**Hermite Spline Approximation**

Hermite Spline Approximation splits function into intervals and for each interval builds a 3rd degree polynomial that conforms to

Or for interval :

Coefficients for the polynomial on can be found with formulas

(Here ):

|  |
| --- |
|  |
|  |
|  |
|  |

Or when substituting and simplifying:

|  |
| --- |
|  |
|  |
|  |
|  |

**Argument x cutoff**

The function drops fast and returns 32 bit values. After certain function values become so small that they will be always rounded to 0.

We find such that :

If x is sampled in such a way that there is 16 bit of samples for every sigma we round up

For 16 bit samples x cutoff is . This is last x that will return non-zero value.

For sampling rate is 32 bit of samples per sigma:

For 32 bit samples x cutoff is . This is the last x that will return non-zero value.

For 60 bit samples x cutoff is 7798021677424194372. This is the last x that will return non-zero value.

**Approximation Error - Theoretical**

On some interval error bound can be found with the formula taken from the literature:

where

* – point where error is evaluated
* – value in interval

is maximized when is in the middle between and . In other words when .

is monotonically decreasing when . Therefor its max value on interval is always at , or:

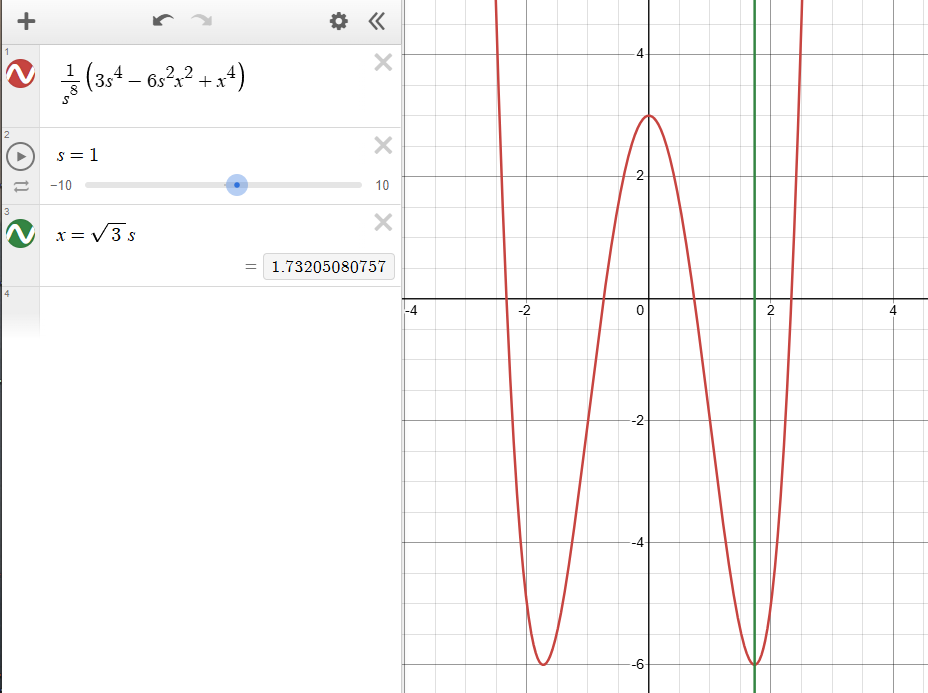
Therefore, max error on the interval can be found:

Note that when the point of min value of is .

And if interval is left of () then function is monotonically falling, and max is at .

If interval is right of () then function is monotonically raising, and max is at .

If interval includes , then max must be chosen between and .



**Approximation Error – practical**

After series of test runs we found out that if function is split into such number of splines that there is 64 splines for each unit (433 splines total) there is sufficient approximation precision such that if function value is represented with 16-bit integer there will be no error whatsoever and if the function value is represented with 32-bit integer value max error will be 2 and error will be 0 on average.

**Coefficients Representation**

For 64-section/unit split max absolute value of a polynomial coefficient is 2.01 (mean is 0.369 with stdev 0.551). Coefficients can be negative and positive.

All coefficients are stored in 64 bit signed integers with 60-bit fractional precision. This leaves 3 bit for the integer part and is consistent with x that is also represented with 60 bit precision.

**Coefficient A**:

* Max absolute value is for spline #47 (when counting from 0).
* Max value is around 0.23.

