On Splines and Their Use in Computer Graphics and Generating Curves

Simon Abrelat

002129–0004

Math HL IA

May 2019

Words:

Abstract

Splines are used everywhere around us, from fonts to animations. They are the way that computers manipulate and store curves, and because of that have a large range of possible applications. Since they are so useful, there are also many of ways to generate these functions and lots of formats they come in. In general, they are fairly computationally cheap which makes them so applicable and expendable. This paper will illustrate and help visualize different types of splines.

Contents

1 Introduction

Splines are not a specific equation. They are more a class of equations that have similar properties. They are simply piecewise polynomials, and are often parameterized so that they can 'loop over themselves' and break the vertical line test. One of the interesting aspects of splines is that they keep this property even at low degrees.

The word spline is derived from wooden splines that curve and are often used in things like braces for ships. The major benefit of splines is how they are suited for computers. Almost all curves on a computer, from fonts to animation paths, are described quickly and accurately by splines such as Bèzier and Hermite curves. This means that these splines are around us all of the time and are almost invisible if you do not know where to look. There are many different kinds of splines with different methods of formulation. However, this paper will primarily focus on Hermite and Bèzier curves.

2 Bèzier Curves

2.1 Theory

Bèzier curves can be made with a linear combination of Bernstein polynomials (Casselman, 1984). Bernstein polynomials were first used in the proof of the Stone-Weierstrass theorem which states that every continuous function defined on a closed interval $[a, b]$ can be approximated as closely as desired by a polynomial (Pinkus, 2000). This means that a Bernstein polynomial can make any other function as the degree approaches infinity (3) A Bernstein polynomial (2) is defined by a linear combination of Bernstein basis functions (1).

Bernstein Polynomials:

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

$$B_n(t) = \sum_{\nu=0}^{n} \beta_\nu b_{\nu,n}(t) \tag{2}$$

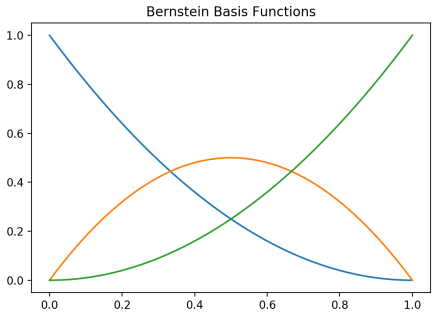
β_ν: the control point component

$$\lim_{n\to\infty} B_n(f) = f \tag{3}$$

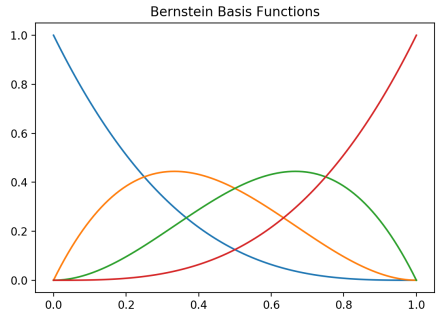
n: degree of the Bernstein function

ν: number of the $n+1$ equations of a function of degree n

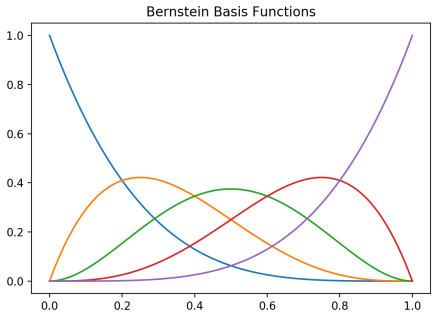
Figure 1: Basis functions for different degrees, Listing 1



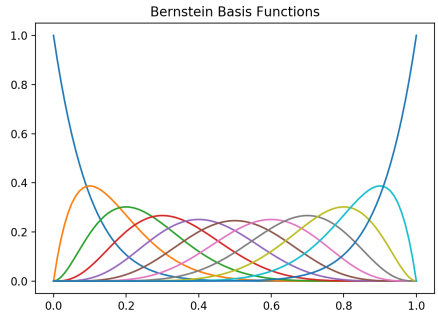
(a) $n = 2$



(b) $n = 3$



(a) $n = 4$



(b) $n = 10$

Bernstein functions are only approximations and to perfectly match the original function you would need an infinite degree Bernstein polynomials, similar to Taylor series (3). The algorithm used to generate a Bézier curve given a set of control points is called De Castlejau's algorithm. It is a recursive algorithm (4) that takes control points

and interpolates a curve between them. This method is a little more complicated than the explicit form (6) which is a summation of the control points multiplied by the basis functions of a given degree n. Then to generate any possible curve, parameterize the x and y axes and fun De Castlejau's algorithm on the scalar x and y quantities.

