A Solution to Inverse Interpolation in Computer Graphics

Mikael Fridenfalk
Department of Game Design
Uppsala University
Visby, Sweden
mikael.fridenfalk@speldesign.uu.se

***Abstract*—Inverse solutions to interpolation techniques in computer graphics may increase the accuracy of the rendered frames. An example is the use of an arbitrary time stamp in time interpolation to evaluate the t-parameter, which in turn can be used for time-variant spatial interpolation. In this paper, an analytic inverse is presented for a shape-preserving piecewise cubic Hermite interpolant, used in context with camera trajectory interpolation.**

Keywords-LPCHIP; PCHIP; polynomial equation; spline

I. Introduction

A solution was presented for camera trajectory interpolation in large-scale worlds, where previous methods result in oscillations at exponential zooms [4]. A logarithmic version, called LPCHIP, of a shape-preserving piecewise cubic Hermite interpolant (called the PCHIP equivalent, or henceforth in this paper just PCHIP) was introduced. To obtain the interpolation parameter t, used for spatial interpolation, an inverse iterative version of PCHIP, called InvPCHIP, was designed based on Newton's method [3]. This paper presents an analytic solution to this inverse interpolation method.

Presently, piecewise cubic Hermite splines are used for high-end interpolation of the trajectories of cameras and 3D objects in computer graphics [2,7,8], such as computer games, but also for computer-controlled cameras in film production. On an implementation level, the standard method to control a camera in computer graphics is by an object called the target camera [9], defined by camera position, a look-at position and the orientation of the camera around the vector pointing from the position of the camera to the look-at position (called roll). To avoid causing the viewer disorientation or nausea, roll is often set to a constant value.

These spline interpolation techniques are however as a rule not based on shape-preserving ones, such as PCHIP in MATLAB [1,5,6], which could be a better choice for camera trajectory control, since PCHIP eliminates the overshooting effects associated with the regular variant, see Fig. 1 (left), thereby increasing the level of control in camera trajectory generation without any practical downside, see Fig. 1 (right). A reason for this could be that MATLAB, which is the application that introduced PCHIP to a wider audience, is presently not widely used in systems for generation of motion picture.

It should however be noted that the PCHIP variant used in this paper is not identical to PCHIP in MATLAB, but a simplified version. The principal difference is that PCHIP is here designed specifically for a constant step size between the breakpoints (also called *keyframes*). However, by the addition of separate interpolation along the horizontal axis, as shown in Figs. 1-3, the step size between the breakpoints becomes automatically variable.

In Figs. 1-3 (left), the trajectories of two sets of breakpoints are evaluated by a regular piecewise cubic Hermite interpolant. While the implementation of the harmonic mean in Fig. 1 (middle), eliminates the overshooting effects of the regular interpolant, Fig. 2 shows that the harmonic mean does not always work properly, unless the tangents (or slopes) m_1 and m_2 are limited by locally monotonic constraints, see Figs. 1-3 (right).

The main difference between a *regular* and a shape-preserving piecewise cubic Hermite interpolant is that in this context, by definition, the tangents m_1 and m_2 are in the regular interpolant, functions of the mean values of the differences of adjacent breakpoints, while the shape-preserving version is based on locally monotonic functions of the harmonic mean, see Eqs. (1)-(4).

Since PCHIP is locally monotonic, *i.e.* the resulting interpolation curve is increasing or decreasing between each two adjacent breakpoints, see Fig. 3, there is nominally always at most one solution to the inverse function of PCHIP. As a result, for strictly increasing breakpoint values, such as in time interpolation, PCHIP is a strictly increasing function. Another benefit shown in [4], was the feasibility of PCHIP for use in logarithmic interpolation.

As a note on logarithmic interpolation, a camera trajectory control system was developed during the NASA International Space Apps Challenge 2013, for the production of a video within the Ad Infinitum project on the challenge *Why We Explore* [4]. Although the control system worked perfectly well within local room dimensions, yet the exponential zoom from microcosm to macrocosm showed to work less than satisfactory due to an uneven change of the experienced zooming speed. This effect is demonstrated in Figs. 4-6 by the dotted curves. In Fig. 4, the effect is best shown using a regular cubic Hermite interpolant with six breakpoints defined by the function 3.146^x, which was the largest base with three decimals that could be used before the interpolant caused a singularity. In this context, x represents the linear horizontal axis in Figs. 4-6. As shown in Figs. 5-6 (dotted curves), where Fig. 6 displays a zoom using the function 10^{16x}, PCHIP does not cause any singularities and is thus a feasible candidate for logarithmic interpolation.

