Please cite as: K. Hormann. Efficient evaluation of interpolating cubic polynomials. Technical Report IfI-08-04, Department of Informatics, Clausthal University of Technology, August 2008.

Efficient Evaluation of Interpolating Cubic Polynomials

Kai Hormann

Department of Informatics
Clausthal University of Technology

Abstract

This short note discusses the efficient evaluation of interpolating cubic polynomials with a focus on the evaluation at the midpoint between the interior interpolation points.

1 Introduction

Consider data $f_0, f_1, f_2, f_3 \in \mathbb{R}^d$ given at the interpolation points t_0, t_1, t_2, t_3 and let $f\colon \mathbb{R} \to \mathbb{R}^d$ be the cubic polynomial satisfying $f(t_i) = f_i$ for $i = 0, 1, 2, 3$. We review several algorithms for evaluating f, in particular at $t_* = (t_1 + t_2)/2$ and discuss their efficiency. This evaluation is the main operation of the parametric four-point subdivision scheme by Dyn, Floater, and Hormann [2] and its implementation significantly influences the performance.

2 Neville's algorithm

A popular choice for evaluating interpolating polynomials is *Neville's algorithm* [4], which computes $f(t)$ recursively:

$$f_0^1 = \frac{(t_1 - t)f_0 + (t - t_0)f_1}{t_1 - t_0}, \qquad f_1^2 = \frac{(t_2 - t)f_1 + (t - t_1)f_2}{t_2 - t_1}, \qquad f_2^3 = \frac{(t_3 - t)f_2 + (t - t_2)f_3}{t_3 - t_2},$$
$$f_0^2 = \frac{(t_2 - t)f_0^1 + (t - t_0)f_1^2}{t_2 - t_0}, \qquad f_1^3 = \frac{(t_3 - t)f_1^2 + (t - t_1)f_2^3}{t_3 - t_1}, \tag{1}$$
$$f_0^3 = \frac{(t_3 - t)f_0^2 + (t - t_0)f_1^3}{t_3 - t_0},$$

and $f(t) = f_0^3$. Computing each intermediate term this way requires $(3 + d)$ additions (A), $2d$ multiplications (M), and d divisions (D). For $d > 1$ we can reduce the number of divisions to one by writing all terms as *affine combinations*; for example,

$$f_0^3 = \alpha_0^3 f_0^2 + \bigl(1 - \alpha_0^3\bigr) f_1^3 \qquad \text{with} \qquad \alpha_0^3 = \frac{t_3 - t}{t_3 - t_0},$$

and likewise for the other terms. Moreover, the six weights α_i^j have only three different numerators, so we can save another 3 A by precomputing them[1]. Overall, this amounts to a total of $(15 + 6d)$ A, $12d$ M, and 6 D. For the evaluation of f at t_* we can further reduce the costs by noticing that

$$t_* - t_1 = t_2 - t_* = \frac{t_2 - t_1}{2} =: T \tag{2}$$

and therefore $f_1^2 = \frac{1}{2}(f_1 + f_2)$. We then end up with $(12 + 6d)$ A, $(1 + 11d)$ M, and 5 D; see Algorithm 1.

[1] For small d it can be better to stick to the formulas in (1). Note that only the four terms $t - t_i$ appear as factors in the numerators so that precomputing them saves 8 A. Thus, we can compute $f(t)$ with $(10 + 6d)$ A, $12d$ M, and $6d$ D.

```
function f (real t[4], vec(d) f[4])
  t[0] = t[0]-t[1]                                   // 1 A
  t[1] = t[2]-t[1]                                   // 1 A
  t[2] = t[2]-t[3]                                   // 1 A
  t[3] = t[1]-t[2]                                   // 1 A
  real T = 0.5*t[1]                                  // 1 M
  real a = T/t[0]; real b = 1-a                      // 1 A, 1 D
  f[0] = a*f[0]+b*f[1]                               // d A, 2d M
  f[1] = 0.5*(f[1]+f[2])                             // d A, d M
  b = T/t[2]; a = 1-b                                // 1 A, 1 D
  f[2] = a*f[2]+b*f[3]                               // d A, 2d M
  a = T/(t[1]-t[0]); b = 1-a                         // 2 A, 1 D
  f[0] = a*f[0]+b*f[1]                               // d A, 2d M
  b = T/t[3]; a = 1-b                                // 1 A, 1 D
  f[1] = a*f[1]+b*f[2]                               // d A, 2d M
  a = (T-t[2])/(t[3]-t[0]); b = 1-a                  // 3 A, 1 D
  f[0] = a*f[0]+b*f[1]                               // d A, 2d M
  return f[0]
```

Algorithm 1: Neville's algorithm for evaluating $f(t_*)$.

