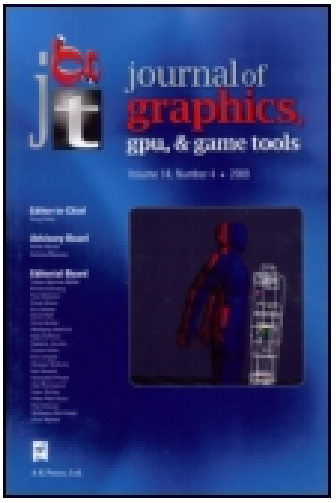

Journal of Graphics, GPU, and Game Tools

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ujgt20

A Fast and Accurate Algorithm for Computing SLERP

David Eberly [a]

[a] Microsoft Corporation

Published online: 21 Oct 2011.

To cite this article: David Eberly (2011) A Fast and Accurate Algorithm for Computing SLERP, Journal of Graphics, GPU, and Game Tools, 15:3, 161-176, DOI: 10.1080/2151237X.2011.610255

To link to this article: http://dx.doi.org/10.1080/2151237X.2011.610255

Vol. 15, No. 3: 161–176

A Fast and Accurate Algorithm for Computing SLERP

David Eberly
Microsoft Corporation

Abstract. The SLERP of quaternions is a common operation in keyframe animation. The operation can be a significant bottleneck in an animation-heavy application. The standard implementation of SLERP for an FPU typically involves trigonometric function evaluations, divisions, and branching. Commonly available SIMD implementations will modify the standard implementation and obtain a moderate speed-up. This paper provides a novel approach to computing SLERP, using only multiplications and additions. The algorithm is based on ideas from Chebyshev polynomials, power series solutions for differential equations, and error balancing using the Chebyshev Equioscillation Theorem and the associated Remez Algorithm. Implementations are provided for the FPU and SIMD. When SLERPing a pair of quaternions in parallel using four time samples, performance measurements show a 10-fold acceleration per SLERP over the standard SLERP implementation on an FPU. Source code is available online.

1. Introduction

A standard operation applied to unit-length quaternions is *spherical linear interpolation* (SLERP), made popular in Computer Graphics by Ken Shoemake [Shoemake 87]. When q_0 and q_1 are viewed as vectors on the unit hypersphere in four dimensions, SLERP is a constant-speed parameterization of the shortest arc connecting the vectors. If the vectors are antipodal, say, $q_1 = -q_0$, then SLERP is not defined because there are infinitely many shortest arcs that connect the vectors.

ISSN: 2151-237X print/2151-2272 online

The mathematical equation for SLERP is

$$S(t; q_0, q_1) = \frac{\sin((1-t)\theta)}{\sin\theta} q_0 + \frac{\sin(t\theta)}{\sin\theta} q_1, \tag{1}$$

where $\theta \in [0, \pi]$ is the angle between q_0 and q_1 and where $t \in [0, 1]$. The angle between the vectors is determined by $\cos\theta = q_0 \cdot q_1$, where the right-hand side is the dot product applied to 4-tuples.

Implementers of Equation (1) must deal with three issues. First, if the quaternions are equal, the angle between them is zero, in which case SLERP has a divide by zero. The same divide-by-zero issue occurs when the quaternions are antipodal, but this is not a problem because SLERP is not defined for antipodal points. Second, quaternions represent rotations but provide a double covering: q and $-q$ represent the same rotation. Implementers will avoid the antipodal points and the double-covering issue by requiring the angle between the quaternions to be acute. Third, the function acos is applied to compute θ from $q_0 \cdot q_1 = \cos\theta$. When using floating-point arithmetic, round-off errors can lead to a dot product slightly larger than one. This condition must be trapped and handled, because the acos function returns a quiet NaN for arguments larger than one.

A typical implementation is

```
Quaternion Slerp (Real t, Quaternion q0, Quaternion q1)
{
  Real cosTheta = Dot (q0, q1);
  if (cosTheta < 0)
  {
    q1 = -q1;
    cosTheta = -cosTheta;
  }
  if (cosTheta < 1)
  {
    // Angle theta is in the interval (0, pi/2].
    Real theta = acos (cosTheta);
    Real invSinTheta = 1/sin (theta);
    Real c0 = sin ((1 - t) * theta) * invSinTheta;
    Real c1 = sin (t * theta) * invSinTheta;
    return c0 * q0 + c1 * q1;
  }
  else
  {
    // Angle theta is zero, so just return one of the inputs.
    return q0;
  }
}
```

SLERP is an expensive function to compute because of one call to acos, three calls to sin, and one division. Moreover, branching is potentially expensive, although on modern CPUs, branching tables for branch prediction eliminate much of that expense. On single instruction, multiple data (SIMD) hardware, the if-then-else processing is replaced by selection, which adds some overhead cost, although the gain from parallelism offsets this.

One way to avoid the expensive function calls is to approximate sin by a polynomial and acos by a square root and a polynomial [Abramowitz and Stegun 65]. However, a simple observation about the coefficients used in SLERP and some basic ideas from linear differential equations lead to a faster SLERP evaluation—one that uses only multiplications and additions and has no branching. For applications that make heavy use of SLERP, the approximation provides a decent approximation with significantly improved performance. Source code is available online at the address listed at the end of this paper.

2. Chebyshev Polynomials

The coefficient $\sin(t\theta)/\sin(\theta)$ in Equation (1) is evaluated for $t \in [0, 1]$. It has the same form as $\sin(n\theta)/\sin(\theta)$ for nonnegative integers n, an expression that is related to *Chebyshev polynomials of the second kind*, $u_n(x)$, defined for $|x| \leq 1$ [Wikipedia 2011a]. The polynomials are defined recursively by

$$u_0(x) = 1, \;\; u_1(x) = 2x, \;\; u_n(x) = 2xu_{n-1}(x) - u_{n-2}(x) \quad \text{for } n \geq 2, \tag{2}$$

and have the property

$$u_{n-1}(\cos(\theta)) = \sin(n\theta)/\sin(\theta). \tag{3}$$

They are solutions to the second-order linear differential equation

$$(x^2 - 1)\, u''_{n-1}(x) + 3xu'_{n-1}(x) + (1 - n^2)\, u_{n-1}(x) = 0, \tag{4}$$

where $x = \cos(\theta) \in [0, 1]$ for angles $\theta \in [0, \pi/2]$.

Equation (4) allows for a continuous variable t rather than the discrete variable n, so if we define $u_{t-1}(\cos(\theta)) = \sin(t\theta)/\sin(\theta)$ for real-valued $t \in [0, 1]$, the SLERP equation is rewritten as

$$S(t; q_0, q_1) = u_{-t}(\cos(\theta))\, q_0 + u_{t-1}(\cos(\theta))\, q_1. \tag{5}$$

Equation (5) suggests that we can construct formulas for u_{-t} and u_{1-t} that depend only on $\cos(\theta) = q_0 \cdot q_1$, thereby avoiding the explicit computation of θ and the calls to the sine function.

3. Power Series Solutions of Differential Equations

Define $f(x;t) = u_{t-1}(x)$, which is viewed as a function of x for a specified real-valued parameter t. It is a solution to Equation (4) with n formally replaced by t,

$$(x^2-1)\,f''(x;t) + 3xf'(x;t) + (1-t^2)\,f(x;t) = 0. \tag{6}$$

The prime symbols denote differentiation with respect to x.

We may specify an initial value for $f(1;\,t)$ at the endpoint $x = 1$. Obtaining a unique solution to a linear second-order differential equation normally requires specifying the derivative value at $x = 1$; however, the equation is singular at $x = 1$, because the coefficient of f'' is 0 at $x = 1$. The uniqueness is guaranteed by specifying only the value of f at the endpoint. When x is 1, θ is 0 and evaluation of $u_{t-1}(0)$ is, in the limiting sense,

$$u_{t-1}(1) = \lim_{\theta\to 0} u_{t-1}(\cos(\theta)) = \lim_{\theta\to 0} \frac{\sin(t\theta)}{\sin(\theta)} = \lim_{\theta\to 0} \frac{t\cos(t\theta)}{\cos(\theta)} = t. \tag{7}$$

The next-to-last equality of Equation (7) uses an application of l'Hôpital's Rule. The initial condition is therefore $f(1;\,t) = t$.

A standard undergraduate course on differential equations shows how to solve the differential equation using power series [Braun 92]. Because we want an expansion at $x = 1$, the powers are $(x - 1)^i$. The next equation lists power series for f and its first- and second-order derivatives with respect to x,

$$f = \sum_{i=0}^{\infty} a_i(x-1)^i, \quad f' = \sum_{i=0}^{\infty} i a_i(x-1)^{i-1}, \quad f'' = \sum_{i=0}^{\infty} i(i-1)a_i(x-1)^{i-2}. \tag{8}$$

The coefficients of the powers of $(x - 1)$ are written to show their functional dependence on t. Substituting these into Equation (6),

$$\begin{aligned} 0 &= (x^2-1)\,f'' + 3xf' + (1-t^2)\,f \\ &= [(x-1)^2 + 2(x-1)]\,f'' + 3\,[(x-1)+1]\,f' + (1-t^2)\,f \\ &= \textstyle\sum_{i=0}^{\infty} [(i+1)(2i+3)a_{i+1} + ((i+1)^2 - t^2))a_i]\,(x-1)^i. \end{aligned} \tag{9}$$

For the power series to be identically zero for all x, it is necessary that the coefficients are all zero. The condition $f(1;\,t) = t$ implies $a_0 = t$. These lead to a recurrence equation with initial condition,

$$a_0 = t, \;\; a_i = \frac{t^2 - i^2}{i(2i+1)} a_{i-1}, \;\; i \geq 1. \tag{10}$$

It is apparent from Equation (10) that $a_i(t)$ is a polynomial in t with degree $2i + 1$. Observe that $a_0(0) = 0$, which implies $a_i(0) = 0$ for $i \geq 0$; this is equivalent to $f(x;0) = 0$. Similarly, $a_0(1) = 1$ and $a_1(1) = 0$, which implies $a_i(1) = 0$ for $i \geq 1$; this is equivalent to $f(x;1) = 1$.

4. Convergence of the Series and Bounds on the Truncation Error

We now have a power series for $f(x;t)$, using powers of $(x - 1)$, and whose coefficients are polynomials in t that may be generated iteratively. We must determine how accurate is an approximation when the power series is truncated,

$$\begin{aligned} f(x;t) &= \sum_{i=0}^{n} a_i(t)(x-1)^i + \sum_{i=n+1}^{\infty} a_i(t)(x-1)^i \\ &= \sum_{i=0}^{n} a_i(t)(x-1)^i + \varepsilon(x;t,n) \;, \end{aligned} \tag{11}$$

where $\varepsilon(x;\, t,\, n)$ is the error of truncation. The graph of the truncation error is shown in Figure 1 for the case $n = 8$. By previous observation about $a_i(0)$ and $a_i(1)$, notice that $\varepsilon(x;0,n) = 0$ and $\varepsilon(x;1,n) = 0$. It is also true that $\varepsilon(1;t,n) = 0$.

For $0 < t < 1$, the coefficients from Equation (10) alternate in sign with $a_0 = t > 0$, $a_1 = t(t^2 - 1)/3 < 0$, $a_2 = t(t^2 - 1)(t^2 - 4)/30 > 0$, and so on. When $x \in [0,\, 1]$, the angle between quaternions is acute and the terms $(x - 1)^i$ alternate in sign. Thus, the terms $a_i(t)(x - 1)^i$ are all positive for $t \in (0,\, 1)$ and for $x \in [0,\, 1)$. This implies that the truncation error is nonnegative and a decreasing function of x for each selected t, which is apparent in Figure (1). It is sufficient to analyze the maximum error of truncation, which occurs when $x = 0$,

$$f(0;t) = \sin(\pi t/2) = \sum_{i=0}^{n} (-1)^i a_i(t) + g(t), \tag{12}$$

where the last equality defines the function $g(t) = \varepsilon(0;t,n) = \sum_{i=n+1}^{\infty} (-1)^i a_i(t)$.

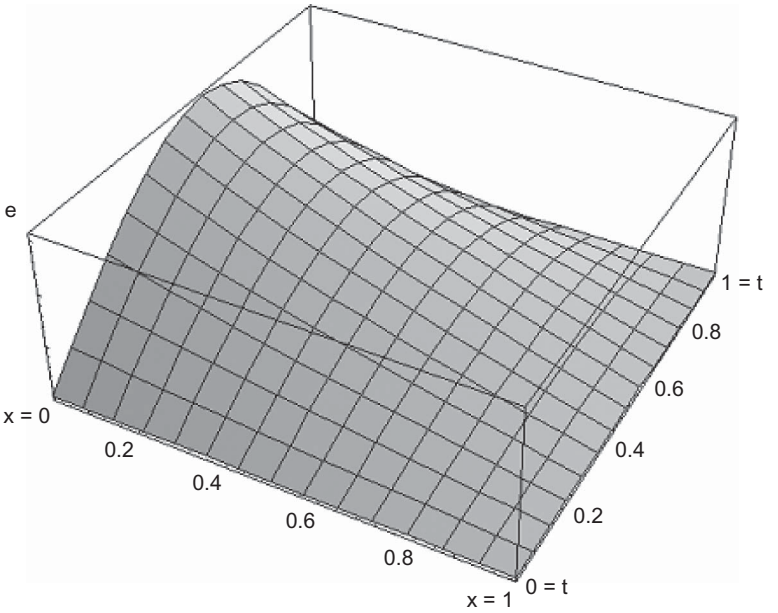

Figure 1. The graph of the truncation error $\epsilon(x;\ t,\ n)$ for $n = 8$. Notice that $\varepsilon(x; 0, n) = 0$, $\varepsilon(x; 1, n) = 0$, and $\varepsilon(1; t, n) = 0$. The rendering is courtesy of Wolfram Research, Inc. [Wolfram 93], with variable names added manually.

Define $\lambda_i(t) = -a_i(t)/a_{i-1}(t) = (i^2 - t^2)/[i(2i + 1)]$, where the last equality follows from Equation (10). It may be shown that for a specified t, $\lambda_i(t)$ is a strictly increasing sequence in i with $\lim_{i\to\infty} \lambda_i(t) = 1/2$. Therefore,

$$0 < \lambda_n(t) < \lambda_i(t) = \frac{(-1)^i a_i(t)}{(-1)^{i-1} a_{i-1}(t)} < \frac{1}{2} \tag{13}$$

for all $i > n$. We know that $(-1)^{i-1} a_{i-1}(t) > 0$ for all i, so

$$\lambda_n(t)(-1)^{i-1} a_{i-1}(t) < (-1)^i a_i(t) < \frac{1}{2}(-1)^{i-1} a_{i-1}(t). \tag{14}$$