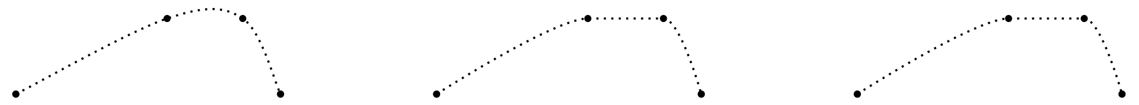

Figure 1: Regular (left), harmonic (middle), harmonic and monotonic (right). As shown in this example, the regular interpolant causes a slight overshoot between the second and the third breakpoints.

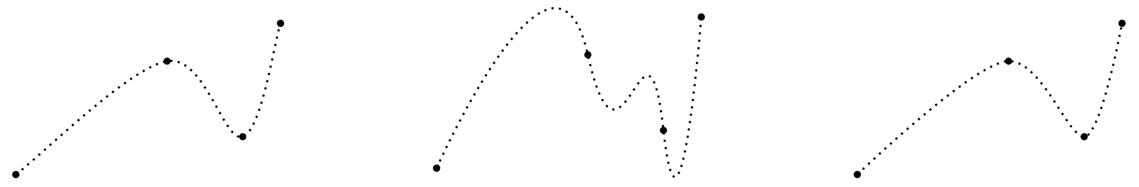

Figure 2: Regular (left), harmonic (middle), harmonic and monotonic (right). While harmonic interpolation solves the overshooting problem of the example in the previous figure, to work properly for all cases, it has to be monotonic.

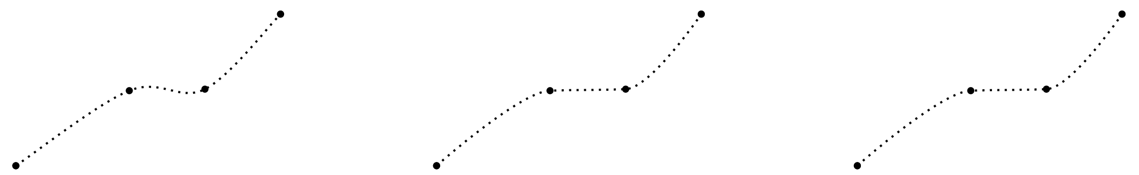

Figure 3: Regular (left), harmonic (middle), harmonic and monotonic (right). As shown here, although the second breakpoint (from left) at regular interpolation has a slightly lower value than the third, there is still a slight overshoot between these breakpoints, which makes it unsuited for time interpolation.

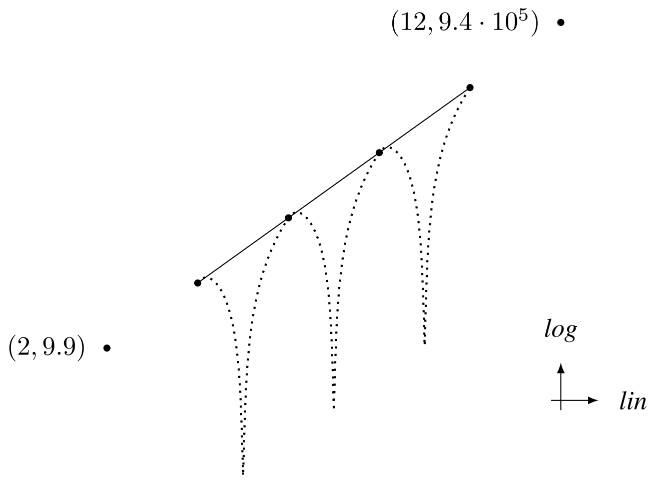

Figure 4: Regular interpolation (dotted) versus LPCHIP (solid) in an even exponential zoom