3 Newton basis

A closely related option, which is generally considered to be more efficient [5], is to write the interpolating polynomial with respect to the *Newton basis*,

$$f(t) = \sum_{i=0}^{3} N_i(t) f_0^i \qquad \text{with} \qquad f_0^i = [t_0, \dots, t_i]f \qquad \text{and} \qquad N_i(t) = \prod_{j=0}^{i-1} (t - t_j)$$

and then use the *Horner scheme* to evaluate it. The first coefficient is $f_0^0 = f_0$ and the others can be computed recursively, like in Neville's algorithm:

$$\begin{gathered} f_0^1 = \frac{f_1 - f_0}{t_1 - t_0}, \qquad f_1^2 = \frac{f_2 - f_1}{t_2 - t_1}, \qquad f_2^3 = \frac{f_3 - f_2}{t_3 - t_2}, \\ f_0^2 = \frac{f_1^2 - f_0^1}{t_2 - t_0}, \qquad f_1^3 = \frac{f_2^3 - f_1^2}{t_3 - t_1}, \\ f_0^3 = \frac{f_1^3 - f_0^2}{t_3 - t_0}. \end{gathered} \tag{3}$$

This requires $(6 + 6d)\,\mathsf{A}$ and $6d\,\mathsf{D}$ and each subsequent evaluation costs another $(3 + 3d)\,\mathsf{A}$ and $3d\,\mathsf{M}$:

$$f(t) = f_0^0 + (t - t_0)\big[f_0^1 + (t - t_1)\big(f_0^2 + (t - t_2) f_0^3\big)\big]. \tag{4}$$

This can be optimized slightly by postponing some of the divisions in (3) to the final evaluation. With

$$\begin{gathered} \tilde{f}_0^2 = f_1^2 - f_0^1, \qquad \tilde{f}_1^3 = \frac{t_2 - t_0}{t_3 - t_1}(f_2^3 - f_1^2), \\ \tilde{f}_0^3 = \tilde{f}_1^3 - \tilde{f}_0^2 \end{gathered} \tag{5}$$

we have

$$f(t) = f_0^0 + (t - t_0)\left[f_0^1 + \frac{t - t_1}{t_2 - t_0}\left(\tilde{f}_0^2 + \frac{t - t_2}{t_3 - t_0}\tilde{f}_0^3\right)\right],$$

yielding overall costs of $(9 + 9d)\,\mathsf{A}$, $4d\,\mathsf{M}$, and $(3 + 3d)\,\mathsf{D}$. For computing $f(t_*)$, we exploit again the symmetry in (2), which saves $2\,\mathsf{A}$ at the cost of $1\,\mathsf{M}$; see Algorithm 2.

```
function f (real t[4], vec(d) f[4])
  t[0] = t[1]-t[0]                                  // 1 A
  t[1] = t[2]-t[1]                                  // 1 A
  t[2] = t[3]-t[2]                                  // 1 A
  t[3] = t[0]+t[1]                                  // 1 A
  f[3] = (f[3]-f[2])/t[2]                           // d A, d D
  f[2] = (f[2]-f[1])/t[1]                           // d A, d D
  f[1] = (f[1]-f[0])/t[0]                           // d A, d D
  f[3] = (f[3]-f[2])*t[3]/(t[1]+t[2])               // (1+d) A, d M, 1 D
  f[2] = f[2]-f[1]                                  // d A
  f[3] = f[3]-f[2]                                  // d A
  real T = 0.5*t[1]                                 // 1 M
  t[2] = T/(t[3]+t[2])                              // 1 A, 1 D
  t[1] = T/t[3]                                     // 1 D
  t[0] = t[3]-T                                     // 1 A
  f[0] = f[0]+t[0]*(f[1]+t[1]*(f[2]-t[2]*f[3]))     // 3d A, 3d M
  return f[0]
```

Algorithm 2: Evaluating $f(t_*)$ using the Newton basis.

4 Lagrange basis

Instead of the Newton basis, we can also consider writing f with respect to the *Lagrange basis*,

$$f(t) = \sum_{i=0}^{3} L_i(t) f_i \qquad \text{with} \qquad L_i(t) = \prod_{j=0, j\neq i}^{3} \frac{t - t_j}{t_i - t_j}. \tag{6}$$

Evaluating f this way requires to compute first all differences $t - t_j$ and $t_i - t_j$ with 10 A, then the terms $L_i(t)$, each with 4 M and 1 D, and finally $f(t)$ with another $3d$ A and $4d$ M, yielding a total of $(10 + 3d)$ A, $(16 + 4d)$ M, and 4 D. However, we can save 4 M by first computing

$$p_0 = \frac{t_3 - t}{(t_1 - t_0)(t_2 - t_0)}, \qquad p_1 = \frac{(t_2 - t)(t - t_1)}{t_0 - t_3}, \qquad p_3 = \frac{t - t_0}{(t_3 - t_1)(t_3 - t_2)}, \qquad p_2 = \frac{p_0 p_3}{t_2 - t_1}$$

and then

$$\begin{aligned} L_0(t) &= p_0 p_1, & L_1(t) &= p_2 (t_2 - t)(t_2 - t_0)(t_3 - t_2), \\ L_3(t) &= p_1 p_3, & L_2(t) &= p_2 (t - t_1)(t_1 - t_0)(t_3 - t_1). \end{aligned}$$

Exploiting symmetry, we can save another 2 A and 1 D for the particular evaluation at t_*; see Algorithm 3.

5 Barycentric form

Another alternative is to write the interpolating cubic in its *barycentric form* [1],

$$f(t) = \sum_{i=0}^{3} \frac{w_i}{t - t_i} f_i \Big/ \sum_{i=0}^{3} \frac{w_i}{t - t_i} \qquad \text{with} \qquad w_i = \prod_{j=0, j\neq i}^{3} \frac{1}{t_i - t_j}. \tag{7}$$

Once the quotients $q_i(t) = w_i/(t - t_i)$ are known, evaluating the barycentric form requires $3d$ A and $4d$ M for the numerator, 3 A for the denominator, and d D for the final division. There are, however, several ways of computing the q_i.

```
function f (real t[4], vec(d) f[4])
  real t10 = t[1]-t[0]                                  // 1 A
  real t20 = t[2]-t[0]                                  // 1 A
  real t31 = t[3]-t[1]                                  // 1 A
  real t32 = t[3]-t[2]                                  // 1 A
  real T = 0.5*(t[2]-t[1])                              // 1 A, 1 M
  t[0] = (T+t32)/(t10*t20)                              // 1 A, 1 M, 1 D
  t[1] = -T*T/(t10+t31)                                 // 1 A, 1 M, 1 D
  t[3] = (T+t10)/(t31*t32)                              // 1 A, 1 M, 1 D
  t[2] = 0.5*t[0]*t[3]                                  // 2 M
  t[0] = t[0]*t[1]                                      // 1 M
  t[3] = t[1]*t[3]                                      // 1 M
  t[1] = t[2]*t20*t32                                   // 2 M
  t[2] = t[2]*t10*t31                                   // 2 M
  f[0] = t[0]*f[0]+t[1]*f[1]+t[2]*f[2]+t[3]*f[3]        // 3d A, 4d M
  return f[0]
```

Algorithm 3: Evaluating $f(t_*)$ using the Lagrange basis.

Equation (7) suggests to first determine all differences $t - t_i$ and $t_i - t_j$ (10 A) and then to compute each $q_i(t)$ with 3 M for the denominator and 1 D for the inversion. Alternatively, we can exploit the fact that the barycentric formula for $f(t)$ in (7) still holds if all *barycentric weights* w_i are multiplied by some common factor. Let this common factor be $(t_2 - t_1)(t_2 - t_0)(t_3 - t_1)$, then we get the alternative weights

$$\tilde{w}_i = r_{i+1} - r_i \qquad \text{with} \qquad r_i = (-1)^i \frac{t_2 - t_1}{t_i - t_{i-1}}.$$

We can now compute first the terms r_i with 4 A and 3 D (notice that $r_2 = 1$), then the $\tilde{w}_i$ with 4 A, and finally the $\tilde{q}_i(t) = \tilde{w}_i/(t - t_i)$ with another 4 A, 4 D. Overall this saves 12 M at the cost of 2 A, 3 D. For the evaluation of f at t_* we can further reduce costs by realizing that

$$\frac{T}{t_* - t_0} = \frac{r_1}{r_1 - 2}, \qquad \frac{T}{t_* - t_1} = 1, \qquad \frac{T}{t_* - t_2} = -1, \qquad \frac{T}{t_* - t_3} = \frac{-r_3}{r_3 - 2}$$

for $T = (t_2 - t_1)/2$, so that evaluating the quotients $\bar{q}_i(t_*) = T\tilde{w}_i/(t_* - t_i)$ requires only 2 A, 2 M, 2 D. We then end up with $(13 + 3d)$ A, $(2 + 4d)$ M, and $(5 + d)$ D in total; see Algorithm 4.

```
function f (real t[4], vec(d) f[4])
  real T = t[2]-t[1]                                    // 1 A
  real r0 = T/(t[0]-t[3])                               // 1 A, 1 D
  real r1 = T/(t[0]-t[1])                               // 1 A, 1 D
  real r3 = T/(t[2]-t[3])                               // 1 A, 1 D
  t[0] = (r1-r0)*r1/(r1-2)                              // 2 A, 1 M, 1 D
  t[1] = 1-r1                                           // 1 A
  t[2] = 1-r3                                           // 1 A
  t[3] = (r3-r0)*r3/(r3-2)                              // 2 A, 1 M, 1 D
  f[0] = t[0]*f[0]+t[1]*f[1]+t[2]*f[2]+t[3]*f[3]        // 3d A, 4d M
  f[0] = f[0]/(t[0]+t[1]+t[2]+t[3])                     // 3 A, d D
  return f[0]
```

Algorithm 4: Evaluating $f(t_*)$ using the barycentric form.

```
function f (real t[4], vec(d) f[4])
  real t10 = t[1]-t[0]                                          // 1 A
  real t21 = t[2]-t[1]                                          // 1 A
  real t32 = t[3]-t[2]                                          // 1 A
  real t30 = t[3]-t[0]                                          // 1 A
  real t20 = t[2]-t[0]                                          // 1 A
  real t31 = t[3]-t[1]                                          // 1 A

  real T = 0.5*t21                                              // 1 M
  real a = t10+T                                                // 1 A
  real b = t32+T                                                // 1 A
  real c = T*t21;                                               // 1 M

  t[1] = t20*t32;                                               // 1 M
  t[2] = t10*t31;                                               // 1 M
  t[0] = b*t31*t32*c;                                           // 3 M
  t[3] = a*t20*t10*c;                                           // 3 M

  T = -0.5/(t[1]*t[2]*t30);                                     // 2 M, 1 D
  c = -a*b*t30;                                                 // 2 M

  t[1] = t[1]*c;                                                // 1 M
  t[2] = t[2]*c;                                                // 1 M

  f[0] = (t[0]*f[0]+t[1]*f[1]+t[2]*f[2]+t[3]*f[3])*T            // 3d A, 5d M

  return f[0]
```

Algorithm 5: Evaluating $f(t_*)$ with a single division.

6 Avoiding divisions

On many processors, divisions are much more costly than additions and multiplications and should be avoided by all means. So whenever we divide a data value $f_i \in \mathbb{R}^d$ by some scalar t_j, it can be advantageous to multiply f_i by $1/t_j$ instead, thus replacing $d\,\mathsf{D}$ with $d\,\mathsf{M}$, $1\,\mathsf{D}$. In Neville's algorithm we avoid these kind of divisions by converting the terms in (1) into affine combinations, and we also eliminate three of them from the algorithm based on the Newton basis when we replace half of the terms in (3) with the ones in (5). However, the remaining three terms f_0^1, f_1^2, f_2^3 can be modified as mentioned and the same holds for the final division of the barycentric from in (7).

Despite these optimizations, the algorithm based on the Lagrange basis remains the one with the fewest divisions. But we can still improve on this and derive an algorithm that gets by with a single scalar division. We simply multiply the Lagrange basis functions in (6) by their common denominator

$$P = (t_1 - t_0)(t_2 - t_0)(t_3 - t_0)(t_3 - t_1)(t_3 - t_2)(t_2 - t_1)$$

and consider

$$f(t) = \frac{1}{P}\sum_{i=0}^{3} \tilde{L}_i(t) f_i \qquad \text{with} \qquad \tilde{L}_i(t) = L_i(t)P.$$

The scaled basis functions $\tilde{L}_i$ and P are products of six factors each, but exploiting that they have up to three factors in common, we can save $6\,\mathsf{M}$ and evaluate them with $19\,\mathsf{M}$ only. Overall, we thus require $(10 + 3d)\,\mathsf{A}$, $(19 + 5d)\,\mathsf{M}$, and $1\,\mathsf{D}$ for computing $f(t)$ this way, and compared to the algorithm described in Section 4 we save $3\,\mathsf{D}$ at the cost $(7 + d)\,\mathsf{M}$. The particular evaluation at t_* can be further optimized due to symmetry; see Algorithm 5.

On architectures where multiplications are considerably more expensive than additions, it can be favourable to notice that

$$P = \sum_{i=0}^{3} \tilde{L}_i(t),$$

because the Lagrange basis functions form a partition of unity.

		additions	multiplications	divisions	total
Neville	(Algorithm 1)	$12+6d$	$1+11d$	5	$19+17d$
Newton	(Algorithm 2)	$7+9d$	$1+7d$	6	$14+16d$
Lagrange	(Algorithm 3)	$8+3d$	$12+4d$	3	$23+7d$
barycentric	(Algorithm 4)	$13+3d$	$2+5d$	6	$21+8d$
single division	(Algorithm 5)	$8+3d$	$16+5d$	1	$25+8d$

Table 1: Costs of the different algorithms for computing $f(t_*)$.

d	1	2	3	4	5	6
Neville	100	128	156	181	211	242
Newton	114	134	162	191	226	262
Lagrange	56	79	86	102	118	129
barycentric	112	118	132	139	158	186
single division	29	45	54	73	86	92

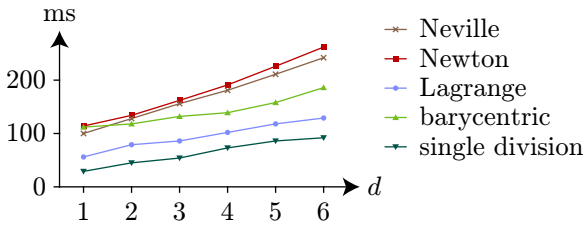

Table 2: Runtimes for 1 000 000 evaluations in ms.

7 Conclusion

We have implemented and timed all algorithms with `double` precision using `C++` and the `gcc` compiler with `-O2` optimization under `cygwin` on an Intel Pentium M processor with 2 GHz and 1 GB of RAM. For $d > 1$ we used a very simple class that groups d `double` values into a vector and overloads the basic math functions (`+`, `-`, `*`, `/`, `=`) accordingly. Since divisions are up to five times slower than multiplications on this processor, we eliminated as many as possible in Algorithms 2 and 4 as described in the beginning of Section 6. Table 1 summarizes the number of additions, multiplications, and divisions for each algorithm and Table 2 shows the costs for 1 000 000 evaluations for random data and interpolation points.

In this test scenario, Algorithm 5 turns out to be the fastest and we got similar results in a second scenario where we used the different algorithms for curve subdivision as described in [2]. However, some of the other tests that we ran were won by Algorithm 3 or even Algorithm 4, so the perfect choice depends on the particular situation. Moreover, the algorithms can always be further optimized and fine-tuned for specific compilers, processors, and operating systems [3].

We finally note that the Newton basis is the best to work with in case the polynomial f must be evaluated not only at t_* but at several parameter values, because once the coefficients in (3) are determined, computing $f(t)$ as in (4) is the most efficient way.

References

[1] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Review*, 46(3):501–517, Sept. 2004.

[2] N. Dyn, M. S. Floater, and K. Hormann. Four-point curve subdivision based on iterated chordal and centripetal parameterizations. Technical Report IfI-07-06, Department of Informatics, Clausthal University of Technology, Oct. 2007.

[3] A. Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. `http://www.agner.org/optimize/optimizing_cpp.pdf`, July 2008.

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*, chapter 3.2. Cambridge University Press, New York, third edition, 2007.

[5] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*, volume 12 of *Texts in Applied Mathematics*, chapter 2.1.3. Springer, New York, third edition, 2002.