COMPARISON OF INTERPOLATION ALGORITHMS IN REAL-TIME SOUND PROCESSING

Porovnání algoritmů interpolace při zpracování v reálném čase

Vladimír Arnošt[*]

Abstract

This paper tries to analyze the most popular interpolation methods and show their speed vs. quality balance. Abstract pseudocode is shown to illustrate their possible implementation and computational requirements.

Introduction

Real-time (RT for short) sound processing has some special requirements than differ from the traditional view of sound processing. The most important factor is *speed*. A discrete real-time system has only a limited amount of time it can spend on one sound sample. All processing must be done within this small interval. If an algorithm requires more time than is available, the whole system ceases to be usable for real-time sound processing. On the other hand, certain degree of *quality* must be reached by the system. The better quality is required, the more complicated algorithms have to be used.

Unfortunately, the requirements of speed and quality are mutually exclusive. Increasing quality often decreases processing speed and vice versa. A decision must be made to find a balance between these two conflicting points by choosing an appropriate algorithm.

The following article might serve the reader as a comprehensible overview of well known interpolation methods and help him choose the best method to fit his needs.

Interpolation Algorithms

Many sound processing applications require changing sound stream sampling rate. A typical example of such an application would be a converter between two incompatible systems, like CD-audio (44 100 samples per second) and DAT (48 000 samples/s). The sampling rates could even change in time in complicated systems, like wave-table synthesizers. All of the above mentioned systems employ some kind of sample rate conversion algorithm. Sample interpolation is the most often used method even though it is not absolutely perfect.

Definitions

Let's suppose we have input sound stream sampled at a frequency f_1 and we want to obtain output sound stream sampled at a different frequency f_2. The ratio of these two values determines the sample position increment called *delta*:

$$delta = \frac{f_1}{f_2} \; . \tag{1}$$

[*] Ing. Vladimír Arnošt, Dept. of Computer Science and Engineering, Božetěchova 2, CZ-612 66, Brno
tel.: 0606-155444 fax.: 05/4114 1270 E-mail: arnost@fit.vutbr.cz

Given any sample position pos_i, the following position pos_{i+1} can be determined by equation:

$$pos_{i+1} = pos_i + delta \ . \tag{2}$$

Sample positions are rational numbers because they have to express also points lying between two discrete samples. In this article, a position will consist of two parts: *pos.int* will hold the integer part (3), and *pos.fract* will contain the fractional part or decimal digits (4). The possible values of *pos.fract* will be in the range $\langle 0;1)$. This notation is used to make algorithm descriptions simpler.

$$pos_i.int = \lfloor pos_i \rfloor \ , \tag{3}$$

$$pos_i.fract = pos_i - \lfloor pos_i \rfloor \ . \tag{4}$$

If the output sampling frequency is lower than input f_1 ($f_2 < f_1$ or $delta > 1$), some kind of anti-aliasing filter must be used. Otherwise spectral components above $f_2/2$ contained in the input signal would be aliased ('folded' down) around this frequency and would cause strong distortion of the resulting signal (ringing, noise, etc.). Such a negative effect is highly unwelcome and must be dealt with. This paper does not deal with the aliasing problem due to space limitation. More information about this can be found in [2].

These aliasing artifacts do not appear if the output frequency is higher than input frequency ($f_2 \geq f_1$). Instead, the new frequency band between f_1 and f_2 is filled with periodic copies of the original spectrum (remember, sampling creates infinite series of copies of the original spectrum spaced at integer multiples of f_S). These 'additional' frequency components could also sound like distortion, so it is desirable to remove them with a low-pass filter too.

The following paragraphs will discuss various interpolation methods from the simplest to quite sophisticated ones.

Sample-and-Hold

Sample-and-Hold (S/H for short) is the simplest conceivable method. It can be viewed as a memory cell that holds the last sampled value until the next one becomes available. The result is a staircase-like curve, where every step corresponds with one value. S/H is actually not an interpolation method, it is just a special case that can be called *zero-order interpolation*.