II. Implementation

In [4], InvPCHIP was evaluated iteratively by Newton's method. A shape-preserving interpolation method is here defined as one that is both harmonic and monotonic. Given the parameters $x_0 - x_3$ and t, the (harmonic) tangents m_k are for $k \in \{1, 2\}$ calculated by:

$$d_k = x_{k+1} - x_k \tag{1}$$

$$\Delta_k = \frac{1}{d_{k-1}} + \frac{1}{d_k} \tag{2}$$

$$m_k = \frac{2}{\Delta_k} \tag{3}$$

with the exception that m_k is set to zero, if either d_{k-1}, d_k or Δ is close to zero. In addition, if the interpolation method is monotonic, m_k is set to zero, if:

$$d_{k-1} d_k < 0 \tag{4}$$

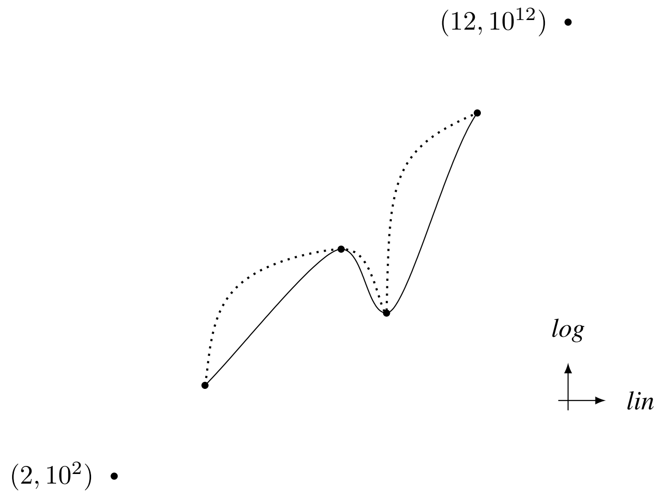

Figure 5: PCHIP (dotted) versus LPCHIP (solid) in a dynamic exponential zoom

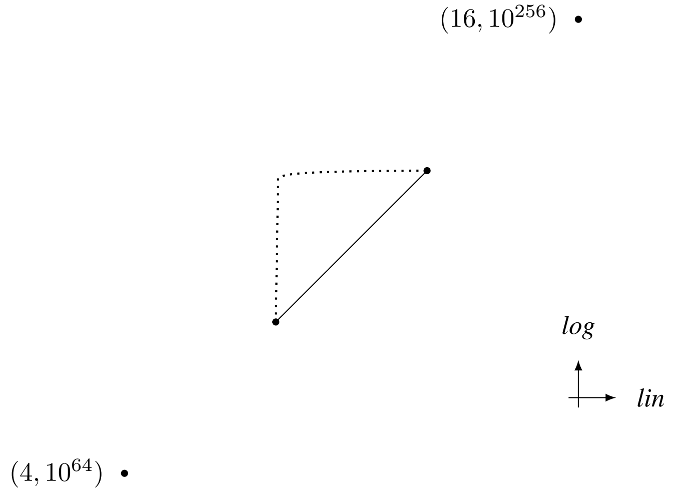

Figure 6: PCHIP (dotted) versus LPCHIP (solid) in a large-scale exponential zoom

The final result is calculated by:

$$h_1 = 2t^3 - 3t^2 + 1 \tag{5}$$
$$h_2 = -2t^3 + 3t^2 \tag{6}$$
$$h_3 = t^3 - 2t^2 + t \tag{7}$$
$$h_4 = t^3 - t^2 \tag{8}$$

$$y = h_1x_1 + h_2x_2 + h_3m_1 + h_4m_2 \tag{9}$$

Given the equations above, the interpolation parameter t may be evaluated by the solution of $f(t) = 0$, using Newton's method:

$$t_{n+1} = t_i - \frac{f(t_n)}{f'(t_n)} \tag{10}$$

with:

$$f(t) = h_1x_1 + h_2x_2 + h_3m_1 + h_4m_2 - y \tag{11}$$

where n denotes each iteration step and $f'(t)$, the derivative of f with respect to t (and in the case of InvPCHIP, with a start value of $t_0 = 0.5$). The derivation of an analytic solution for inverse PCHIP includes thus the solution of the following third degree polynomial equation:

$$f(t) = At^3 + Bt^2 + Ct + D = 0 \tag{12}$$

with:

$$A = 2(x_1 - x_2) + m_1 + m_2 \tag{13}$$
$$B = 3(x_2 - x_1) - 2m_1 - m_2 \tag{14}$$
$$C = m_1 \tag{15}$$
$$D = x_1 - y \tag{16}$$

The general solution to a third degree polynomial equation was in this work derived using the Wolfram Alpha online symbolic calculator [10]. Further simplification of the rendered formulas yielded the following solution for the three roots:

$$a_0 = 9ABC - 27A^2D - 2B^3 \tag{17}$$

$$a_1 = 3AC - B^2 \tag{18}$$

$$a_2 = \sqrt[3]{2} \tag{19}$$

$$b_0 = a_0^2 + 4a_1^3 \tag{20}$$

$$u_0 = \frac{1}{6a_2A},\ u_1 = \frac{2u_0a_1}{a_2},\ u_2 = -\frac{a_1a_2}{3A} \tag{21}$$

$$u_3 = \left(a_0 + \sqrt{b_0}\right)^{1/3} \tag{22}$$

$$u_4 = -\frac{B}{3A} \tag{23}$$

$$t_1 = 2u_0u_3 + \frac{u_2}{u_3} + u_4 \tag{24}$$

$$t_{2,3} = -u_0u_3(1 \pm i\sqrt{3}) + \frac{u_1}{u_3}(1 \mp i\sqrt{3}) + u_4 \tag{25}$$

A more optimized version of this solver is implemented in C++ by Poly3 (Fig. 10). To handle singularities, if A is sufficiently small, Poly2 will be called by Poly3. If B in turn is sufficiently small, Poly1 will be called by Poly2. The implementation of the methods listed in Fig. 12 is straightforward, but as a clarification note, CX_Mk takes the real and imaginary parts of a complex number and returns a complex number of the data type CMPLX.

III. Results

In a series of experiments, the robustness and the execution speed of the iterative, versus the analytic version of inverse PCHIP were measured. Running the algorithms a billion times each, using one thousand sets of randomized values for $x_0 - x_3$ and t, showed that the iterative method was for an error less than $\delta = 10^{-9}$ (before the iterative loop in Newton's method is ended), approximately 2.0 times as fast as the analytic method. Since both functions showed to execute in typically less than a microsecond on a modern laptop computer (with a single processor dedicated to the calculations), and InvPCHIP is not expected to be called more often than the frame update frequency (which is typically less than 100 frames per second for a modern computer game), the execution time of InvPCHIP is in practice considered to be negligible.

A robustness test was performed by the execution of both algorithms a million times each, using randomized parameter settings for each call to evaluate $\epsilon = |t - t_0|$, based on:

$$t = \text{InvPCHIP\_Poly3}(x_0, ..., x_3, \text{PCHIP}(x_0, ..., x_3, t_0)) \tag{26}$$

The result of the experiment was that for the iterative version of inverse PCHIP, $\epsilon > 10^{-9}$ in 17 cases out of a million, of which in two cases, $\epsilon > 10^{-8}$, but in no case was $\epsilon > 10^{-7}$. Regarding the same test for the analytic method, in 10 cases out of a million, $\epsilon > 10^{-16}$, of which in two cases $\epsilon > 10^{-15}$, but there was no case where $\epsilon > 10^{-14}$ (except of course for the special case of $x_1 = x_2$, where by design, the returned value t is set to 0.5). This result was expected, since the floating point operations in the algorithms analyzed in this paper are based on the data type *double* in C++, consisting of a sign bit, 11 bits for the exponent and 52 bits for the fraction, with $2^{52} = 4.50 \cdot 10^{15}$. Further analysis showed that when δ was increased to 10^{-8}, the outcome of the iterative test was increased from 17 to 23, but that a smaller value for δ than 10^{-9} did not lead to any further improvements of the accuracy. Note by the way that the code provided in this paper is only a starting point, intended for use in computer graphics. For hazardous applications (where for instance human safety is at stake), further analysis and improvements of the code may be required, such as the replacement of double with a more accurate data type, accompanied by further robustness analysis.

