_2} + u^3\mathbf{p_3}$$

bernsteins.py draws the Bernstein curves for a given Bézier curve degree order. For instance, Fig. 15.10 shows the four curves $(B_{0,3}(), B_{1,3}(), B_{2,3}(), B_{3,3}())$.

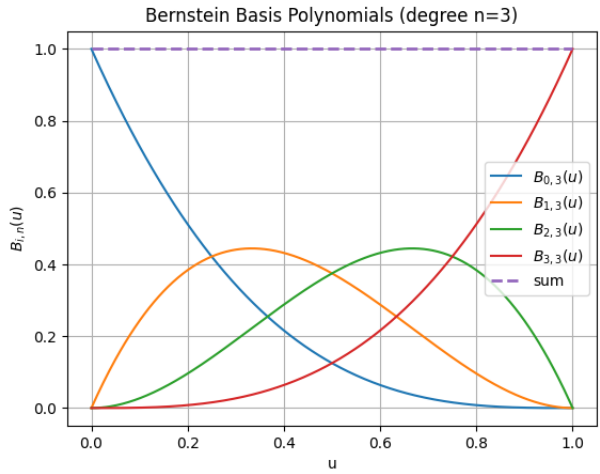


Figure 15.10. Bernstein Polynomials for a Cubic Bézier

The contribution of the first control point, $\mathbf{p_0}$, is propagated throughout the curve by $B_{0,3}()$ and is most influential at $u = 0$. Note that the other control points don't contribute to $\mathbf{p}(u)$ at $u = 0$ at all because $B_{1,3}(0) = B_{2,3}(0) = B_{3,3}(0) = 0$.

Control point $\mathbf{p_1}$ is most influential at $u = 1/3$, and $\mathbf{p_2}$ at $u = 2/3$. At $u = 1$, only $\mathbf{p_3}$ affects $\mathbf{p}(u)$. Also, notice the symmetry of $B_{0,1}(u)$ and $B_{3,3}(u)$, as well as that of $B_{1,3}(u)$ and $B_{2,3}(u)$.

Another useful way to get a feeling for how control points influence a curve is by moving the points interactively in dragPath.py. Three screenshots of the program running are shown in Fig. 15.11.

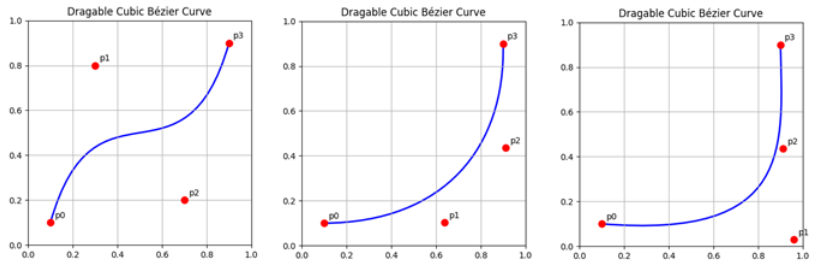


Figure 15.11. Moving Control Point P1

The images show the effect on a cubic curve of moving control point $P1$ from the top-left to the bottom-right of the plot. Indeed, any of the four points can be moved.

15.5.4 Implementing Bézier Using Bernstein Functions.

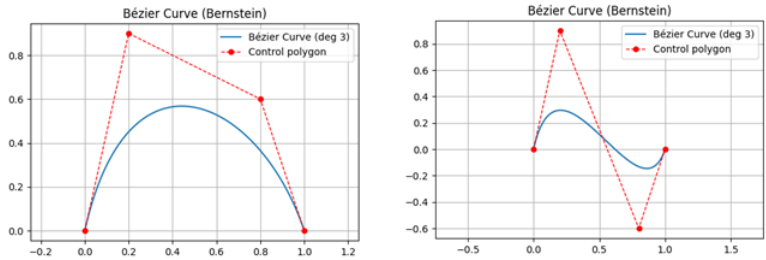
bezBern.py plot a Bézier curve by calling `bezierCurve()` with a list of control points. The relevant code snippet in `bezBern.py` is:

```
ctrlPts = [(0,0),(0.2,0.9),(0.8,0.6),(1,0)]
curveX, curveY = bezUtils.bezierCurve(ctrlPts, 200)
```

The graph is shown in Fig. 15.12(a). If the third point is changed to `(0.8,-0.6)` then the plot on the right (b) is produced. As we'd expect, Fig. 15.12 is identical to Fig. 15.9.

Much of `bezierCurve()` in bezUtils.py is similar to `bezierCurveDC()`:

```
def bezierCurve(ctrlPts, nSamps=100):
  n = len(ctrlPts) - 1
  xs = []
  ys = []
  if nSamps < 2:
    nSamps = 2
  for step in range(nSamps + 1):
    u = step / nSamps
    qx, qy = 0, 0
    for i in range(n+1):
      b = bernstein(n, i, u)
      qx += b * ctrlPts[i][0]  # x
      qy += b * ctrlPts[i][1]  # y
```



(a) Third Control Point: (0.8, 0.6) (b) Third Control Point: (0.8, -0.6)

Figure 15.12. Using Bernstein Functions to Draw Bézier Curves

```
    xs.append(qx)
    ys.append(qy)
  return xs, ys

def bernstein(n, i, u):
  # Bernstein polynomial B_{i,n}(u)
  return math.comb(n, i) * (u**i) * ((1 - u)**(n - i))
```

The important differences lie inside the i loop, which calls `bernstein(n, i, u)` to obtain a particular $B_{i,n}(u)$ value. `bernstein()` is a direct translation of Equation (15.16).

15.5.5 Bézier 3D curves. Although we've been concentrating on 2D curves, it's not much work to extend the ideas to 3D. `bezier3D()` in bezUtils.py is virtually identical to `bezierCurve()` given above, but accepts control points of the form (x, y, z). The most important change this causes is the need to use `bernstein()`'s result *three* times rather than twice inside the i loop:

```
    # inside bezier3D()
    for i in range(n+1):
      b = bernstein(n, i, u)
      qx += b * ctrlPts[i][0]   # x
      qy += b * ctrlPts[i][1]   # y
      qz += b * ctrlPts[i][2]   # z
```

bez3D.py illustrates how to use `bezier3D()`:

```
ctrlPts = [ [0,0,0], [1,5,2],
            [5,1,8], [6,6,0] ]
curveX, curveY, curveZ = bezUtils.bezier3D(ctrlPts)
```

It generates the plot shown in Fig. 15.13.

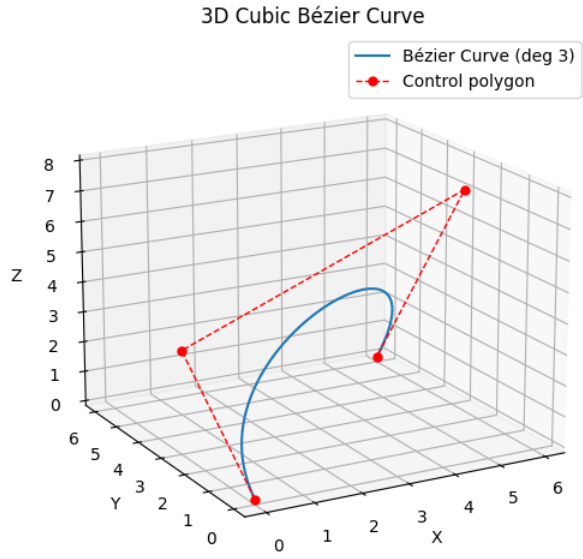


Figure 15.13. Using Bernstein Functions to Draw a 3D Bézier Curve

15.5.6 Bézier curves in Matplotlib. The Matplotlib `Path` class supports a set of `moveto`, `lineto`, and `curveto` commands for drawing line segments and curves. Most relevant for this section are the commands `CURVE3` and `CURVE4` for drawing a quadratic Bézier curve using three control points and a cubic curve with four points (`https://matplotlib.org/stable/users/explain/artists/paths.html`). Sadly, it's not currently possibly to add move control points to the curve, although it is possible to stitch together multiple quadratic and/or cubic curves using other `Path` commands.