This method can be expressed by a math formula:

$$output = samples[pos.int] \ . \tag{5}$$

Pseudocode:

```
move   R2,Pos.int
move   R1,Samples[R2]
move   Output,R1
```

Explanation of symbols used in the pseudocode:

`Pos.int`	an integer part of current sample position
`Pos.fract`	a fractional part of current sample position
`Samples[]`	an array of source samples
`Output`	current output sample value
R*n*	work register, $n \in \langle 1; n_{max} \rangle$

All pseudo-instructions use the same format:

```
<operation> <destination>,<source>
```

Summary:

Pros: simple and very fast method

Cons: broadband noise, strong audible distortion, very low quality

Linear Interpolation

Linear interpolation is a major improvement over the previous method. Samples lying between any two points are determined by a line connecting these two points. The resulting curve is much smoother than in the previous case of S/H. This method can also be called the *first-order interpolation*. Detailed analysis can be found in [1].

The math representation of this method uses two variables s_0 and s_1 to hold adjacent sample values to make the calculation more readable. The value s_0 is actually equivalent to the S/H method output.

$$s_0 = samples[pos.int],$$
$$s_1 = samples[pos.int + 1],$$
$$output = s_0 + (s_1 - s_0) \cdot pos.fract\ . \quad (6)$$

Pseudocode:
```
move   R2,Pos.int
move   R1,Samples[R2]
move   R2,Samples[R2+1]
move   R3,Pos.fract
sub    R2,R1            ; R2 = R2 - R1
mult   R2,R3            ; R2 = R2 * R3
add    R1,R2
move   Output,R1
```

Summary:

Pros: simple method, good results

Cons: noise, audible distortion in higher frequencies, requires multiplication operation

Quadratic Interpolation

Quadratic interpolation places a parabolic curve between three adjacent control points. A problem arises when this curve is used to interpolate between two points. The third control point can be placed either before or after the two main points. It can be demonstrated that such asymmetrical interpolation schemes are not very suitable.

Lagrange polynomial can be used to derive equation (7). Parameter name *pos.fract* has been shortened to p_f to save space. The third point s_2 has been placed after the main points s_1 and s_2. The equation would look similar even if point s_2 was replaced by s_{-1}.

$$s_0 = samples[pos.int],$$
$$s_1 = samples[pos.int + 1],$$
$$s_2 = samples[pos.int + 2],$$

$$output = s_0 + \frac{p_f}{2}\left[(p_f - 3) \cdot s_0 - 2 \cdot (p_f - 2) \cdot s_1 + (p_f - 1) \cdot s_2\right], \text{ or} \quad (7)$$

$$output = s_0 + \frac{p_f}{2}\left[p_f (s_0 - 2s_1 + s_2) - 3s_0 + 4s_1 - s_2\right]. \quad (8)$$

Pseudocode:
```
    move   R3,Pos.int
    move   R4,Pos.fract
    move   R1,Samples[R3]
    move   R2,Samples[R3+1]
    move   R3,Samples[R3+2]
    move   R5,R4
    sub    R5,1
    mult   R3,R5             ; R3 = R3 * (R4 - 1)
    sub    R5,1
    mult   R2,R5             ; R2 = R2 * (R4 - 2)
    sub    R5,1
    add    R2,R2             ; R2 = R2 * 2
    shr    R4,1              ; R4'= R4 / 2
    sub    R3,R2
    move   R2,R1
    mult   R1,R5             ; R1 = R1 * (R4 - 3)
    add    R1,R3
    mult   R1,R4             ; R1 = R1 * (R4 / 2)
    add    R1,R2
    move   Output,R1
```
Summary:

Pros: better than plain linear interpolation

Cons: asymmetrically placed curve, requires 4 multiplication operations, sharp edges between adjacent curves

Cubic Interpolation

Cubic interpolation is a natural extension of quadratic interpolation. It uses four control points to create a third-order curve. Unlike the previous case, the control points are symmetrically distributed around the main interval (0,1).