As a final note, for 10 million randomized parameter values in Eq. (26), in the case where one of the three roots in Poly3 had a feasible solution (*i.e.* Poly2 was not called), t_1 was selected 4% and t_2, 96% of the time, why t_2 is chosen to be solved before t_1 in Poly3. For randomized parameter values however, such that $x_k < x_{k+1}$ for $k \in \{0, 1, 2\}$, t_1 was selected 44% and t_2, 56% of the time. In both cases, t_3 was also selected, but only approximately 0.1% versus 0.4% of the time.

IV. Conclusion

The iterative solution based on Newton's method used in the related paper for the evaluation of inverse PCHIP [4], showed for the required accuracy in our applications, to be in average twice as fast as the analytic version, and in addition much more straightforward to implement. Using the same data type *double*, the analytic solution showed however to be more accurate. Analythical methods are in general very useful, since although in practice also implemented iteratively on the microprocessor level, they may give closer insights into a problem, and at systems integration, enable simplification of the final mathematical results.

References

[1] C. de Boor, K. Höllig, and M. Sabin, "High accuracy geometric Hermite interpolation", *Computer Aided Geometric Design*, vol. 4 (1987), no. 4, pp. 269-278.

[2] M. Christie, P. Olivier, and J. Normand, "Camera control in computer graphics", *Computer Graphics*, vol. 27 (2008), no. 8, pp. 2197-2218.

[3] P. Deuflhard, *Newton methods for nonlinear problems: Affine invariance and adaptive algorithms*, Springer, 2011.

[4] M. Fridenfalk, "Camera trajectory evaluation in computer graphics based on logarithmic interpolation", *Proc. of the Eighth International Conference on Software Engineering Advances*, Venice, Italy, 2013, pp. 551-557.

[5] F. N. Fritch and R. E. Carlson, "Monotone piecewise cubic interpolation", *SIAM Journal on Numerical Analysis*, vol. 17 (1980), no. 2, pp. 238-246.

[6] C. Moler, *Numerical computing with MATLAB*, society for industrial and applied mathematics, 2010.

[7] T. Mullen, *Mastering Blender*, John Wiley & Sons, 2010.

[8] T. Palamar and E. Keller, *Mastering Autodesk Maya 2012*, Sybex, Hoboken, NJ, USA, 2011.

[9] H. Smith, *Foundation 3ds Max 8: Architectural visualization*, Dreamtech Press, 2007.

[10] Wolfram Alpha. [Online]. Available: http://www.wolframalpha.com/

```
CMPLX XPCHIP::CX_Power(CMPLX b, double e){
    double L1 = sqrt(b.x * b.x + b.y * b.y), L2 = pow(L1,e);
    double A1 = atan2(b.y,b.x), A2 = e * A1;
    CMPLX ret; ret.x = L2 * cos(A2); ret.y = L2 * sin(A2);
    return ret;
}
```

Figure 7: Exponentiation of a complex base b with a real exponent e

```
double XPCHIP::Poly1(double C, double D){
    //Solve Ct + D = 0 for 0 <= t <= 1
    if (fabs(C) < 1e-20) return .5;
    double t = -D/C;
    return t >= 0. && t <= 1. ? t : -1.;
}
```

Figure 8: A singularity manager for Poly2

```
double XPCHIP::Poly2(double B, double C, double D){
    //Solve Bt^2 + Ct + D = 0 for 0 <= t <= 1
    if (fabs(B) < 1e-20){return Poly1(C,D);}//Line
    double c1 = 1./(2.*B);
    double c2 = C*C - 4*B*D;
    if (c2 < 0.) return -1.;//Complex
    if (fabs(c2) < 1e-20){//Identical
        double x = - C * c1;
        if (x >= 0. && x <= 1.) return x;
        return -1.;
    }
    double c3 = sqrt(c2);
    double t1 = - c1 * (C - c3);
    if (t1 >= 0. && t1 <= 1.) return t1;
    double t2 = - c1 * (C + c3);
    return t2 >= 0. && t2 <= 1. ? t2 : -1;
}
```