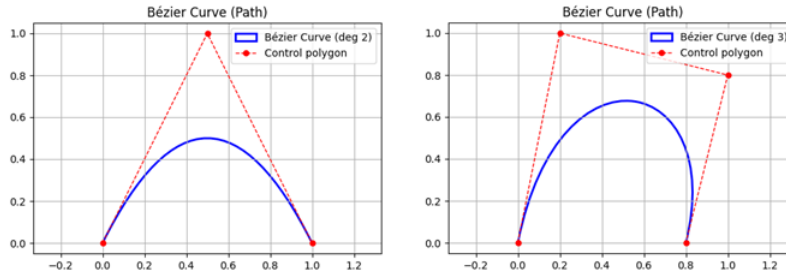
bezPath.py builds a Matplotlib patch using `Path` by calling `bezPath()`:

```
# control points (choose one of these)
# ctrlPts = [(0,0),(0.5,1),(1,0)]
ctrlPts = [(0,0),(0.2,1),(1,0.8),(0.8,0)]

bezierPath = bezUtils.bezPath(ctrlPts)
patch = PathPatch(bezierPath,
          facecolor="none", edgecolor="blue", lw=2,
          label=f"Bézier Curve (deg {len(ctrlPts)-1})")
```

Fig. 15.14 shows the two possible plots depending on which version of `ctrlPts` is uncommented.

`bezPath()` in bezUtils.py decides which `Path` commands to string together depending on the number of control points:



(a) Three Control Points (CURVE3) (b) Four Control Points (CURVE4)

Figure 15.14. Bézier Curves Using Matplotlib Paths

```
def bezPath(ctrlPts):
  n = len(ctrlPts)
  codes = []
  if n == 3:   # CURVE3 is for a quadratic B\'{e}zier
    codes = [Path.MOVETO, Path.CURVE3, Path.CURVE3]
  elif n == 4: # CURVE4 is for a cubic B\'{e}zier
    codes = [Path.MOVETO, Path.CURVE4,
             Path.CURVE4, Path.CURVE4]
  else:
    raise ValueError("Path lengths must == 3 or 4")
  return Path(ctrlPts, codes)
```

15.5.7 Rational Bézier Curves. We represent a rational Bézier curve in vector form as

$$\mathbf{p}(u) = \frac{\sum h_i \mathbf{p_i} B_{i,n}(u)}{\sum h_i B_{i,n}(u)}$$

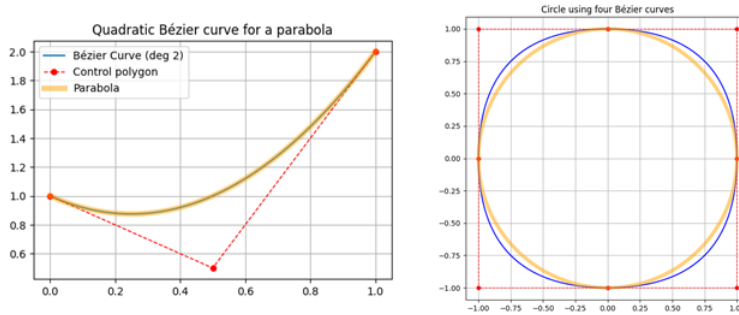
where the $\mathbf{p_i}$s are the control points and the h_is are weights. Note that this equation reduces to the previous definition for Bézier curves when all $h_i = 1$ (we'll call these *integral* Bézier curves from here on). Weights allow a much wider range of curves to be modeled, with one important class being conics; in particular, the circle, ellipse, and hyperbola. The parabola doesn't need the use of weights.

The two programs, parabola.py and circle.py, use integral (i.e. non-weighted) Bézier curves to mimic a parabola and a circle. The results are shown in Fig. 15.15.