Summing for $i \geq n + 1$, we have

$$\lambda_n(t) \sum_{i=n+1}^{\infty} (-1)^{i-1} a_{i-1}(t) < \sum_{i=n+1}^{\infty} (-1)^i a_i(t) < \frac{1}{2} \sum_{i=n+1}^{\infty} (-1)^{i-1} a_{i-1}(t), \tag{15}$$

or equivalently,

$$\lambda_n(t)\left((-1)^n a_n(t) + g(t)\right) < g(t) < \frac{1}{2}\left((-1)^n a_n(t) + g(t)\right). \tag{16}$$

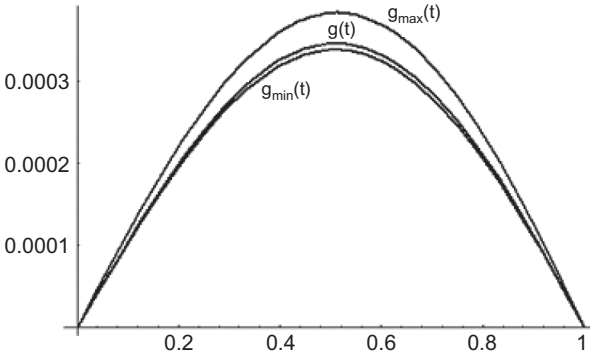

Figure 2. Plots of the bounding functions and the truncation error function for $n = 8$. The rendering is courtesy of Wolfram Research, Inc. [Wolfram 93], with annotations added manually.

Solving each of the two inequalities for $g(t)$, we have

$$g_{\min}(t) = \left(\frac{\lambda_n(t)}{1-\lambda_n(t)}\right)(-1)^n a_n(t) < g(t) < (-1)^n a_n(t) = g_{\max}(t), \qquad (17)$$

where the first and last equalities define the bounding functions $g_{\min}(t)$ and $g_{\max}(t)$. Figure 2 shows plots of the bounding functions and the truncation error function for $n = 8$.

The truncation error $g(t)$ is a transcendental function, because it is the difference of a sinusoidal and a polynomial. For fast computation, we want to approximate $g(t)$ by a polynomial. Both $g_{\min}(t)$ and $g_{\max}(t)$ are polynomials, making them potential candidates. Figure 2 shows that $g_{\min}(t)$ underestimates $g(t)$ and that $g_{\max}(t)$ overestimates $g(t)$. If we had to choose one or the other, $g_{\min}(t)$ appears to be closer to $g(t)$ than $g_{\max}(t)$. However, a better approximating polynomial is one for which some errors are positive and some are negative, but also where the maximum error is about half that when the errors are all of one sign.

To motivate such an approximation, consider the case $n = 8$. Figure 3 shows a plot of the bounding functions and truncation error function divided by $a_8(t)$. The grey line in the figure is the graph of a constant function μ for which some values of $g(t)/a_8(t)$ are larger than μ and some are smaller than μ. We want to choose μ to provide a *balance of errors*, so to speak. Choosing μ so that $g(t) - \mu$ is always positive or is always negative is not what one would intuitively call balance. One might try to provide balance by choosing μ to be the average of $g(t)/a_8(t)$ for $t \in [0, 1]$. The average is computed as $\mu = \int_0^1 g(t)/a_8(t)\,dt$. However, we are interested in approximating $g(t)$ rather than

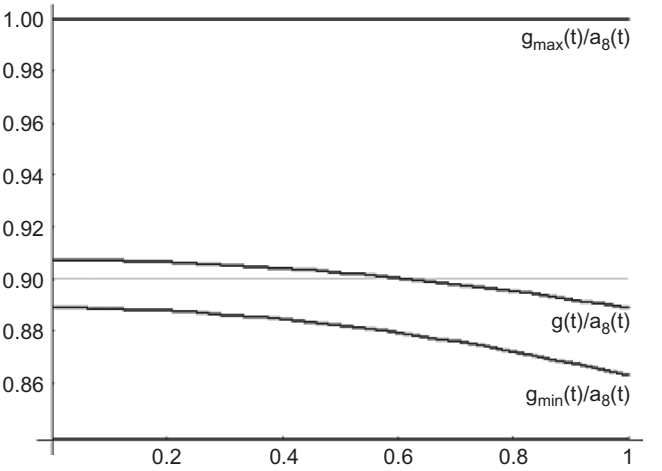

Figure 3. Plots of the bounding functions and the truncation error function for $n = 8$, but divided by $a_8(t)$. Observe that $g_{\max}(t)/a_8(t) = 1$ and $g_{\min}(t)/a_8(t) = \lambda_8(t)/(1 - \lambda_8(t))$. The grey line is the graph of a constant function μ for which some values of $g(t)/a_8(t)$ are larger than μ and some are smaller than μ. The rendering is courtesy of Wolfram Research, Inc. [Wolfram 93], with annotations added manually.