As in the previous method, Lagrange polynomial was used to derive equation (9).

$$s_{-1} = samples[pos.int-1],\quad s_0 = samples[pos.int],$$
$$s_1 = samples[pos.int+1],\quad s_2 = samples[pos.int+2],$$
$$output = s_0 + \frac{p_f}{6}\begin{bmatrix}\left(-p_f^2+3p_f-2\right)\cdot s_{-1} + 3\cdot\left(p_f^2-2p_f-1\right)\cdot s_0 + \\ +3\cdot\left(-p_f^2+p_f+2\right)\cdot s_1 + \left(p_f^2-1\right)\cdot s_2\end{bmatrix}, \tag{9}$$

or alternatively

$$output = s_0 + \frac{p_f}{6}\begin{Bmatrix} p_f\left[p_f\left(-s_{-1}+3s_0-3s_1+s_2\right)+3s_{-1}-6s_0+3s_1\right]- \\ -2s_{-1}-3s_0+6s_1-s_2\end{Bmatrix}. \tag{10}$$

Pseudocode: (shortened)
```
    move   R4,Pos.int
    move   R5,Pos.fract
    move   R1,Samples[R4-1]
    move   R2,Samples[R4]
    move   R3,Samples[R4+1]
    move   R4,Samples[R4+2]
    move   R6,0.1666666667   ; 1/6
    mult   R6,R5             ; R6 = R5 * (1/6)
    move   R7,R5
    mult   R7,R7             ; R7 = R5 ^ 2
    ...                      ; R2 = [...]
    mult   R2,R6             ; R2 = R5/6 * [...]
    add    R1,R2
    move   Output,R1
```

Summary:

Pros: usable results

Cons: computationally intensive, requires many multiplication operations and work registers, sharp edges between adjacent curves

Cubic Spline Interpolation

This method is similar to ordinary cubic curve, but it is modified so that adjacent curves are linked smoothly without sharp edges. The first derivation (tangent) of the first curve end (s_1') must be equal to the tangent of the second curve beginning (s_0'). Derivations are replaced by differences for simplicity.

$$s_0' = s_0 - s_{-1}, \quad s_1' = s_2 - s_1$$

$$output = s_0 + \frac{p_f}{2}\left[\left(2p_f^2 - 3p_f - 1\right)\cdot\left(s_0 - s_1\right) + \left(p_f^2 - 2p_f + 1\right)\cdot s_0' + \left(p_f^2 - p_f\right)\cdot s_1'\right],$$

or
$$output = s_0 + \frac{p_f}{2}\begin{bmatrix}\left(-p_f^2 + 2p_f - 1\right)\cdot s_{-1} + \left(3p_f^2 - 5p_f + 2\right)\cdot s_0 + \\ + \left(-3p_f^2 + 4p_f + 1\right)\cdot s_1 + \left(p_f^2 - p_f\right)\cdot s_2\end{bmatrix}. \qquad (11)$$

Summary:

Pros: good results, smooth connection of adjacent curves

Cons: computationally intensive, requires many multiplication operations and work registers

Other Algorithms

The algorithms described in this paper are not the only possible. They present only a subset of known interpolation methods. Resampling algorithms are not described here in detail even though they are very important, but their analysis would exceed the limited space of this paper. Detailed information about advanced resampling algorithms can be found in references [2, 3, 4, 5].

Smith-Gossett Resampling Algorithm

One of the most commonly used resampling methods is the *Smith-Gossett algorithm* [4]. It simulates an "ideal" reconstruction low-pass filter whose output contains no frequency components above f_s ("virtual analog" signal) and which can be sampled at a new sampling frequency afterwards. The algorithm calculates only the required output samples so it is relatively efficient. Its built-in FIR filter has impulse response proportional to sinc(t) function (12) subsequently weighted by Kaiser window to limit it to reasonable length.

$$h(t) \mathrel{\widehat{=}} \mathrm{sinc}(f_s t) \mathrel{\widehat{=}} \frac{\sin(\pi f_s t)}{\pi f_s t}. \qquad (12)$$

Filter impulse response is pre-calculated and stored in a table for speed. Since sinc(t) is a continuous function and only its discrete samples are available in tabular form, linear interpolation is usually used to get intervening values. Such a simplification adds an error which can be minimized by properly choosing impulse response sampling interval according to needed precision.

FIR filter output is calculated by a modified convolution operation in which input samples $x(pos.int - i)$ are multiplied by $h(pos.fract + i)$. The use of *pos.fract* value inside the calculation creates a so-called "fractional delay filter".