In parabola.py, the parabolic function is defined as:

```
a, b, c = 2, -1, 1

def parabola(x):
  return a*x**2 + b*x + c
```



(a) parabola.py (b) circle.py

Figure 15.15. Integral Bézier Curves for a Parabola and a Circle

The a, b, and c coefficients are also used to specify the curve's control points:

```
ctrlPts = [(0, c), (0.5, c/2), (1, a+b+c)]
```

The plot in Fig. 15.15(a) strongly suggests that the integral Bézier curve is an exact fit for the parabola, and it's not difficult to prove that the curve is a perfect match. A quadratic Bézier curve is:

$$\mathbf{p}(u) = (1-u)^2\mathbf{p_0} + 2(1-u)u\mathbf{p_1} + u^2\mathbf{p_2}$$

Consider a simple parabola, such as $y = x^2$, and three points on it:

$$\mathbf{p_0} = (0,0) \quad \mathbf{p_1} = (0.5, 0.25) \quad \mathbf{p_2} = (1,1)$$

Plug those into the x-axis and y-axis components of $\mathbf{p}(u)$:

$$\begin{aligned} x(u) &= (1-u)^2 \cdot 0 + 2(1-u)u \cdot 0.5 + u^2 \cdot 1 = u \\ y(u) &= (1-u)^2 \cdot 0 + 2(1-u)u \cdot 0.25 + u^2 \cdot 1 = u^2. \end{aligned}$$

This shows that the polynomial Bézier is equivalent to the parabolic equation.

Things are not so rosy for the curves mimicking a circle. Fig. 15.15(b) employs four curves to model the quadrants, and it's clear that the fit is rather poor, especially around the middle of each quadrant. The curves are implemented using four control point lists in `circle.py`:

```
cpArcs = [ # control points for each quarter arc
  [(1,0), (1,1), (0,1)],
  [(0,1), (-1,1), (-1,0)],
  [(-1,0), (-1,-1), (0,-1)],
  [(0,-1), (1,-1), (1,0)],
]
```

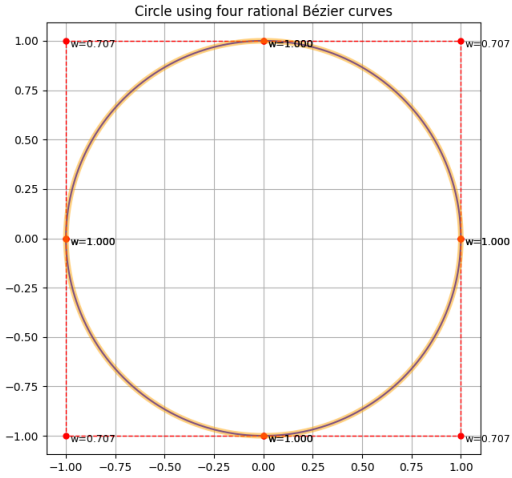


Figure 15.16. Rational Bézier Curve for a Circle

ratBez.py contains several rational Bézier curve examples, with the third one being the rational curve for a circle, shown in Fig. 15.16.

The relevant changes to the data from `circle.py` are:

```
weights = [1, math.sqrt(2)/2, 1]
cpArcs = [  # control points for each quarter arc
  [(1,0), (1,1), (0,1)],
  [(0,1), (-1,1), (-1,0)],
  [(-1,0), (-1,-1), (0,-1)],
  [(0,-1), (1,-1), (1,0)],
]

% some lines not shown...
curveX, curveY = bezUtils.ratBezier(ctrlPts, weights, nSamps)
```

The control point lists have *not* changed. What's new is a list of weights that reduces the emphasis of the middle control point of each list.

`ratBezier()` in bezUtils.py is very similar to `bezierCurve()` for integral curves.

```
def ratBezier(ctrlPts, weights, nSamps=100):
  if len(ctrlPts) != len(weights):
    raise ValueError("length error")
  n = len(ctrlPts) - 1
  xs = []; ys = []
  if nSamps < 2:
    nSamps = 2
```

```
for s in range(nSamps+1):
  u = s / nSamps
  denom = 0
  qx, qy = 0, 0
  for i in range(n+1):
    b = bernstein(n, i, u) * weights[i]
    denom += b
    qx += b * ctrlPts[i][0]
    qy += b * ctrlPts[i][1]
  xt = qx/denom
  yt = qy/denom
  xs.append(xt)
  ys.append(yt)
return xs, ys
```

The main alteration is inside the i loop where the weights are multiplied to each Bernstein function and the denominator is summed for use after the loop finishes.

A quadratic rational Bézier curve has the form

$$\mathbf{p}(u) = \frac{w_0\mathbf{p}_0 B_{0,2}(u) + w_1\mathbf{p}_1 B_{1,2}(u) + w_2\mathbf{p}_2 B_{2,2}(u)}{w_0 B_{0,2}(u) + w_1 B_{1,2}(u) + w_2 B_{2,2}(u)}.$$

where w_i are the weight associated with each control point.

Fig. 15.16 is a strong indicator that a circle can be exactly modeled by four rational Bézier curves, and it isn't too difficult to prove it. Consider the upper right hand quadrant where

$$\mathbf{p_0} = (1,0), \quad \mathbf{p_1} = (1,1), \quad \mathbf{p_2} = (0,1), \qquad w_0 = w_2 = 1,\ w_1 = \frac{\sqrt{2}}{2}.$$

The quadratic equation becomes

$$\mathbf{p}(u) = \frac{(1-u)^2 w_0\mathbf{p_0} + 2(1-u)uw_1\mathbf{p_1} + u^2 w_2\mathbf{p_2}}{(1-u)^2 w_0 + 2(1-u)uw_1 + u^2 w_2}.$$

Let's write $a = 1-u$ and $b = u$ for convenience (so $a+b=1$), and compute the numerator components and the denominator for a given point (x,y) on the curve. The numerator x-component:

$$N_x = a^2 \cdot 1 + 2ab \cdot w_1 \cdot 1 + b^2 \cdot 0 = a^2 + 2w_1\, ab.$$

The numerator y-component:

$$N_y = a^2 \cdot 0 + 2ab \cdot w_1 \cdot 1 + b^2 \cdot 1 = 2w_1\, ab + b^2.$$

The denominator:

$$D = a^2 + 2w_1\, ab + b^2.$$

So

$$x = \frac{N_x}{D} = \frac{a^2 + 2w_1ab}{D}, \qquad y = \frac{N_y}{D} = \frac{2w_1ab + b^2}{D}.$$

We want to show that $x^2 + y^2 = 1$, so compute the numerator of our $x^2 + y^2$:

$$N_x^2 + N_y^2 = (a^2 + 2w_1ab)^2 + (2w_1ab + b^2)^2.$$

If we let $A = a^2$, $B = b^2$, and $C = 2w_1ab$, then after several steps, we obtain

$$N_x^2 + N_y^2 - D^2 = (4w_1^2 - 2)AB.$$

Let's choose a value for w_1 so that $4w_1^2 - 2 = 0$, i.e.

$$w_1^2 = \frac{1}{2} \quad \Longrightarrow \quad w_1 = \frac{1}{\sqrt{2}} = \frac{\sqrt{2}}{2},$$

then $N_x^2 + N_y^2 - D^2 = 0$, and so $N_x^2 + N_y^2 = D^2$. Therefore $x^2 + y^2 = \frac{N_x^2 + N_y^2}{D^2} = 1$.

That proves that the rational quadratic Bézier with $\mathbf{p_0} = (1, 0)$, $\mathbf{p_1} = (1, 1)$, $\mathbf{p_2} = (0, 1)$ and weight $w_1 = \frac{\sqrt{2}}{2}$ lies exactly on the quarter arc from $(1, 0)$ to $(0, 1)$.

To model a particular conic, the weights of the curve must be as shown in Table 15.2.

ellipse	$w_1^2 - w_0w_2 < 0$
parabola	$w_1^2 - w_0w_2 = 0$
hyperbola	$w_1^2 - w_0w_2 > 0$

Table 15.2. Conic weights in a quadratic rational Bézier curve

The ability to represent a parabola using an integral Bézier curve is reflected by the fact that when $w_1^2 - w_0w_2 = 0$ then the weights can be eliminated from the rational equation.

In the circle example, we used $w_0 = w_2 = 1$ and $w_1 = \frac{\sqrt{2}}{2}$, so

$$w_1^2 - w_0w_2 = \frac{1}{2} - 1 = -\frac{1}{2} < 0,$$

so the curve is an ellipse (in fact, a circle).