$g(t)/a_8(t)$, which suggests choosing an approximating polynomial of the form $\mu a_8(t)$. In this form, we must decide on what balancing means and then select μ to obtain balance. The natural choice is to minimize $|g(t) - \mu a_8(t)|$ over all values of t. The Chebyshev Equioscillation Theorem and the associated Remez Algorithm are the standard tools for such a minimization. The algorithm is described very briefly at Wikipedia [Wikipedia 2011b], but a much more detailed description is provided by Carl Pearson [Pearson 85].

For the general problem, define $p(t) = (-1)^n a_n(t)$. We want to choose μ to minimize $|g(t) - \mu p(t)|$ over all values of t. The Remez Algorithm is used to compute two numbers t_0 and t_1 in (0, 1) so that $g(t_0) - \mu p(t_0) = +e$, $g'(t_0) - \mu p'(t_0) = 0$, $g(t_1) - \mu p(t_1) = -e$, and $g'(t_1) - \mu p'(t_1) = 0$, where $e > 0$ is the extreme value. The pseudocode to determine μ and e follows:

```
t0 = 0.25;  t1 = 0.75;
do until convergence
{
  g0 = g(t0);  p0 = p(t0);  g1 = g(t1);  p1 = p(t1);
  u = (g0 + g1)/(p0 + p1);
  e = (g0 * p1 - g1 * p0)/(p0 + p1);
  solve g' (t) - u * p' (t) = 0 for t0 in [0,1/2];
  solve g' (t) - u * p' (t) = 0 for t1 in [1/2,1];
}
```

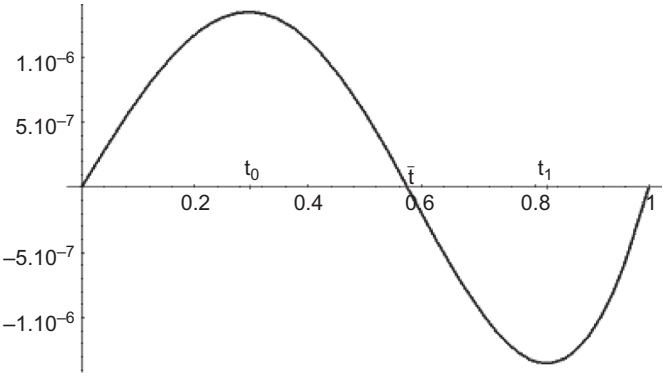

Figure 4. The graph of the error $g(t) - \mu p(t)$ for the computed value of u when $n = 8$. The equioscillation is apparent. The rendering is courtesy of Wolfram Research, Inc. [Wolfram 93], with annotations added manually.

Initial values are provided for t_0 and t_1. The equations $\mu p(t_0) + e = g(t_0)$ and $\mu p(t_1) - e = g(t_1)$ are solved for μ and e. New values for t_0 and t_1 are computed as the roots of $g'(t_0) - \mu p'(t_0) = 0$ and $g'(t_1) - \mu p'(t_1) = 0$. The process is repeated until the computed t_0 and t_1 converge, say, until their computed values do not change significantly from those of the last iteration.

Figure 4 shows a graph of the balanced error function when $n = 8$. The graph indicates that the minimum error of zero occurs when $t = 0$, $t = 1$, and at some value $\bar{t} \in (0, 1)$ which is not $1/2$, because the graph is not symmetric about $t = 1/2$. The maximum error e occurs at two values t_0 and t_1 in the interval (0, 1).

Table 1 shows the output of the Remez Algorithm for several values of n. The choice of n will be tied to how much error you want to allow in SLERP, which in turn relates to the corresponding e in the table. The value μ in the table is required for the implementation of the approximate SLERP and depends on your choice of n. The table values t_0 and t_1 are provided only to show all the outputs of the Remez Algorithm, but these are not needed in an implementation.

To summarize, if μ_n and e_n are the outputs of the Remez Algorithm for the specified n, then $|g(t) - \mu_n p(t)| \le e_n$ for all t. The approximation to $f(x;t) = \sin(t\theta)/\sin(\theta)$ with $x = \cos(\theta)$ is

$$\hat{f}(x;t) = \sum_{i=0}^{n} a_i(t)(x-1)^i + \mu a_n(t)(x-1)^n, \tag{18}$$

where the approximation error bound is $|f(x;t) - \hat{f}(x;t)| \le e_n$.