Summary:

Pros: very good results, low distortion, can be used to band-limit original signal too

Cons: very computationally intensive, requires many multiplication/add operations and a large pre-computed table of impulse response function $h(t)$

Comparison of Interpolation Algorithms

Quality Comparison

Interpolation quality can be measured by mean square error. The lower the error, the better. The error value is the difference between ideally resampled signal and the interpolation output value.

$$err_i = output_i - ideal_i \ . \qquad (13)$$

$$\sigma^2 = \tfrac{1}{n}\sum_{i=0}^{n-1} err_i^2 \ . \qquad (14)$$

The mean square error measurements are shown in a table:

Method	Sine 1	Sine 2	Square	Saw
Sample-and-Hold	0.110552	1.054130	0.380800	0.334155
Linear Curve	0.017905	0.859352	0.137723	0.140336
Quadratic Curve	0.016429	0.957986	0.150208	0.152964
Cubic Curve	0.016125	0.957727	0.141162	0.143916
Cubic Spline	0.016146	0.966322	0.142320	0.145050

All test signals have amplitude ±1. Sine 1 is defined as $\sin(2\pi k/100)$, Sine 2 is $\sin(2\pi k/7.3)$, square wave and saw tooth have a period of 64 samples. 5000 samples of real signal were downsampled to 381 samples and then interpolated back to 5000 samples. The results were compared with real signal and mean square error was calculated.

The table indicates that more complicated methods do not always give better results and that mean square error is not the best quality measure. All methods except for S/H seem to be rather good. Cubic spline method is able to produce smooth curve without sharp edges at the cost of slower reaction to sudden changes. Linear curve fits best the square and sawtooth signals and is surprisingly good also in sine wave interpolation. For many applications, linear interpolation seems to suffice, because more complicated methods are too complex and do not provide such a visible improvement as linear interpolation over S/H. Smith-Gossett algorithm quality wasn't analyzed.

Speed Comparison

All methods have been benchmarked and the results are shown in a table:

Method	Time	Time*	Relative Speed
Empty Loop	628	0	N/A
Sample-and-Hold	2734	2106	1.00
Linear Curve	4340	3712	1.76
Quadratic Curve	5318	4690	2.23
Cubic Curve	6128	5500	2.61
Cubic Spline	6042	5414	2.57

Time is an average execution time of 5 measurements given in ms. Time* is the execution time without empty loop overhead. The benchmark routines were written in the C language and used double precision floating point numbers for all computations.

It is interesting to see that cubic spline is actually slightly faster than cubic curve. This phenomenon can be accounted to compiler optimizations because both methods have the same complexity and the only significant differences are in the polynomial coefficients.

Conclusions

This paper has analyzed some of the most popular interpolation methods. It has been shown that the speed is inversely proportional to method complexity and that quality depends on many factors including the type of interpolated signal, frequency ratios, etc. Cubic curve and cubic spline differ only in a few coefficients, but their output is rather different.

The reader may choose his own criteria to pick the most suitable method. It is not possible to say which method is the best, because it depends mostly on its intended use.

References

[1] PROAKIS, J. G., MANOLAKIS, D. G.: *Digital Signal Processing: Principles, Algorithms and Applications*. 3rd edition, Prentice-Hall International, New Jersey, 1996.

[2] ROTHACHER, F. M.: *Sample-Rate Conversion: Algorithms and VLSI Implementation*. [Ph.D. thesis], Swiss Federal Institute of Technology, Zurich, 1995.

[3] SMITH, J. O.: *Bandlimited Interpolation: Introduction and Algorithm*. Stanford University, Stanford, California, 1993.

[4] SMITH, J. O., GOSSETT, P.: *A Flexible Sampling-Rate Conversion Method.* Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, San Diego, March 1984 (ICASSP-84), Volume II, pp.19.4.1–19.4.2, New York, 1984.

[5] SMITH, J. O.: *Digital Audio Resampling Home Page*, Stanford University, Stanford, California, 2001. `http://www-ccrma.stanford.edu/~jos/resample/`

[6] ŽÁRA, J., BENEŠ, B., FELKEL, P.: *Moderní počítačová grafika*. Computer Press, Praha, 1998, ISBN 80-7226-049-9.