15.5.8 Beyond Bézier Curves. Most textbooks that describe Bézier curves also contain chapters on two common extensions: B-splines (Basis splines) and NURBS (Non-Uniform Rational B-Splines).

B-splines add the concept of *knots* to smoothly attach multiple curves together. The utilize the same continuity ideas as in cubic splines – the use of tangent and curvature equality between the end point of one curve and the start point of the next.

NURBS add weights to B-splines in much the same way that we added them to Bézier curves in the last section.

Exercises

(1) For the given functions $f(x)$, let $x_0 = 0$, $x_1 = 0.6$, and $x_2 = 0.9$. Construct polynomials of degree one and two to approximate $f(0.45)$, and find the absolute errors.

(a) $f(x) = \cos x$
(b) $f(x) = \sqrt{1+x}$
(c) $f(x) = \ln(x+1)$
(d) $f(x) = \tan x$

(2) A census of the population of the United States is taken every 10 years. The following table lists the population, in thousands of people, from 1960 up to 2020.

Year	1960	1970	1980	1990	2000	2010	2020
Population (in thousands)	179,323	203,302	226,542	249,633	281,422	308,746	331,449

(a) Approximate the population in the years 1950, 1975, 2025, and 2030.
(b) The population in 1950 was approximately 150,697,360, and in 2025 the population was estimated to be 344,234,000. How accurate do you think your 1975 and 2030 figures are?

(3) Construct the natural cubic splines for the following data.

(a)

x	$f(x)$
−0.5	−0.0247500
−0.25	0.3349375
0	1.1010000

(b)

x	$f(x)$
0.1	−0.62049958
0.2	−0.28398668
0.3	0.00660095
0.4	0.24842440

(4) The data in the previous exercise was generated using the following functions. Calculate the errors due to the use of cubic splines.

(a) $f(x) = x^3 + 4.001x^2 + 4.002x + 1.101$; approximate $f(-1/3)$.
(b) $f(x) = x\cos x - 2x^2 + 3x - 1$; approximate $f(0.25)$.

(5) A natural cubic spline S on $[0, 2]$ is defined by

$$S(x) = \begin{cases} S_0(x) = 1 + 2x - x^3, & \text{if } 0 \le x < 1, \\ S_1(x) = 2 + b(x-1) + c(x-1)^2 + d(x-1)^3, & \text{if } 1 \le x \le 2. \end{cases}$$

Find b, c, and d.

(6) Construct enough points on the Bézier curve, whose control points are $p_0 = (4, 2)$, $p_1 = (8, 8)$, and $p_2 = (16, 4)$, to draw an accurate sketch.
(a) What degree is the curve?
(b) What are the coordinates at $u = 0.5$?

(7) A cubic Bézier curve has control points $(1, 0)$, $(3, 3)$, $(5, 5)$, and $(7, 2)$. Find the coordinate at $u = 0.25$ by applying the de Casteljau algorithm. Draw a sketch illustrating the curve and points.

(8) One property of Bernstein functions is that $\sum_i B_{i,n}(u) = 1$. (This property is displayed as a dashed line in Fig. 15.10.) Show that this property is true for $n = 2$.

(9) At what values of u do $B_{i,5}(u)$ reach a maximum?

(10) Prove that when the control points are colinear, that the resulting Bézier curve is a straight line segment.

(11) Fill in the gaps in the proof that the quadratic rational Bézier curve for a quadrant of a circle is a perfect match for that quadrant (see Section 15.5.7).

(12) Show that the equation for a quadratic rational Bézier curve for a parabola can be rewritten so that its weights w_i disappear, leaving an integral equation. *Hint*: set $a = \sqrt{w_0}$ and $b = \sqrt{w_2}$.