Figure 9: A singularity manager for Poly3

```
double XPCHIP::Poly3(double A, double B, double C, double D){
    //Solve At^3 + Bt^2 + Ct + D = 0 for 0 <= t <= 1
    if (fabs(A) < 1e-20){return Poly2(B,C,D);}
    double a0, a1, a2, a3, b0, e0 = 1e-7, e1 = 1.+e0, u0, u1, u2;
    CMPLX  b1, c1, c2, t1, t2, t3, u3, u3_inv, u4, v1, v2, w1, w2;
    a0 = 9.*A*B*C - 27.*A*A*D - 2.*B*B*B;
    a1 = 3.*A*C - B*B; a2 = pow(2.,1./3.); a3 = sqrt(3.);
    b0 = a0*a0 + 4.*a1*a1*a1;
    b1 = b0 >= 0. ? CX_Mk(a0 + sqrt(b0),0.) : CX_Mk(a0,sqrt(-b0));
    u0 = 1./(6.*a2*A);
    u1 = 2.*u0*a1/a2;
    u2 = - a1*a2/(3.*A);
    u3 = CX_Power(b1,1./3.);
    u3_inv = CX_Inv(u3);
    u4 = CX_Mk(-B/(3.*A),0.);
    v1 = CX_Mk(-u0*u3.x,-u0*u3.y);
    v2 = CX_Mk(u1*u3_inv.x,u1*u3_inv.y);
    w1 = CX_Mk(1.,a3); w2 = CX_Mk(1.,-a3);
    c1 = CX_Mult(v1,w1); c2 = CX_Mult(v2,w2);
    t2 = CX_Add(c1,c2,u4);
    if (fabs(t2.y) < e0 && t2.x >= -e0 && t2.x <= e1) return t2.x;
    c1 = CX_Mk(2.*u0*u3.x,2.*u0*u3.y);
    c2 = CX_Mk(u2*u3_inv.x,u2*u3_inv.y);
    t1 = CX_Add(c1,c2,u4);
    if (fabs(t1.y) < e0 && t1.x >= -e0 && t1.x <= e1) return t1.x;
    c1 = CX_Mult(v1,w2); c2 = CX_Mult(v2,w1);
    t3 = CX_Add(c1,c2,u4);
    return fabs(t3.y) < e0 && t3.x >= -e0 && t3.x <= e1 ? t3.x : -1.;
}
```

Figure 10: A specialized version of an analytic solver for third degree polynomial equations

```
double XPCHIP::InvPCHIP_Poly3(double x0, double x1,
                              double x2, double x3, double y){
    double A, B, C, D, a0, a1, d0, d1, d2, den, eps = 1e-20, m1, m2;
    d0 = x1 - x0; d1 = x2 - x1; d2 = x3 - x2;
    a0 = d0 * d1 < 0.; a1 = d1 * d2 < 0.;
    bool b = fabs(d1) < eps;
    if (a0 || fabs(d0) < eps || b ||
        fabs(den = 1./d0 + 1./d1) < eps) m1 = 0.;
    else m1 = 2./den;
    if (a1 || b || fabs(d2) < eps ||
        fabs(den = 1./d1 + 1./d2) < eps) m2 = 0.;
    else m2 = 2./den;
    A =  2.*x1 - 2.*x2 +    m1 + m2;
    B = -3.*x1 + 3.*x2 - 2.*m1 - m2;
    C = m1;
    D = x1 - y;
    double t = Poly3(A,B,C,D);
    return t < 0. ? 0. : t > 1. ? 1. : t;
}
```

Figure 11: An analytic solution to inverse PCHIP, returning an interpolation parameter $t \in [0, 1]$

```
struct CMPLX {double x; double y;};
...
class XPCHIP ... {
    ...
    CMPLX CX_Mk(double x, double y);
    CMPLX CX_Add(CMPLX u, CMPLX v, CMPLX w);
    CMPLX CX_Inv(CMPLX z);
    CMPLX CX_Mult(CMPLX u, CMPLX v);
};
```

Figure 12: A selection of declarations