$$\beta_1^{(0)} = \beta_i \quad i = 0, \ldots, n \tag{4}$$

$$\beta_i^{(j)} = \beta_\nu^{(j-1)}(1 - t_0) + \beta_\nu^{(j-1)} t_0 \tag{5}$$

$j = 1, \ldots, n$

$\nu = 0, \ldots, n - j$

$$B(t) = \sum_{\nu=0}^{n} \beta_\nu b_{\nu,n}(t) \tag{6}$$

β_ν: ith control point

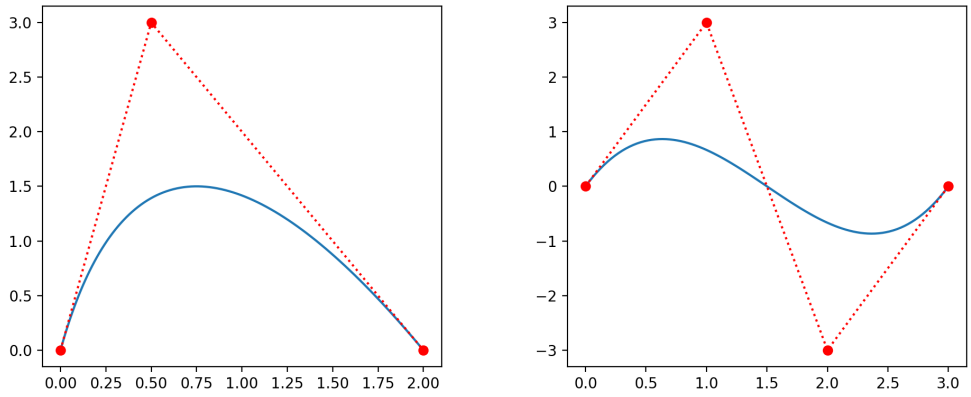
$b_{\nu,n}(t)$: Bernstein basis function for the number and degree

2.2 Application

There are practically an infinite number of uses for Bèzier curves. They, and other similar methods, are used to display all curves in computers which means that anything made graphically with curves

use Bèzier curves or similar splines. However, the hidden ubiquitous use of Bèzier curves in particular is in fonts. All PostScript fonts, the basis of PDFs, use cubic Bèzier curves (a) and TrueType fonts, abbreviated to ttf, use quadratic Bèzier curves (b).

Figure 3: Cubic and Quadratic Curves, Listing 2



(a) Cubic curve, 3 control points

(b) Quadratic curve, 4 control points

Most of the computer generated graphics, such as advertisements, animations, icons contain Bèzier curves. The popular software used for graphic design is Adobe Illustrator® which is a vector graphics editor that is entirely based on manipulating Bèzier curves. It generates curves with specific parameters being 2 anchor and 2 control points which is exactly the quadratic Bèzier curve.

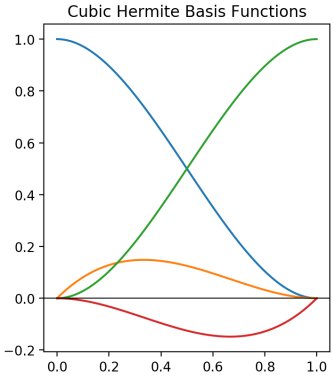
The other uses for Bèzier curves are in optimization problems. Since they are used to approximate a curve, and the control point can be related to the derivative of a function, they are very useful for optimization problems. They can be used to minimize path lengths (Jusko, 2016) and can even have restrictions such as a maximum acceleration or velocity (G. Jolly et al., 2009). The acceleration and velocity limits are caused by the control points which can be controlled since they are tangential to their anchor points meaning the angle is related to the derivative of the function and the magnitude of the control point is related to the acceleration or multiple control points. These values can even be used to model resistance in a parameterized manner so yet another use of Bèzier curves would be the generation of an optimal airfoil (Rogalsky et al., 2000).