Answers

(1) (a) P1(x) = $-0.148878x + 1$; P2(x) = $-0.452592x^2 - 0.0131009x + 1$;
P1(0.45) = 0.933005; || f(0.45) - P1(0.45)|| = 0.032558;
P2(0.45) = 0.902455; || f(0.45) - P2(0.45) || = 0.002008
(b) P1(x) = $0.4672511x + 1$; P2(x) = $-0.0780026x^2 + 0.4906522x + 1$;
P1(0.45) = 1.210263; || f(0.45) - P1(0.45)|| = 0.006104;
P2(0.45) = 1.204998; || f(0.45) - P2(045) || = 0.000839

(c) P1(x) = 0.874548x; P2(x) = $-0.268961x^2 + 0.955236x$;
P1(0.45) = 0.393546; || f(0.45) - P1(0.45)|| = 0.0212983;
P2(0.45) = 0.375392; || f(0.45) - P2(0.45)|| = 0.003828

(d) P1(x) = 1.031121x; P2(x) = $0.615092x^2 + 0.846593x$;
P1(0.45) = 0.464004; || f(0.45) - P1(0.45)|| = 0.019051;
P2(0.45) = 0.505523; || f(0.45) - P2(0.45)|| = 0.022468

(3) (a)

i	a_i	b_i	c_i	d_i
0	−0.02475000	1.03237500	0.00000000	6.50200000
1	0.33493750	2.25150000	4.87650000	−6.50200000

(b)

i	a_i	b_i	c_i	d_i
0	−0.62049958	3.45508693	0.00000000	−8.9957933
1	−0.28398668	3.18521313	−2.69873800	−0.94630333
2	0.00660095	2.61707643	−2.98262900	9.9420966

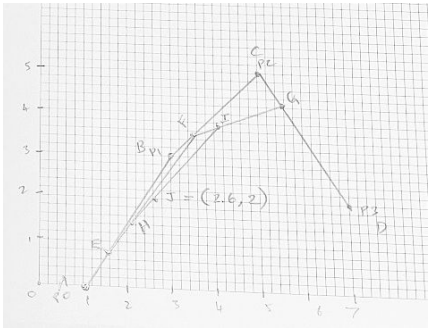
(4) x; Approx to $f(x)$; Actual $f(x)$; Error

(a) -1/3; 0.1774144; 0.17451852; 2.8959×10^{-3}

(b) 0.25; -0.1315912; -0.13277189; 1.1807×10^{-3}

(5) $b = -1, c = -3, d = 1$

(7)



$E = (1.5, 0.75), F = (3.5, 3.5), G = (5.5, 4.3),$
$H = (2, 1.4), I = (4.1, 3.7),$
$J = (2.6, 2).$

(10) When the control points are colinear the convex hull is a line segment. The Bézier curve is contained inside that line segment, so must itself be a line segment.

(12) The rational quadratic Bézier with control points P_0, P_1, P_2 and weights w_0, w_1, w_2 is

$$P(u) = \frac{w_0(1-u)^2 P_0 + 2w_1(1-u)uP_1 + w_2u^2P_2}{w_0(1-u)^2 + 2w_1(1-u)u + w_2u^2}, \qquad u \in [0,1].$$

Assuming the parabola condition $w_1^2 = w_0w_2$, then set $a = \sqrt{w_0}$ and $b = \sqrt{w_2}$. Then $w_1 = ab$ and the denominator factors as a perfect square:

$$D(u) = w_0(1-u)^2 + 2w_1(1-u)u + w_2u^2 = \big(a(1-u) + bu\big)^2.$$

The numerator becomes

$$\begin{aligned} N(u) &= a^2(1-u)^2P_0 + 2ab(1-u)tP_1 + b^2u^2P_2 \\ &= \big(a(1-u) + bu\big)^2\Big[(1-t)^2P_0 + 2ut(1-t)P_1 + t^2P_2\Big], \end{aligned}$$

if we define

$$t = \frac{bu}{a(1-u) + bu}.$$

(Indeed $1 - t = \dfrac{a(1-u)}{a(1-u) + bu}$, so the usual Bernstein quadratic appears.) Therefore

$$P(u) = \frac{N(u)}{D(u)} = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2,$$

i.e. the rational curve equals an ordinary (polynomial) quadratic Bézier evaluated at the parameter t. The mapping $u \mapsto t$ is a smooth reparametrization of $[0,1]$ (it sends $u = 0 \mapsto t = 0, u = 1 \mapsto t = 1$) making it the same quadratic Bézier curve with control points P_0, P_1, P_2.