Chapter 13

Floating Point Error

Most scientific computing is performed in *floating point* arithmetic in the binary number system. For our discussion, consider the floating point number represented using 32 bits of memory, with $d = 23$ bits of fraction and 8 bits of exponent

$$b = \overbrace{s}^{\text{sign}} \,|\, \underbrace{e_s}_{\text{exponent sign}} \; \overbrace{e_7 e_6 e_5 e_4 e_3 e_2 e_1 e_0}^{\text{exponent}} \,|\, \overbrace{f_1 f_2 f_3 f_4 \dots f_{21} f_{22} f_{23}}^{\text{fraction}} . \tag{13.1}$$

This string of 32 bits serve as sort of a "code" for representing numbers using the base 2 number system. The decoding formula is to

$$b = (-1)^s f^e = (-1)^s \left[\sum_{i=1}^{d} f_i 2^{-i} \right]^{(-1)^{e_s} \sum j=0^6 e_j 2^j} . \tag{13.2}$$

For example,

$$0|000000100|11010000000000000000000 = .1101_2 \times 2^4 = 1101_2 = 13$$

$$0|100000001|11100000000000000000000 = .111_2 \times 2^{-1} = .0111_2 = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = \frac{7}{16}$$

$$1|000000100|10011000000000000000000 = -.10011_2 \times 2^4 = -1001.1_2 = -9\frac{1}{2}$$

One limitation of this floating point number system is that not every number can be represented exactly. For example, the number $\frac{1}{3}$ is a repeating decimal in the binary number system:

$$\frac{1}{3} = 0.01010\bar{1}_2$$

and thus there is no floating point number which exactly equals $\frac{1}{3}$. It lies between the two adjacent floating point numbers

$$0|100000001|10101010101010101010101 < \frac{1}{3} < 0|100000001|10101010101010101010110$$

Therefore, a floating point *number* is best thought of as an *interval* which contains all numbers which are closer to it than to any other floating point number. If d is the number of bits in the fraction of a given floating point number b, then that interval is

$$[b] = [b(1 - 2^{-(d+1)}), b(1 + 2^{-(d+1)})] = b[1 - 2^{-(d+1)}, 1 + 2^{-(d+1)}]. \tag{13.3}$$

This chapter discusses the floating point error incurred in evaluating a Bézier curve. Figure 13.1 shows a Bézier curve which is tangent to the horizontal axis at $t = 0.25$. Due to numerical innacuracies in floating point arithmetic, it is not surprising that when the curve is evaluated at $t = .25$ (using either the de Casteljau algorithm or Horner's method) the answer is not exactly zero. It is interesting to note what happens when we evaluate the curve at several hundred values close to $t = 0.25$. Rather than a producing a smooth graph, the evaluations fluctuate chaotically within a band, as shown in the blowup in figure 13.1.

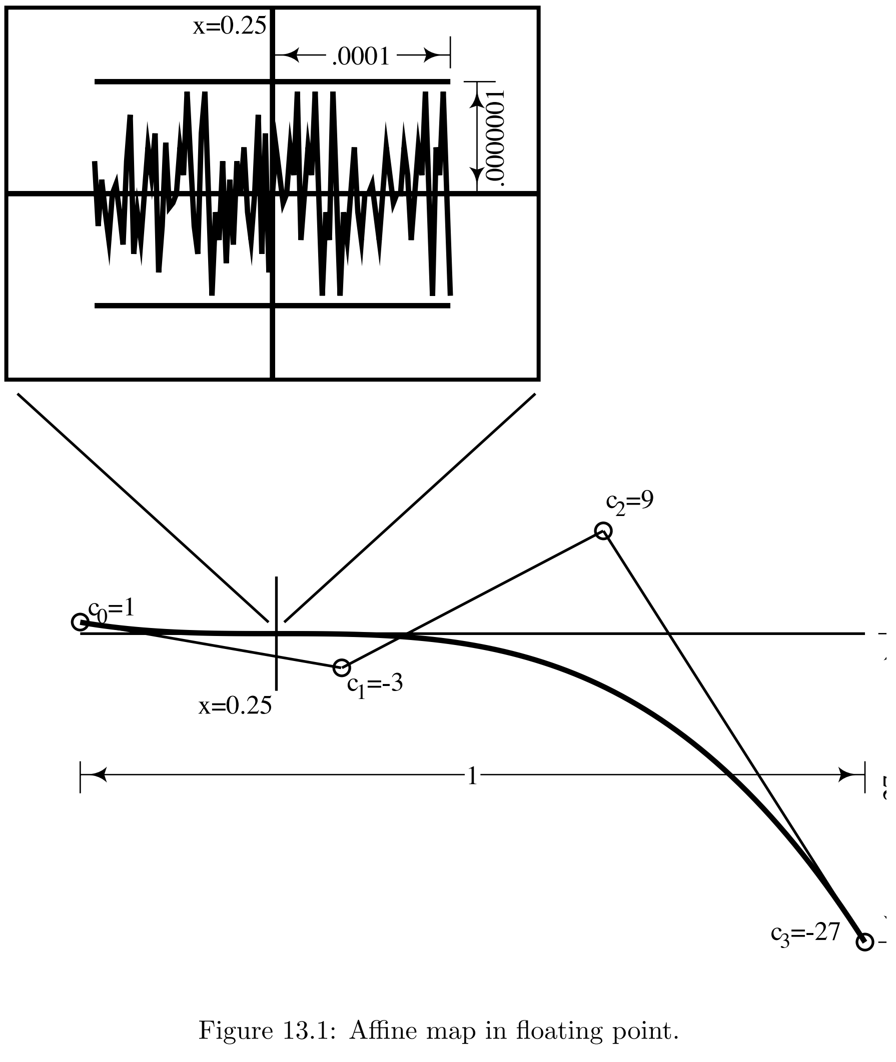

Figure 13.1: Affine map in floating point.

So, how does one determine the effects of floating point error on calculations involving Bézier curves?

Two sources of error can be considered somewhat independently. The first source of error is in the initial floating point representation of the control points. First

Our previous discussions on interval Bézier curves should be modified slightly if one wishes to accurately bound the uncertainty introduced when using finite–precision arithmetic. Let us first revisit the problem of computing the affine map of two vector intervals shown in Figure 12.4. When finite precision arithmetic is involved, some uncertainty is introduced. In order to robustly represent that uncertainty, the resulting interval should be widened an appropriate amount $\epsilon(t)[i]$ as shown in Figure 13.2. Here, the affine map for $t = \frac{1}{2}$ is shown with the exact arithmetic in solid and floating point error bounds in dashed line.

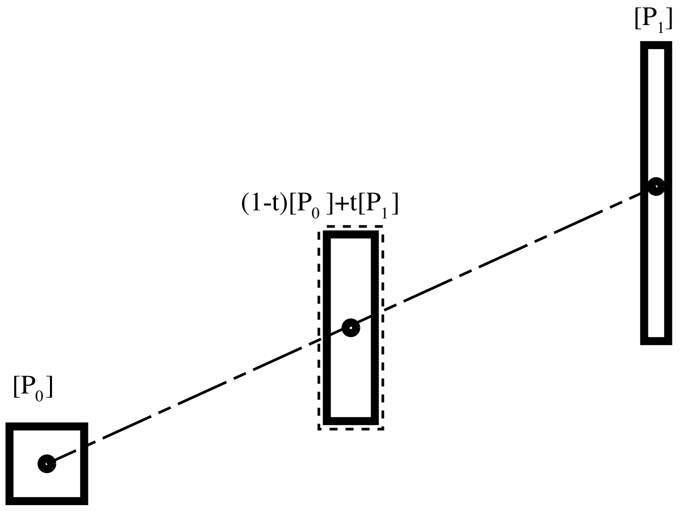

Figure 13.2: Affine map in floating point.

Given a floating point mantissa of d binary digits, the *machine unit* for roundoff is

$$\eta = 2^{-d}. \tag{13.4}$$

If $x * y$, where $* \in \{+, -, \times, \div\}$, denotes an exact computation, and $\mathrm{fl}(x * y)$ denotes the floating point, imprecise computation, then

$$\mathrm{fl}(x * y) \in x * y\,[1-\eta, 1+\eta] = x * y\,(1 + \eta[i]). \tag{13.5}$$

Applying this to the affine map equation 12.26 yields (see equation 26 in [FR87])

$$\mathrm{fl}[\mathbf{M}](p_0 + e_0[i], p_1 + e_1[i], t) \in [\mathbf{M}](p_0 + e_0[i], p_1 + e_1[i], t) + \epsilon(t) \tag{13.6}$$

where

$$\begin{aligned} \epsilon(t) &= \begin{cases} \{|p_0|(1-t) - |p_1|t + |p_0(1-t) + p_1 t|\}\,\eta\,[i] & \text{for } t < 0\,, \\ \{|p_0|(1-t) + |p_1|t + |p_0(1-t) + p_1 t|\}\,\eta\,[i] & \text{for } 0 \le t \le 1\,, \\ \{|p_0|(t-1) + |p_1|t + |p_0(1-t) + p_1 t|\}\,\eta\,[i] & \text{for } t > 1\,, \end{cases} && (13.7) \\ &= \{|1-t||p_0| + |t||p_1| + |p_0(1-t) + p_1 t|\}\,\eta\,[i]\,. && (13.8) \end{aligned}$$

In words, to represent the effects of floating point roundoff in computing an affine map, the rectangle obtained by computing the affine map in exact arithmetic must be fattened an absolute amount $\epsilon(t)$.

Additional discussion on the error propagation in operations on Bernstein–form polynomials can be found in [FR87].