3 Hermite

3.1 Theory

A cubic Hermite spline takes two control points and the derivatives. There are also higher order Hermite splines such as quintic splines that take 2 points their derivatives as well as their second derivatives. For this section, however, we will be focusing on cubic splines. These cubic splines have 4 basis functions much like the Bernstein basis functions for Bèzier curves. In fact, you can express Hermite basis functions in terms of their Bernstein counterparts (equation 1); however, a linear combination of Bernstein Basis functions approximates any polynomial. Two of those functions are for the points and then there are 2 for the derivatives.

Figure 4: Hermite Bases, Listing 3



$$
\begin{aligned}
h_{0,0}(t) = (1+2t)(1-t)^2 &= \beta_0(t) + \beta_1(t) \\
h_{1,0}(t) = t(1-t)^2 &= \frac{\beta_1(t)}{3} \\
h_{0,1}(t) = t^2(3-2t) &= \beta_2(t) + \beta_3(t) \\
h_{1,1}(t) = t^2(1-t) &= \frac{-\beta_2(t)}{3}
\end{aligned} \tag{7}
$$

i

$h_{0,0}$: is function for the first point

$h_{0,1}$: is function for the first derivative

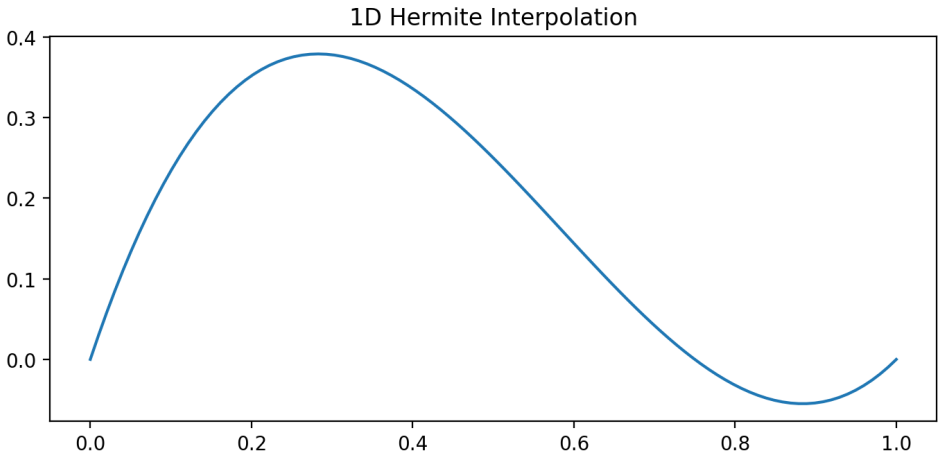
$h_{1,0}$: is function for the second point

$h_{1,1}$: is function for the second derivative

β_n: Bernstein basis function with degree 3 for index n

Another way to describe Hermite splines, as well as Bèzier curves, is through matrices (8). These matrices are equivalent to the basis functions. The benefit of using matrices is that it is easier to change between different types of splines for different purposes. For example, if one were making a application they could pass in the basis matrix to generate Hermite, Bèzier, Catmull-Rom, or other forms.

Figure 5: 1D Hermite Interpolation, Listing 4



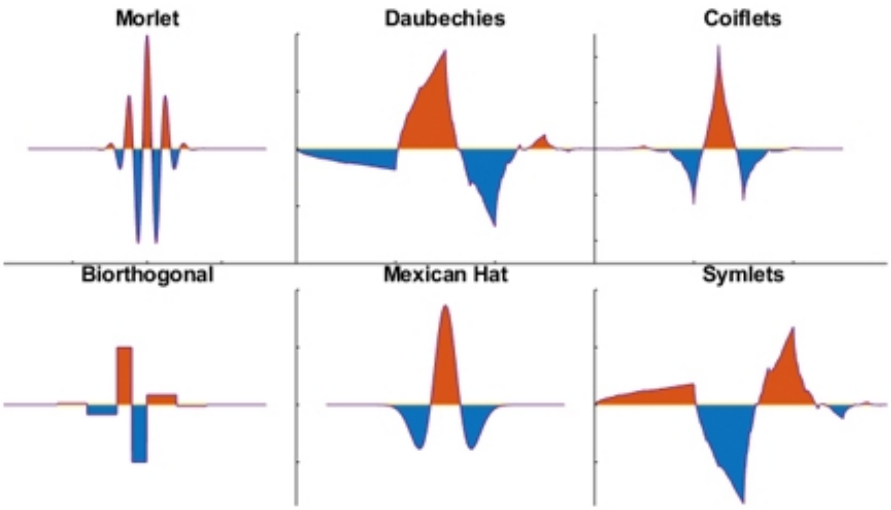
$$p = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{n-1} \\ p \\ p_{n+1} \\ p_{n+2} \end{bmatrix} \tag{8}$$

3.2 Application

Interpolated splines are useful for a host of different applications. Hermite splines are not as ubiquitous, but they have more specific uses.

They are often used for statistics since they can generate intermediate values between data points as well as being used for regression and filtering (Giles, 2010). Hermite splines are also used for path generation for simple mobile or differential robots. In the study Boeing (2008), they use genetic algorithms to optimize Hermite paths to simulate bipedal, tripedal, and 'snake-like' motion.

Figure 6: The different Types of Wavelets



One of the biggest uses of Hermite splines are wavelets (Averbuch et al., 2007). These are wavelets and not waves since they terminate and do not oscillate to infinity (figure 6). In signals processing, Fourier Transforms isolate the frequency and amplitude of a signal which in-

cludes all of the image. Wavelets allow one to find the local transformations such as edges or lines in an image. Most of the purely academic uses of Hermite splines are in modeling different forms of wavelets or other signal analysis. For example, in Uhlmann et al. (2014) they are using wavelets for edge detection as well as for modeling the blobs, or objects that make edges, to automatically analyze medical photos. As previously alluded to, these interpolating splines can be used for smoothing or filtering data. Hermite splines can also be used for interpolation between data points which can be used for filtering. So for the application of signals processing, splines can be used for generating data and cleaning it up.

4 Conclusion

Bèzier and Hermite splines are just two examples in a sea of possible splines. These are two simple types that are useful for outlining the two archetypes of splines, interpolating and weighted, while still being computationally simple. However, their simplicity does not subtract from their functionality or abundance in real life applications.

There are often 'better' or more specialized splines for any given use case; however, that misses the point of their power and generality. The naively simple task of making an approximation of curve between points has much more depth than one can possibly cover. The simplicity and generality also explains why splines are everywhere.

References

Averbuch, A. Z., Zheludev, V. A., and Cohen, T. (2007). Multiwavelet frames in signal space originated from hermite splines. *IEEE Transactions on Signal Processing*, 55(3):797–808.

Boeing, A. (2008). Morphology independent dynamic locomotion control for virtual characters. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 283–289.

Casselman, B. (1984). From bézier to bernstein. *American Mathematical Society*, 39(6):1087–1097.

G. Jolly, K., Sreerama Kumar, R., and Vijayakumar, R. (2009). A bezier curve based path planning in a multi-agent robot soccer system without violating the acceleration limits. 57:23–33.

Giles, D. E. (2010). Hermite regression analysis of multi-modal count data. *Economics Bulletin*, 30(4):2936–2945.

Jusko, T. (2016). Scalable trajectory optimization based on bézier curves. Deutscher Luft- und Raumfahrtkongress.

Park, K. S., Cho, B. H., Lee, D. H., Song, S. H., Lee, J. S., Chee, Y. J., Kim, I. Y., and Kim, S. I. (2008). Hierarchical support vector machine based heartbeat classification using higher order statistics and hermite basis function. In *2008 Computers in Cardiology*, pages 229–232.

Pinkus, A. (2000). Weierstrass and approximation theory. *Journal of Approximation Theory*, 107:1–66.

Rogalsky, T., W. Derksen, R., N, R., and Kocabiyik, S. (2000). Differential evolution in aerodynamic optimization. *Canadian Aeronautics and Space Journal*.

Uhlmann, V., Delgado-Gonzalo, R., Conti, C., Romani, L., and Unser, M. (2014). Exponential hermite splines for the analysis of biomedical images. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1631–1634.

5 Appendix

List of Figures

Listing 1: Bernstein Basis Functions

```
from numpy import linspace as lin  # linespace
from scipy.special import binom  # binomial coefficient
import matplotlib.pyplot as plt  # plotting
from math import pow # power function

x = lin(0, 1, 100)  # Input as 100 floats between 0 and 1
num: int = 10  # The degree

# returns the value from the Bernstein basis function at the point x
def bbf(i: int, n: int, x: float) -> float:
    return binom(n, i) * pow(x, i) * pow((1 - x), (n-i))

# Iterates through all of the functions for the degree
for c in range(num + 1):
```

```
15      # Plots all of the functions
16      plt.plot(x, [bbf(c, num, i) for i in x])
17
18  plt.title("Bernstein Basis Functions")  # Title
19  plt.show()  # displays graphs
```

Listing 2: Bèzier Curves

```
from numpy import linspace as lin  # linespace
from scipy.special import binom  # binomial coefficient
import matplotlib.pyplot as plt  # plotting
from typing import List # type annoations and checking
from math import pow # power function

cpx: List[float] = [0.0, 0.5, 2.0]
cpy: List[float] = [0.0, 3.0, 0.0]

t = lin(0, 1, 100)  # Input as 100 floats between 0 and 1
num: int = 2  # The degree 0 indexed

# returns the value from the Bernstein basis function at the point x
def bbf(i: int, n: int, x: float) -> float:
    return binom(n, i) * pow(x, i) * pow((1 - x), (n-i))

# Generalized De Casteljau's Explicit formula
def f(a: List[float], t: float) -> float:
    ret: float = 0
    for i in range(num + 1):
        ret += a[i] * bbf(i, num, t)
    return ret

def x(t: float) -> float: return f(cpx, t) # implemented on X's
   control points
def y(t: float) -> float: return f(cpy, t) # implemented on Y's
```

```
    control points
26
27 plt.plot([x(i) for i in t], [y(i) for i in t]) # plots X and Y
28 plt.plot(cpx, cpy, 'ro') # plots the control points
29 plt.plot(cpx, cpy, 'r:') # plots lines between control points
30 plt.show()
```

Listing 3: Hermite Basis Functions

```
1  from numpy import linspace as lin  # linespace
2  from scipy.special import binom # binomial coefficient
3  import matplotlib.pyplot as plt # Graphs
4  from math import pow # power function
5
6  d: int = 3 # Degree of the polynomial
7  t = lin(0, 1, 100) # The time
8
9  # The cubic hermite basis functions
10 def H(n: int, x: float) -> float:
11     if  (n == 0): return (2 * pow(x, 3)) - (3 * pow(x, 2)) + 1 # H00
12     elif(n == 1): return pow(x, 3) - (2 * pow(x, 2)) + x       # H01
13     elif(n == 2): return (-2 * pow(x, 3)) + (3 * pow(x, 2))    # H10
14     elif(n == 3): return pow(x, 3) - pow(x, 2)           # H11
15     else: return 0
16
17 # returns the value from the Bernstein basis function at the point x
18 def bbf(i: int, n: int, x: float) -> float:
19     return binom(n, i) * pow(x, i) * pow((1 - x), (n-i))
20
```

```
# The bernstein form of the cubic hermite basis function
def BH(n: int, x: float) -> float:
    B = lambda i: bbf(i, 3, x) # pythonic function curry
    if  (n == 0): return B(0) + B(1) # H00
    elif(n == 1): return B(1) / 3    # H01
    elif(n == 2): return B(3) + B(2) # H10
    elif(n == 3): return B(2) / -3   # H11
    else: return 0

# graph configs
plt.title("Cubic Hermite Basis Functions")
plt.axis('scaled') # equalize axes
plt.axhline(0, color='black', linewidth=.75) # y axis line

for n in range(d+1): plt.plot(t, [H(n, i) for i in t]) # plots all
   of the funcs
plt.show()
```

Listing 4: Hermite 1D interpolation

```
import matplotlib.pyplot as plt # Graphing Lib
from typing import List # Type Inference
from math import pow # Poewr function
import numpy as np # Arrays and matrices

p: List[float] = [0, 3] # The control points
d: List[float] = [0, 1] # the derivative at said control points
t = np.linspace(0, 1, 100) # an array of time values

```

```
# hermite spline interpolation between 2 points
def hermite(t1: float, t2: float, dt1: float, dt2: float, s: float)
    -> float:
     S = np.matrix([[pow(s, 3), pow(s, 2), s, 1]]) # horizontal
    matrix for x
     C = np.matrix([[t1, t2, dt1, dt2]]).T # verticle matrix for
    input values

     h = np.matrix([[ 2, -2,  1,  1], # weight matrix that represent
                    [-3,  3, -2, -1], # the basis functions
                    [ 0,  0,  1,  0],
                    [ 1,  0,  0,  0]])
     # multiplies the matrices all together and gets the scalar from
    them
     return np.asscalar(S * h * C)

# graph configs
plt.title("1D Hermite Interpolation")
# graphs the equation
plt.plot(t, [hermite(p[0], d[0], p[1], d[1], i) for i in t])
plt.show()
```