**Coefficient B**:

* Max absolute value is for spline #47 (when counting from 0).
* Max value is around 0.683.

**Coefficient C**:

* Max absolute value is for spline #149 (when counting from 0).
* Max absolute value is around abs(-1.86).

**Coefficient D**:

* Max absolute value is for spline #149 (when counting from 0).
* Max absolute value is around 2.01.

**Calculating function value**

Function approximation is 3rd degree polynomial . It is put into form and is calculated in 3 multiplications and 3 additions.

Internally x is represented as 64-bit integer with 60-bit fixed precision. Polynomial coefficients are also represented this way. And intermediate values are also presented in this manner. Let’s look at this representation:

* 1 bit – sign.
* 3 bits – integer part. Max value of x is 6.76, max value between coefficients is 2.01 and max value of any of 6 calculation steps is -1.95 (step 5). Therefore, 3 bits are sufficient to cover any integer number we can encounter.
* 60 bit – fractional part. Fractional part is represented as 60-bit integer number each unit of this part is .

Multiplication step:

Since the numbers involved are expected to use almost all of 64 bits used for their representation there is a strong possibility of overflow when performing multiplication. To avoid overflow numbers are split into top and low parts each and parts are multiplied individually.

Both parts are stored in 64-bit variables. Both parts are no bigger than 32 bits of significant digits. When we multiply two 32-bit wide numbers there is no possibility to get more than 64 bits wide result. Note, that 64 bits of the result include sign leaving effectively 63 bits for multiplication result, but because top part also contains sign it is at most 31 bits wide, so no overflow occurs.

* Bottom part – To get bottom part mask out 32 top bits of original number with mask **4294967295**. This will always produce positive number. Fractional precision of this part will be same as original – 60 bits of precision.
* Top part – To get to part mask out bottom 32 bits of original number with mask **18446744069414584320** and then shift right 32 places. Fractional precision of this part will be bits.

With 2-complement representation of negative integers just adding up these two numbers will produce the original number (of course left shift of top part is required first).

This way when we have two numbers represented with integers and to multiply we split each number into lower and upper part and we will get.

To get the final result each of these multiplication terms must be shifted to 60-bit precision and then added up. Here bit width of the multiplication result is guaranteed to be no more than 63 bits, because we multiply 32-bit width number with 31-bit width number.

* – This will always produce 4 zero bits at the bottom.
* – Up to bits are expected.
* – Up to bits are expected.
* – Only up to bits are expected.

It is guaranteed that no overflow will occur regardless of signs and order of addition.

Addition step

Simply add 60-bit precision number to another 60-bit precision number. Since it is guaranteed that on any step absolute value of the result is no more than -1.95 this addition should not overflow 3 bits that were dedicated to the integer part.

Steps evaluation statistics

Max 64 bit signed [+9223372036854775807]

Min 64 bit signed [-9223372036854775808]