n	μ	e	t_0	t_1
1	0.62943436108234530	5.745259×10^{-3}	0.3214	0.8477
2	0.73965850021313961	1.092666×10^{-3}	0.3080	0.8344
3	0.79701067629566813	2.809387×10^{-4}	0.3026	0.8288
4	0.83291820510335812	8.409177×10^{-5}	0.2998	0.8258
5	0.85772477879039977	2.763477×10^{-5}	0.2980	0.8238
6	0.87596835698904785	9.678992×10^{-6}	0.2967	0.8224
7	0.88998444919711206	3.551215×10^{-6}	0.2958	0.8214
8	0.90110745351730037	1.349968×10^{-6}	0.2952	0.8206
9	0.91015881189952352	5.277561×10^{-7}	0.2946	0.8200
10	0.91767344933047190	2.110597×10^{-7}	0.2942	0.8195
11	0.92401541194159076	8.600881×10^{-8}	0.2938	0.8191
12	0.92944142668012797	3.560875×10^{-8}	0.2935	0.8188
13	0.93413793373091059	1.494321×10^{-8}	0.2932	0.8184
14	0.93824371262559758	6.344653×10^{-9}	0.2930	0.8182
15	0.94186426368404708	2.721482×10^{-9}	0.2928	0.8180
16	0.94508125972497303	1.177902×10^{-9}	0.2926	0.8178

Table 1. The output of the Remez Algorithm for several values of n.

This relates to SLERP in the following way: Define $f_0 = f(x; 1-t)$ and $f_1 = f(x; t)$ for a specified $x = \cos(\theta)$ with $\theta \in [0, \pi/2]$ and for a specified $t \in [0, 1]$. If q_0 and q_1 are quaternions, then their SLERP is $s = f_0 q_0 + f_1 q_1$. Let $\hat{f}_i$ be the approximations to f_i using Equation (18); then $\hat{f}_i = f_i + \delta_i$ for some errors δ_i with $|\delta_i| \le e_n$. The SLERP using these coefficients is $\hat{s} = \hat{f}_0 q_0 + \hat{f}_1 q_1$. A measure of error is the length ℓ of the difference of the quaternions treated as 4-tuple vectors,

$$\begin{aligned} \ell^2 = \; & |\delta_0 q_0 + \delta_1 q_1|^2 = \delta_0^2 + 2\delta_0\delta_1 q_0 \cdot q_1 + \delta_1^2 \\ \le \; & \delta_0^2 + 2|\delta_0||\delta_1| + \delta_0^2 = (|\delta_0| + |\delta_1|)^2 \le (2e_n)^2, \end{aligned} \tag{19}$$

where we have used $|q_0 \cdot q_1| = |\cos\theta| \le 1$. Therefore, $\ell \le 2e_n$. If you wish to have no more than error $\varepsilon > 0$ for ℓ, require $\ell \le 2e_n \le \varepsilon$. Table 1 may be used to select the appropriate n to ensure $e_n \le \varepsilon/2$.

5. Computing the Approximation

Naturally, we wish to compute the approximation of Equation (18) as efficiently as possible. To illustrate, consider the case $n = 2$. Using Equation (10) for the coefficients, the approximation is

$$\begin{aligned}\hat{f}(x;t) &= a_0(t) + a_1(t)(x-1) + (1+\mu)a_2(t)(x-1)^2 \\ &= t\left(1 + \left(\frac{(t^2-1)(x-1)}{3}\right) + (1+\mu)\left(\frac{(t^2-1)(x-1)}{3}\right)\left(\frac{(t^2-4)(x-1)}{10}\right)\right) \\ &= t\left(1 + b_0 + (1+\mu)b_0 b_1\right) \\ &= t\left(1 + b_0\left(1 + (1+\mu)b_1\right)\right), \end{aligned} \tag{20}$$

where $b_0 = (x-1)(t^2-1)/3$ and $b_1 = (x-1)(t^2-4)/10$. For $n = 3$, the approximation is $\hat{f}(x;t) = t(1 + b_0(1 + b_1(1 + (1+\mu)b_2)))$ for b_0 and b_1 as in the case $n = 2$ and for $b_2 = (x-1)(t^3-9)/21$. For general n, the nested expression is similar,

$$b_i(x,t) = (x-1)\left(\frac{t^2-(i+1)^2}{(i+1)(2i+3)}\right) = (x-1)(u_i t^2 - v_i), \tag{21}$$

for $1 \le i \le n$. The last equality defines the constants u_i and v_i; these are precomputed in an implementation.

5.1. *Implementation for the Floating-Point Unit*

An implementation for the floating-point unit (FPU) is presented for $n = 8$ so that the approximate SLERP values have errors on the order of 10^{-6} when using 32-bit floating-point numbers. The code assumes a 4-tuple class `FTuple4` that has an addition operator, a scalar-multiplication operator, and a dot-product function. We need to evaluate $\sin(t\theta)/\sin(\theta)$ and $\sin((1-t)\theta)/\sin(\theta)$, so the code computes the approximation for both t and $1 - t$. Finally, the $(1 + \mu)$ term is absorbed by the coefficients u_{n-1} and v_{n-1}.

```
// Precomputed constants.
const float opmu = 1.90110745351730037f;
const float u[8] = // 1/[i(2i + 1)] for i >= 1
{
  1.f/(1 * 3), 1.f/(2 * 5), 1.f/(3 * 7), 1.f/(4 * 9),
  1.f/(5 * 11), 1.f/(6 * 13), 1.f/(7 * 15), opmu/(8 * 17)
} ;
const float v[8] = // i /(2i+ 1) for i >= 1
{
  1.f/3, 2.f/5, 3.f/7, 4.f/9,
  5.f/11, 6.f/13, 7.f/15, opmu * 8/17
} ;
```

```
FTuple4 SlerpFPU (float t, FTuple4 q0, FTuple4 q1)
{
  float x = q0 . Dot (q1); // cos (theta)
  float sign = (x >= 0 ? 1 : (x = -x, -1));
  float xm1 = x - 1;
  float d = 1 - t, sqrT = t * t, sqrD = d * d;
  float bT[8], bD[8];
  for (int i = 7; i >= 0; --i)
  {
    bT[i] = (u[i] * sqrT - v[i]) * xm1;
    bD[i] = (u[i] * sqrD - v[i]) * xm1;
  }
  float cT = sign * t *(
    1 + bT[0] * (1 + bT[1] * (1 + bT[2] * (1 + bT[3] * (
    1 + bT[4] * (1 + bT[5] * (1 + bT[6] * (1 + bT [7] ))))))));
  float cD = d * (
    1 + bD[0] * (1 + bD[1] * (1 + bD[2] * (1 + bD[3] * (
    1 + bD[4] * (1 + bD[5] * (1 + bD[6] * (1 + bD[7] ))))))));
  FTuple4 slerp = q0 * cD + q1 * cT;
  return slerp;
}
```

Observe that only multiplications and additions are used—no transcendental function calls, no divisions, and no branching.

5.2. *Implementation for SIMD*

The choice $n = 8$ is convenient for a fast SIMD implementation. The source code presented next is for an Intel® processor with streaming SIMD extension (SSE) support. An implementation for the Xbox360® implementation is similar, although you can use an intrinsic for the dot product and fused-multiply-add instructions for the evaluation of the nested expression for the approximation.

```
// Precomputed constants.
const float opmu = 1.90110745351730037 f;
const __m128 u0123 =
  _mm_setr_ps (1 . f /(1*3), 1 . f /(2*5), 1 . f /(3*7), 1 . f / (4*9));
const __m128 u4567 =
  _mm_setr_ps (1 . f /(5 * 11), 1 . f /(6 * 13), 1 . f /(7 * 15), opmu/ (8
  * 17));
const __m128 v0123 =
  _mm_setr_ps (1 . f /3, 2 . f /5, 3 . f /7, 4 . f / 9);
```

```
const __m128 v4567 =
  _mm_setr_ps (5.f/11, 6.f/13, 7.f/15, opmu*8/17);
const __m128 signMask = _mm_set1_ps (-0.f);
const __m128 one = _mm_set1_ps (1.f);
// Dot product of 4-tuples.
__m128 Dot (const __m128 tuple0, const __m128 tuple1)
{
  __m128 t0 = _mm_mul_ps (tuple0, tuple1);
  __m128 t1 = _mm_shuffle_ps (tuple1, t0, _MM_SHUFFLE(1, 0, 0, 0));
  t1 = _mm_add_ps (t0, t1);
  t0 = _mm_shuffle_ps (t0, t1, _MM_SHUFFLE(2, 0, 0, 0));
  t0 = _mm_add_ps (t0, t1);
  t0 = _mm_shuffle_ps (t0, t0, _MM_SHUFFLE(3, 3, 3, 3));
  return t0;
}
// Common code for computing the scalar coefficients of SLERP.
__m128 Coefficient1 (const __m128 t, const __m128 xm1)
{
  __m128 sqrT = _mm_mul_ps (t, t);
  __m128 b0123, b4567, b, c;
  // (b4, b5, b6, b7) =
  // (x - 1) * (u4 * t^2 - v4, u5 * t^2 - v5, u6 * t^2 - v6,
  u7 * t^2 - v7)
  b4567 = _mm_mul_ps (u4567, sqrT);
  b4567 = _mm_sub_ps (b4567, v4567);
  b4567 = _mm_mul_ps (b4567, xm1);
  // (b7, b7, b7, b7)
  b = _mm_shuffle_ps (b4567, b4567, _MM_SHUFFLE(3, 3, 3, 3));
  c = _mm_add_ps (b, one);
  // (b6, b6, b6, b6)
  b = _mm_shuffle_ps (b4567, b4567, _MM_SHUFFLE(2, 2, 2, 2));
  c = _mm_mul_ps (b, c);
  c = _mm_add_ps (one, c);
  // (b5, b5, b5, b5)
  b = _mm_shuffle_ps (b4567, b4567, _MM_SHUFFLE(1, 1, 1, 1));
  c = _mm_mul_ps (b, c);
  c = _mm_add_ps (one, c);
  // (b4, b4, b4, b4)
  b = _mm_shuffle_ps (b4567, b4567, _MM_SHUFFLE(0, 0, 0, 0));
  c = _mm_mul_ps (b, c);
  c = _mm_add_ps (one, c);
  // (b0, b1, b2, b3) =
  // (x-1)*(u0*t^2-v0, u1*t^2-v1, u2*t^2-v2, u3*t^2-v3)
```

```
    b0123 = _mm_mul_ps (u0123, sqrT);
    b0123 = _mm_sub_ps (b0123, v0123);
    b0123 = _mm_mul_ps (b0123, xm1);
    // (b3, b3, b3, b3)
    b = _mm_shuffle_ps (b0123, b0123, _MM_SHUFFLE(3, 3, 3, 3));
    c = _mm_mul_ps (b, c);
    c = _mm_add_ps (one, c);
    // (b2, b2, b2, b2)
    b = _mm_shuffle_ps (b0123, b0123, _MM_SHUFFLE(2, 2, 2, 2));
    c = _mm_mul_ps (b, c);
    c = _mm_add_ps (one, c);
    // (b1, b1, b1, b1)
    b = _mm_shuffle_ps (b0123, b0123, _MM_SHUFFLE(1, 1, 1, 1));
    c = _mm_mul_ps (b, c);
    c = _mm_add_ps (one, c);
    // (b0, b0, b0, b0)
    b = _mm_shuffle_ps (b0123, b0123, _MM_SHUFFLE(0, 0, 0, 0));
    c = _mm_mul_ps (b, c);
    c = _mm_add_ps (one, c);
    c = _mm_mul_ps (t, c);
    return c;
  }
  void SlerpSSE1 (float t [1], const __m128 q0, const__m128 q1,
    __m128 slerp [1])
  {
    __m128 x = Dot (q0, q1); // cos (theta) in all components
    __m128 sign = _mm_and_ps (signMask, x);
    x = _mm_xor_ps (sign, x);
    __m128 localQ1 = _mm_xor_ps (sign, q1);
    __m128 xm1 = _mm_sub_ps (x, one);
    __m128 splatT = _mm_set1_ps (t);
    __m128 splatD = _mm_sub_ps (one, splatT);
    __m128 cT = Coefficient (splatT, xm1);
    __m128 cD = Coefficient (splatD, xm1);
    cT = _mm_mul_ps (cT, localQ1);
    cD = _mm_mul_ps (cD, q0);
    __m128 slerp = _mm_add_ps (cT, cD);
    return slerp;
  }
```

With the use of shuffling, you can process a pair of quaternions for two different times or four different times. The source-code listings are lengthy

Function	32-bit time (total)	32-bit time (per SLERP)
Slerp	55.0	55.0
SlerpFPU	19.8	19.8
SlerpSSE1	6.1	6.1
SlerpSSE2	11.0	5.5
SlerpSSE4	18.0	4.5
	64-bit time (total)	64-bit time (per SLERP)
Slerp	38.0	38.0
SlerpFPU	17.0	17.0
SlerpSSE1	5.4	5.4
SlerpSSE2	9.8	4.9
SlerpSSE4	17.2	4.3

Table 2. Performance of SLERP. The execution times are for 2^{28} function calls. The per-SLERP times are the execution times divided by the number of SLERPs computed per function call.

and are not included here. You may find them at the address listed at the end of this paper.

5.3. Performance Measurements

The performance of the functions `Slerp` (standard implementation), `SlerpFPU, SlerpSSE1, SlerpSSE2`, and `SlerpSSE4` was measured by calling each function in a loop that executed 2^{28} times. The 32-bit times are on an Intel® Core™ 2 Quad CPU Q9550 running at 2.83 GHz. The 64-bit times are on an Intel® Core™ i7 Q740 running at 1.73 GHz. Table 2 shows the execution times, rounded to the nearest tenth of a second. Similar timings were obtained by running the test program using Intel® VTune™ Amplifier XE 2011.

References

[Abramowitz and Stegun 65] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. New York: Dover Publications, Inc., 1965.

[Braun 92] Martin Braun. *Differential Equations and Their Applications : An Introduction to Applied Mathematics*, Texts in Applied Mathematics, Vol. 11, fourth edition, New York: Springer-Verlag, 1992.

[Pearson 85] Carl E. Pearson. *Handbook of Applied Mathematics: Selected Results and Methods*, second edition, New York: Van Nostrand Reinhold, 1985.

[Shoemake 87] Ken Shoemake. Animating Rotation with Quaternion Calculus. *ACM SIGGRAPH '87, Course Notes 10, Computer Animation: 3-D Motion, Specification, and Control.*

[Wikipedia 2011a] Wikipedia. "Chebyshev Polynomials." *Wikipedia*. Available at http://en.wikipedia.org/wiki/Chebyshev_polynomials, 2011.

[Wikipedia 2011b] Wikipedia. "Remez Algorithm." *Wikipedia*. Available at |http://en.wikipedia.org/wiki/Remez_algorithm, 2011.

[Wolfram 93] Wolfram Research, Inc. Mathematica, Version 2.2, Champaign, IL 1993.

Web Information:

Source code is available online at http://www.geometrictools.com/JGT/FastSlerp.cpp/

David Eberly, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA (E-mail: daeberly@microsoft.com)

Received April 3, 2011; accepted in revised form July 18, 2011.