**Step 1**

Max Abs: 0.20373397674448994431155439101550434276155993607537

At x: 1

In spline: 63

**Step 2**

Max Abs: -0.53532127454710790894013917743028489408717857594743

At x: 17/32

In spline: 34

**Step 3**

Max Abs: 1.0388105979612388105055823537677071219993864729834

At x: 157/64

In spline: 156

**Step 4**

Max Abs: -0.86876470443268397172883264882865594698691774009914

At x: 135/64

In spline: 135

**Step 5**

Max Abs: -1.9514218209790160666012186883768360435159579700813

At x: 153/64

In spline: 152

**Step 6**

Max Abs: 1.0000000000000000000000000000000000000000000000000

At x: 0

In spline: 0

**Areas - algorithm**

Algorithm for calculating function integral (needed for averaging) will work in following way. We split each unit of x axis into 64 sections for function value approximation. For each section precise pre-calculated area is stored. When you need to find area on some interval you sum up needed number of segment areas. On edges of the interval there might be sub-scale parts that are smaller than width of pre-calculated segment. For these two parts we calculate areas directly by integrating 5th degree polynomial approximation. Coefficients for this integral are also pre-calculated. To avoid summing up large number of areas (since this is not only time consuming, but also introduces errors) we use pre-calculated sums. We pre-calculate a ladder of smaller and smaller segments with steps of multiples of 2. We will store pre-calculated areas of 64, 32, 16, 8, 4, 2, 1 segment per unit splits. And in addition to that there are areas that cover 2, 4 and 8 units. When summing algorithm starts with largest segments that are in the interval and when it goes under smallest scale it calculates the rest directly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Scale index** | **Segments per unit** | **Fractional precision** | **Number of segments** | **Width in 60 bits precision** |
| Scale 9 | 1/8u | -3 bit | 1 | 9223372036854775808 |
| Scale 8 | 1/4u | -2 bit | 2 | 4611686018427387904 |
| Scale 7 | 1/2u | -1 bit | 4 | 2305843009213693952 |
| Scale 6 | 1/u | 0 bit | 7 | 1152921504606846976 |
| Scale 5 | 2/u | 1 bit | 13 | 576460752303423488 |
| Scale 4 | 4/u | 2 bit | 26 | 288230376151711744 |
| Scale 3 | 8/u | 3 bit | 52 | 144115188075855872 |
| Scale 2 | 16/u | 4 bit | 104 | 72057594037927936 |
| Scale 1 | 32/u | 5 bit | 208 | 36028797018963968 |
| Scale 0 | 64/u | 6 bit | 415 | 18014398509481984 |

Algorithm for calculating the area works in 3 stages. First two stages iterate through scales from larger width to smaller with current scale passed from the first stage to the second.

First stage searches for a segment of the largest scale that can fit into the interval. There might be one, or two such segments, but no more than two. Scales are iterated from large to small until first suitable scale is found.

Second stage continues on the scale that comes after the largest one. For each subsequent scale it checks if additional segment of this scale can fit on left or right side of segments that were found in previous steps. Interval that was already accounted for grows from segment found at the first stage on left and right sides.

Third stage starts after smallest scale 0 is checked on the second stage. In this stage we integrate left and right sub-scale intervals.

**Areas – representation**

Since areas are going to be summed which might produce errors and are also used for finding average values (areas are going to be divided) it is reasonable to keep them as precise as possible.

Since total area for this function (positive half) is around 1.25 we need only 1 bit to represent integer part of any sub-area.

Therefore, areas are represented with 64 bit uint where top bit is integer part and lower 63 bits are fractional part.

**Areas – cutoff**

We will find x cutoff such that area from to infinity will be less than half a step of 32 bit integer number:

Solution for this inequality is found with wolfram alpha:

Then with fractional precision of 60 bits cutoff value for x is **7468738554291142405**.

**Indefinite integral for the area**

We use integral of the approximation polynomial to find approximation of the area within boundaries of a spline.

Approximation polynomial of 3rd degree used for finding function values is not suitable for this task because the lack of precision. To find areas approximation we use more precise 5th degree polynomial:

Coefficients for this polynomial on interval are found from following equations system, where first 4 equations are the same as in value approximation and additional two add constraints for total area and the first moment:

Coefficients are produced with sympy automatic solver.

Indefinite integral is:

And we find the area by calculating , and finding difference:

is put in the following form and is calculated in 11 steps – 6 multiplications and 5 additions:

To avoid performing divisions that do not affect precision much, coefficients are pre-divided and following numbers are actually stored:

, , , , ,

Value of this polynomial is calculated in the same manner as the approximation value polynomial. 60-bit precision for calculation steps can be again used since max values of calculation steps have at most 1 bit for the integer part.

Steps evaluation statistics

Step 1

Max Abs: 0.050933494186122486077888597753876085690389984018844

At x: 1

In spline: 63

Step 2

Max Abs: -0.18879797608280883393518796441367548933061371914423

At x: 39/64

In spline: 39

Step 3

Max Abs: 0.37690215641015059308219668189700764021371790337618

At x: 79/32

In spline: 157

Step 4

Max Abs: -0.56707332823248236090927312356010214521623332308656

At x: 141/64

In spline: 141

Step 5

Max Abs: -1.3213969438012629460572603266304864501563087720082

At x: 157/64

In spline: 156

Step 6

Max Abs: 1.0000000000000000000000000000000000000000000000000

At x: 0

In spline: 0

Step 7

Max Abs: 1.6539995482012362129683822036131497663739585804257

At x: 19/8

In spline: 151

**Normal Distribution**

Normal Distribution function takes values from GaussCurve function and then normalizes them by dividing by .

Normalization constant:

With 64-bit fractional precision:

Sigma values smaller than 1 will increase the value. Since we cannot return values larger or equal to 1.0 in a 32-bit variable we allow sigma values only down to certain limit. Minimal sigma: