Faster Math Functions

Robin Green
R&D Programmer
Sony Computer Entertainment America

What Is This Talk About?

- **This is an Advanced Lecture**
  - There will be equations
  - Programming experience is assumed
- **Writing your own Math functions**
  - Optimize for Speed
  - Optimize for Accuracy
  - Optimize for Space
  - Understand the trade-offs

Running Order

- **Part One – 10:00 to 11:00**
  - Floating Point Recap
  - Measuring Error
  - Incremental Methods
    - Sine and Cosine
- **Part Two – 11:15 to 12:30**
  - Table Based Methods
  - Range Reduction
  - Polynomial Approximation

Running Order

- **Part Three – 2:00 to 4:00**
  - Fast Polynomial Evaluation
  - Higher Order functions
    - Tangent
    - Arctangent, Arcsine and Arccosine
- **Part Four – 4:15 to 6:00**
  - More Functions
    - Exponent and Logarithm
    - Raising to a Power
  - Q&A

Floating Point Formats

32-bit Single Precision Float

Floating Point Standards

- **IEEE 754 is undergoing revision.**
  - In process right now.
- **Get to know the issues.**
  - Quiet and Signaling NaNs.
  - Specifying Transcendental Functions.
  - Fused Multiply-Add instructions.

History of IEEE 754

History of IEEE 754

- IEEE754 ratified in 1985 after 8 years of meetings.
- A story of pride, ignorance, political intrigue, industrial secrets and genius.
- A battle of Good Enough vs. The Best.

Timeline: The Dark Ages

- **Tower of Babel**
  - On one machine, values acted as non-zero for add/subtract and zero for multiply-divide.

```
b = b * 1.0;
if(b==0.0) error;
else return a/b;
```

  - On another platform, some values would overflow if multiplied by 1.0, but could grow by addition.
  - On another platform, multiplying by 1.0 would remove the lowest 4 bits of your value.
  - Programmers got used to storing numbers like this

```
b = (a + a) - a;
```

Timeline: 8087 needs "The Best"

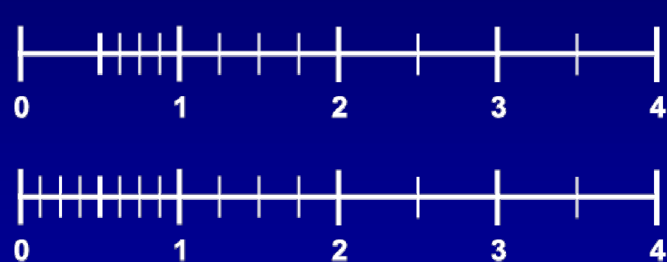
- **Intel decided the 8087 has to appeal to the new mass market.**
  - Help "normal" programmers avoid the counterintuitive traps.
  - Full math library in hardware, using only 40,000 gates.
  - Kahan, Coonen and Stone prepare draft spec, the K-C-S document.

Timeline: IEEE Meetings

- **Nat Semi, IBM, DEC, Zilog, Motorola, Intel all present specifications.**
  - Cray and CDC do not attend…
- **DEC with VAX has largest installed base.**
  - Double float had 8-bit exponent.
  - Added an 11-bit “G” format to match K-C-S, but with a different exponent bias.
- **K-C-S has mixed response.**
  - Looks complicated and expensive to build.
  - But there is a rationale behind every detail.

Timeline: The Big Argument

- **K-C-S specified Gradual Underflow.**
- **DEC didn’t.**



Timeline: The Big Argument

- Both Cray and VAX had no way of detecting flush-to-zero.
- Experienced programmers could add extra code to handle these exceptions.
- How to measure the Cost/Benefit ratio?

Timeline: Trench Warfare

- DEC vs. Intel
  - DEC argued that Gradual Underflow was impossible to implement on VAX and too expensive.
  - Intel had cheap solutions that they couldn't share (similar to a pipelined cache miss).
- Advocates fought for every inch
  - George Taylor from U.C.Berkeley built a drop-in VAX replacement FPU.
  - The argument for "impossible to build" was broken.

Timeline: Trench Warfare

- **DEC turned to theoretical arguments**
  - If DEC could show that GU was unnecessary then K-C-S would be forced to be identical to VAX.
- **K-C-S had hard working advocates**
  - Prof Donald Knuth, programming guru.
  - Dr. J.H. Wilkinson, error-analysis & FORTRAN.
- **Then DEC decided to force the impasse…**

Timeline: Showdown

- **DEC found themselves a hired gun**
  - U.Maryland Prof G.W.Stewart III, a highly respected numerical analyst and independent researcher
- **In 1981 in Boston, he delivered his verdict verbally…**

> *"On balance, I think Gradual Underflow is the right thing to do."*

Timeline: Aftermath

- By 1984, IEEE 754 had been implemented in hardware by:
  - Intel
  - AMD
  - Apple
  - IBM
  - Nat. Semi.
  - Weitek
  - Zilog
  - AT&T
- It was the *de facto* standard long before being a published standard.

Why IEEE 754 is best

The Format

- **Sign, Exponent, Mantissa**
  - Mantissa used to be called "Significand"
- **Why base2?**
  - Base2 has the smallest "wobble".
  - Base2 also has the hidden bit.
    - More accuracy than any other base for N bits.
    - Base3 arguments always argue using fixed-point values
- **Why 32, 64 and 80-bit formats?**
  - Because 8087 could only do 64-bits of carry propagation in a cycle!

Why A Biased Exponent?

- **For sorting.**
- **Biased towards underflow.**

```
exp_max =  127;
exp_min = -126;
```

- Small number reciprocals will never Overflow.
- Large numbers will use Gradual Underflow.

The Format

- **Note the Symmetry**

1	11111111	???????????????????????	Not A Number
1	11111111	00000000000000000000000	Negative Infinity
1	11111110	???????????????????????	Negative Numbers
1	00000000	??????????????????????1	Negative Denormal
1	00000000	00000000000000000000000	Negative Zero
0	00000000	00000000000000000000000	Positive Zero
0	00000000	??????????????????????1	Positive Denormal
0	00000001	???????????????????????	Positive Numbers
0	11111111	00000000000000000000000	Positive Infinity
0	11111111	???????????????????????	Not A Number

Rounding

- **IEEE says operations must be "exactly rounded towards even".**
- **Why towards even?**
  - To stop iterations slewing towards infinity.
  - Cheap to do using hidden "guard digits".
- **Why support different rounding modes?**
  - Used in special algorithms, e.g. decimal to binary conversion.

Rounding

- **How to round irrational numbers?**
  - Impossible to round infinite numbers accurately.
  - Called the *Table Makers Dilemma*.
    - In order to calculate the correct rounding, you need to calculate worst case values to infinite precision.
    - E.g. Sin(x) = 0.02310000000000000007
- **IEEE754 just doesn't specify functions**
  - Recent work looking into worst case values

Special Values

- **Zero**
  - 0.0 = 0x00000000
- **NaN**
  - Not an number.
  - NaN = sqrt(-x), 0*infinity, 0/0, etc.
  - Propagates into later expressions.

Special Values

- **±Infinity**
  - Allows calculation to continue without overflow.
- **Why does 0/0=NaN when ±x/0=±infinity?**
  - Because of limit values.
  - a/b can approach many values, e.g.

$$\left.\begin{array}{l} \dfrac{\sin(x)}{x} \rightarrow 1 \\ \dfrac{1-\cos(x)}{x} \rightarrow 0 \end{array}\right\} \text{ as } x \rightarrow 0$$

Signed Zero

- **Basically, WTF?**
  - Guaranteed that +0 = -0, so no worries.
- **Used to recover the sign of an overflowed value**
  - Allows 1/(1/x) = x as x→+inf
  - Allows log(0)=-inf and log(-x)=NaN
  - In complex math, sqrt(1/-1) = 1/sqrt(-1) only works if you have signed zero

Destructive Cancellation

- **The nastiest problem in floating point.**
- **Caused by subtracting two very similar values**
  - For example, in quadratic equation if $b^2 \approx 4ac$
  - In fixed point…

```
  1.10010011010010010011101
- 1.10010011010010010011100
  -------------------------
  0.00000000000000000000001
```

  - Which gets renormalised with no signal that almost all digits have been lost.

Compiler "Optimizations"

- **Floating Point does not obey the laws of algebra.**
  - Replace `x/2` with `0.5*x` – good
  - Replace `x/10` with `0.1*x` – bad
  - Replace `x*y-x*z` with `x*(y-z)` – bad if y≈z
  - Replace `(x+y)+z` with `x+(y+z)` – bad
- **A good compiler will not alter or reorder floating point expressions.**
  - Compilers should flag bad constants, e.g.

```
float x = 1.0e-40;
```

Decimal to Binary Conversion

- **In order to reconstruct the correct binary value from a decimal constant**

Single float : **9** digits

Double float : **17** digits

- Loose proof in the Proceedings
  - works by analyzing the number of representable values in sub-ranges of the number line, showing a need for between 6 and 9 decimal digits for single precision

Approximation Error

Measuring Error

- **Absolute Error**
  - Measures the size of deviation, but tell us nothing about the significance
  - The abs() is often ignored for graphing

$$error_{abs} = \left| f_{actual} - f_{approx} \right|$$

Measuring Error

- **Absolute Error sometimes written ULPs**
  - Units in the Last Place

Approx	Actual	ULPs
0.0312	0.0314	2
0.0314	0.0314159	0.159

Measuring Error

- **Relative Error**
  - A measure of how important the error is.

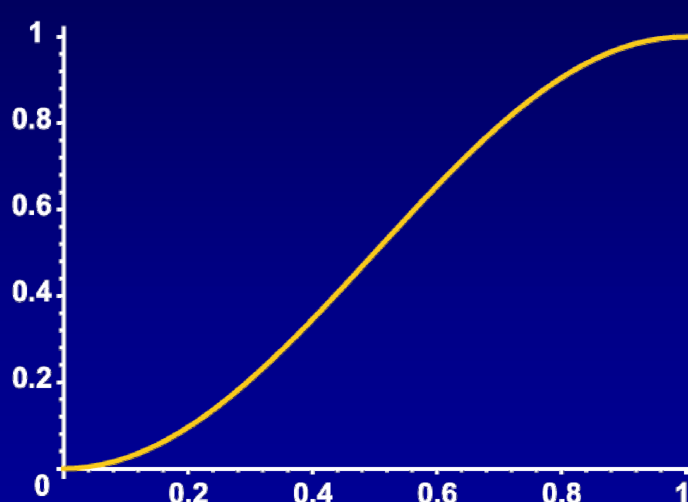
$$error_{rel} = 1 - \frac{f_{approx}}{f_{actual}}$$

Example: Smoothstep Function

- **Used for ease-in ease-out animations and anti-aliasing hard edges**
  - Flat tangents at x=0 and x=1

$$f(x) = \frac{1}{2} - \frac{\cos(\pi x)}{2}$$

Smoothstep Function

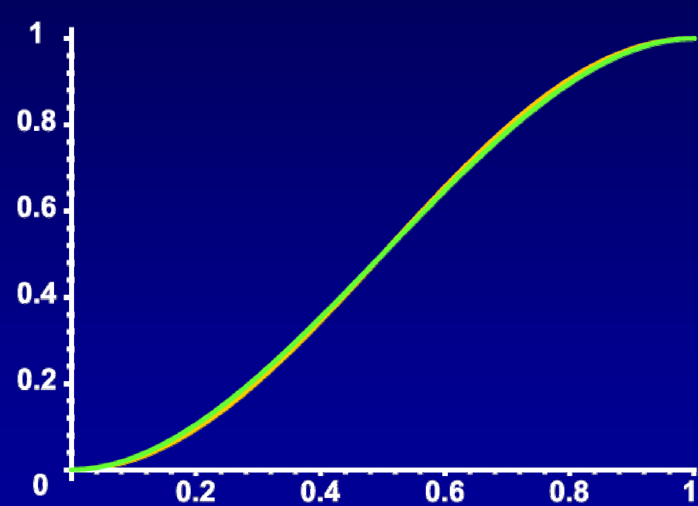


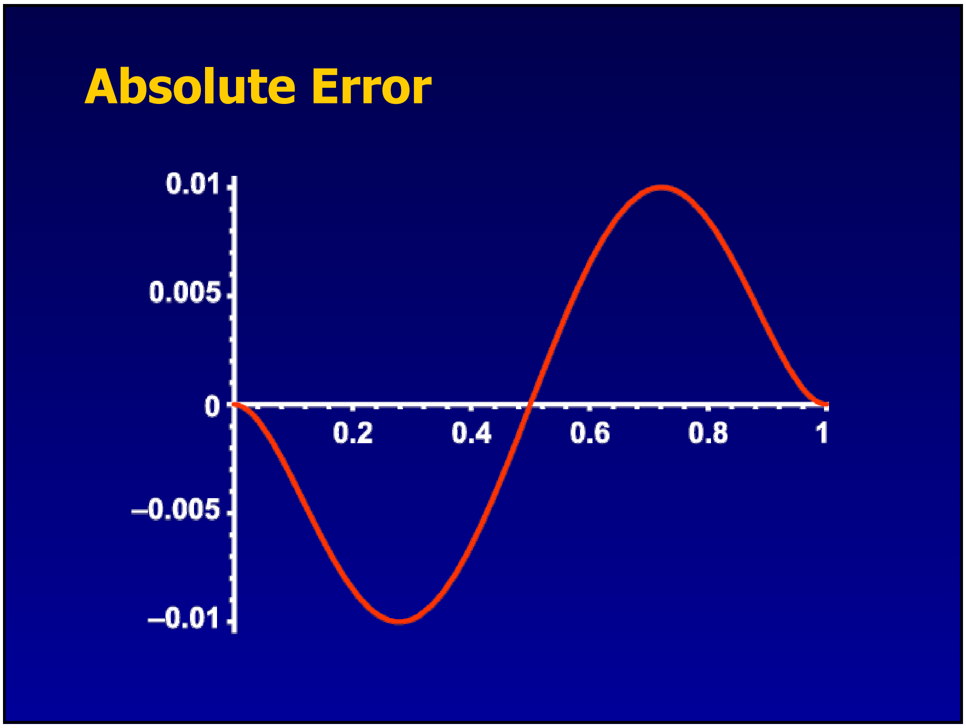
Smoothstep Approximation

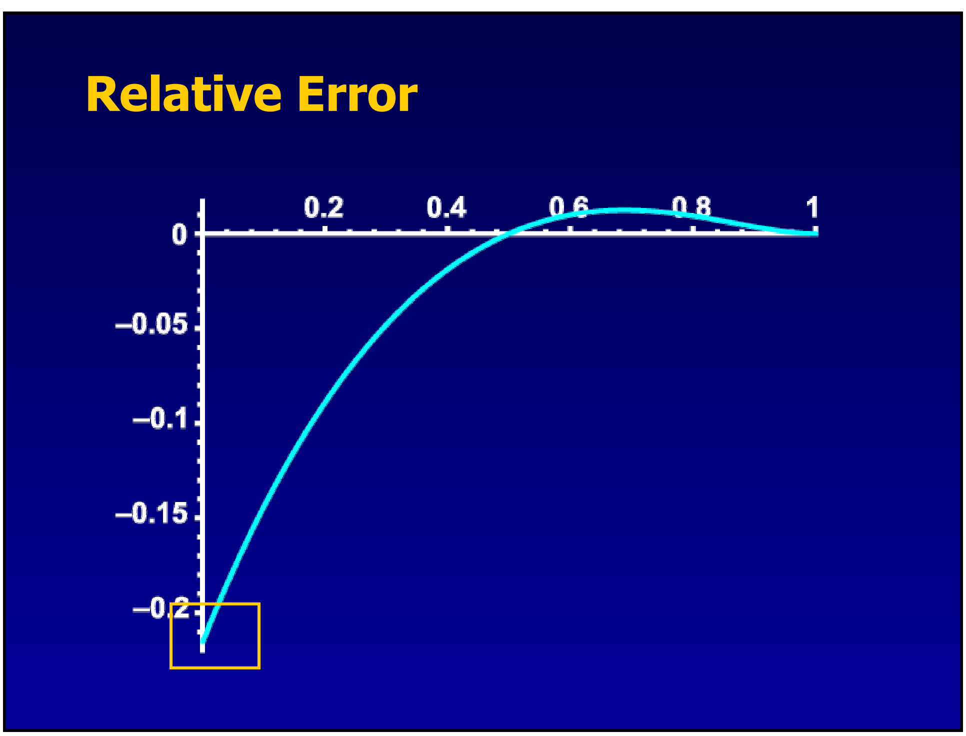
- **A cheap polynomial approximation**
  - From the family of Hermite blending functions.

$$f_{approx}(x) = 3x^2 - 2x^3$$

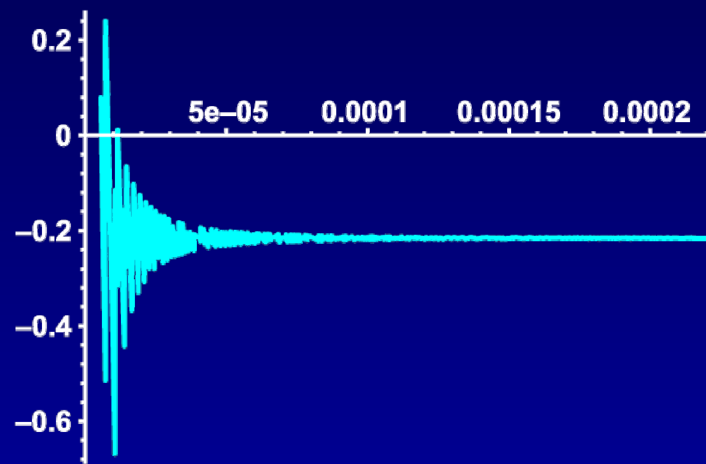
Smoothstep Approximation




Absolute Error
0.01
0.005
0
−0.005
−0.01
0.2
0.4
0.6
0.8
1


Relative Error
0.2
0.4
0.6
0.8
1
0
−0.05
−0.1
−0.15
−0.2

Relative Error Detail



Incremental Algorithms

Incremental Methods

Q: What is the fastest method to calculate sine and cosine of an angle?

A: Just two instructions.

There are however two provisos.

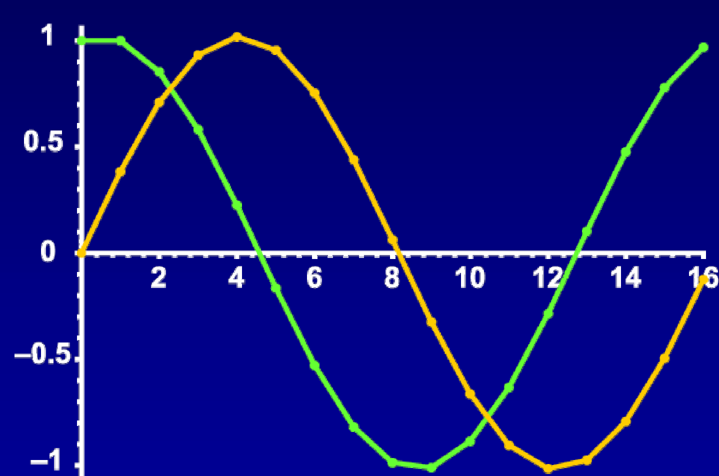
1. You have a previous answer to the problem.
2. You are taking equally spaced steps.

Resonant Filter

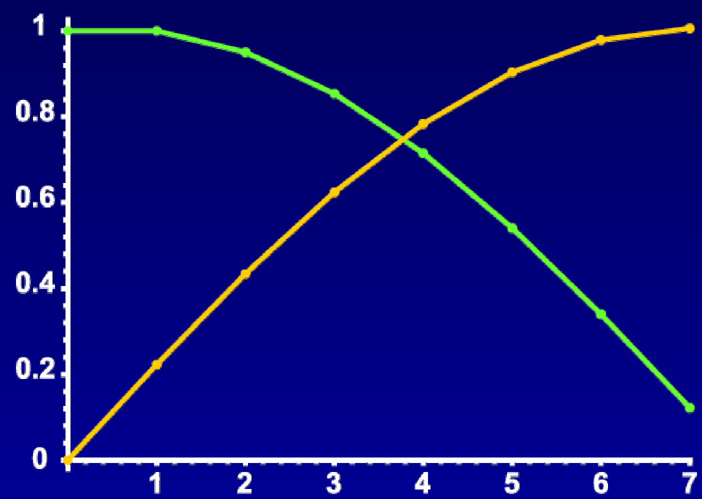
- Example using 64 steps per cycle.
- NOTE: new `s` uses the previously updated `c`.

```
int N = 64;
float a = sin(2PI/N);
float c = 1.0f;
float s = 0.0f;
for(int i=0; i<M; ++i) {
   output_sin = s;
   output_cos = c;
   c = c - s*a;
   s = s + c*a;
   ...
}
```

Resonant Filter Graph



Resonant Filter Quarter Circle

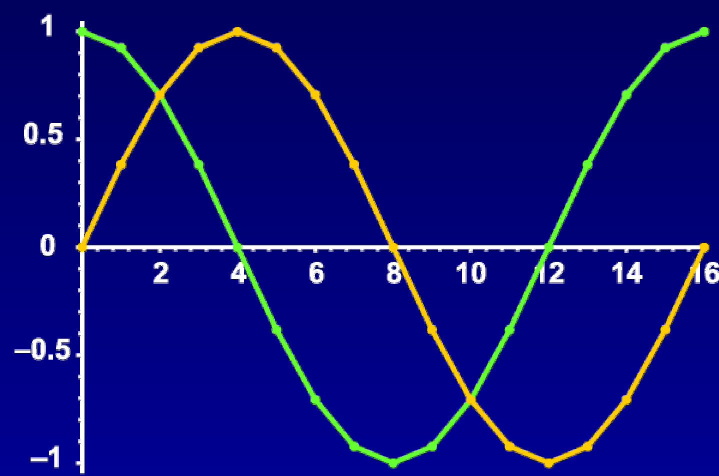


Goertzels Algorithm

- **A more accurate algorithm**
  - Uses two previous samples (Second Order)
- **Calculates** `x = sin(a+n*b)` **for all integer** `n`

```
float cb = 2*cos(b);
float s2 = sin(a+b);
float s1 = sin(a+2*b);
float c2 = cos(a+b);
float c1 = cos(a+2*b);
float s,c;
for(int i=0; i<m; ++i) {
   s = cb*s1-s2;
   c = cb*c1-c2;
   s2 = s1; c2 = c1;
   s1 = s; c1 = c;
   output_sin = s;
   output_cos = c;
   ...
}
```

Goertzels Algorithm Graph



Goertzels Initialization

- **Needs careful initialization**
  - You must account for a three iteration lag

```
// N steps over 2PI radians
float b = 2PI/N;

// subtract three steps from initial value
float new_a = a - 3.0f * b;
```

Goertzels Algorithm Quarter Circle

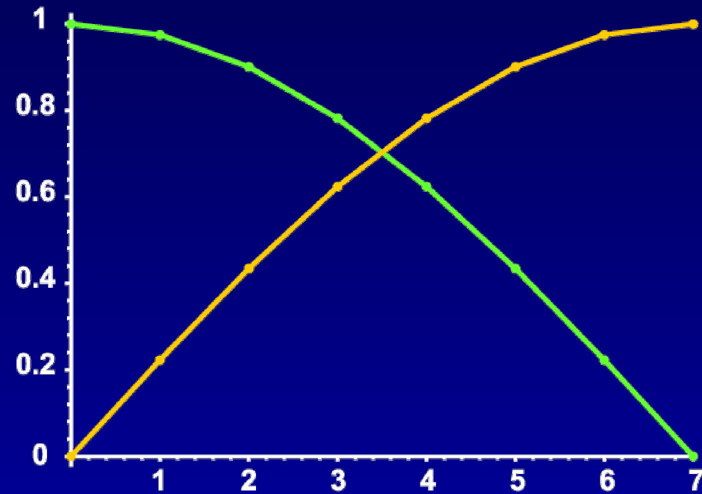


Table Based Solutions

Table Based Algorithms

- **Traditionally the sine/cosine table was the fastest possible algorithm**
  - With slow memory accesses, it no longer is
- **New architectures resurrect the technique**
  - Vector processors with closely coupled memory
  - Large caches with small tables forced in-cache
- **Calculate point samples of the function**
  - Hash off the input value to find the nearest samples
  - Interpolate these closest samples to get the result

Table Based Sine

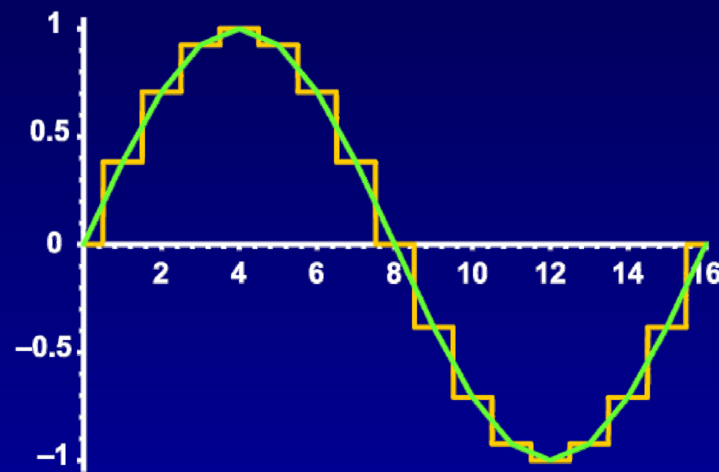
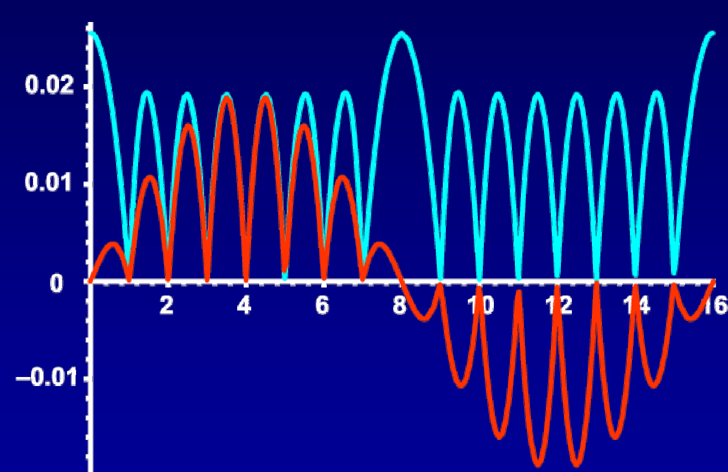


Table Based Sine Error



Precalculating Gradients

- Given an index `i`, the approximation is…

$$\sin(x) \approx \text{table}[i] + \Delta * (\text{table}[i+1] - \text{table}[i])$$
$$= \text{table}[i] + \Delta * \text{gradient}[i]$$

- Which fits nicely into a 4-vector…

`sine`	`cosine`	`sin-grad`	`cos-grad`

How Accurate Is My Table?

- **The largest error occurs when two samples straddle the highest curvature.**
  - Given a stepsize of Δx, the error E is:

$$E = 1 - \cos\left(\frac{\Delta x}{2}\right)$$

  - e.g. for 16 samples, the error will be:

$$1 - \cos(\pi/16) = 0.0192147$$

How Big Should My Table Be?

- **Turning the problem around, how big should a table be for an accuracy of E?**
  - We just invert the expression…

$$
\begin{aligned}
E &= 1\% \\
1-\cos(\pi/N) &< 1\% \\
\cos(\pi/N) &> 1-0.01 \\
N &> \pi/\arccos(0.99) \\
N &> 22.19587\ldots \\
N &\approx 23
\end{aligned}
$$

How Big Should My Table Be?

- **We can replace the arccos() with a small angle approximation, giving us a looser bound.**

$$N = \frac{\pi}{\sqrt{2E}}$$

- **Applying this to different accuracies gives us a feel for where tables are best used.**

Table Sizes

	E	360°	45°
1% accurate	0.01	23	3
0.1% accurate	0.001	71	9
0.01% accurate	0.0001	223	28
1 degree	0.01745	17	3
0.1 degree	0.001745	54	7
8-bit int	2^{-7}	26	4
16-bit int	2^{-15}	403	51
24-bit float	10^{-5}	703	88
32-bit float	10^{-7}	7025	880
64-bit float	10^{-17}	~infinite	8.7e+8

Range Reduction

Range Reduction

- **We need to map an infinite range of input values `x` onto a finite working range `[0..C]`.**

- **For most transcendentals we use a technique called "Additive Range Reduction"**
  - Works like `y = x mod C` but without a divide.
  - We just work out how many copies of `C` to subtract from `x` to get it within the target range.

Additive Range Reduction

1. We remap 0..C into the 0..1 range by scaling

```
const float C = range;
const float invC = 1.0f/C;
x = x*invC;
```

2. We then truncate towards zero (e.g. convert to int)

```
int k = (int)(x*invC);
       // or (x*invC+0.5f);
```

3. We then subtract `k` copies of `C` from `x`.

```
float y = x - (float)k*C;
```

High Accuracy Range Reduction

- **Notice that `y = x-k*C` has a destructive subtraction.**
- **Avoid this by encoding C in several constants.**
  - First constant `C1` is a rational that has M bits of `C`'s mantissa, e.g. `PI = 201/64 = 3.140625`
  - Second constant `C2 = C - C1`
  - Overall effect is to encode `C` using more bits than machine accuracy.

```
float n = (float)k;
float y = (x - n*C1) - n*C2;
```

Truncation Towards Zero

- **Another method for truncation**
  - Add the infamous 1.5 * 2^{24} constant to your float
  - Subtract it again
  - You will have lost the fractional bits of the mantissa

```
A = 123.45    = 1111011.01110011001100110
B = 1.5*2^24  = 110000000000000000000000.
A = A+B       = 110000000000000001111011.
A = A-B       = 1111011.00000000000000000
```

  - This technique requires you know the range of your input parameter…

Quadrant Tests

- **Instead of range reducing to a whole cycle, let's use `C=Pi/2` - a quarter cycle**
  - The lower bits of `k` now holds which quadrant our angle is in
- **Why is this useful?**
  - Because we can use double angle formulas
  - A is our range reduced angle.
  - B is our quadrant offset angle.

$$\sin(A+B) = \sin(A)\cos(B) + \cos(A)\sin(B)$$

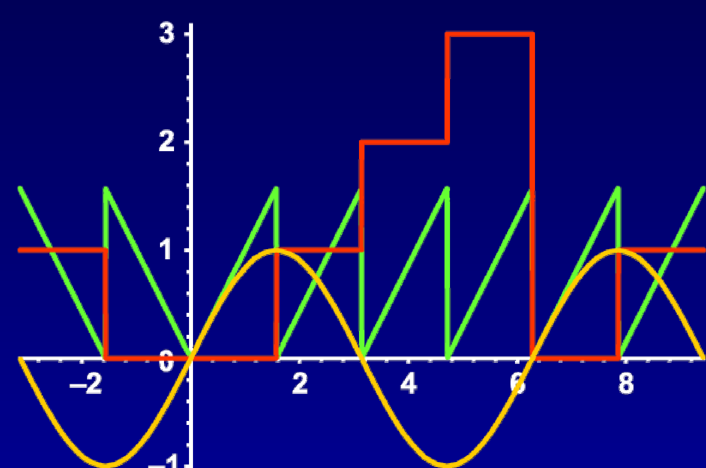
$$\cos(A+B) = \cos(A)\cos(B) + \sin(A)\sin(B)$$

Double Angle Formulas

- With four quadrants, the double angle formulas now collapses into this useful form

$$\sin(y + 0 * \pi/2) = \sin(y)$$
$$\sin(y + 1 * \pi/2) = \cos(y)$$
$$\sin(y + 2 * \pi/2) = -\cos(y)$$
$$\sin(y + 3 * \pi/2) = -\sin(y)$$

Four Segment Sine



A Sine Function

- **Leading to code like this:**

```
float table_sin(float x)
{
  const float C = PI/2.0f;
  const float invC = 2.0f/PI;
  int k = (int)(x*invC);
  float y = x-(float)k*C;
  switch(k&3) {
    case 0: return sintable(y);
    case 1: return sintable(TABLE_SIZE-y);
    case 2: return -sintable(TABLE_SIZE-y);
    default: return -sintable(y);
  }
  return 0;
}
```

More Quadrants

- **Why stop at just four quadrants?**
  - If we have more quadrants we need to calculate both the sine and the cosine of y.
  - This is called the *reconstruction* phase.

$$\sin\left(y+\frac{3\pi}{16}\right)=\sin(y)*\cos\left(\frac{3\pi}{16}\right)+\cos(y)*\sin\left(\frac{3\pi}{16}\right)$$

  - Precalculate and store these constants.
  - For little extra effort, why not return both the sine AND cosine of the angle at the same time?
  - This function traditionally called `sincos()` in FORTRAN libraries

Sixteen Segment Sine

```
float table_sin(float x)
{
  const float C = PI/2.0f;
  const float invC = 2.0f/PI;
  int k = (int)(x*invC);
  float y = x-(float)k*C;
  float s = sintable(y);
  float c = costable(y);
  switch(k&15) {
    case 0: return s;
    case 1: return s*0.923879533f + c*0.382683432f;
    case 2: return s*0.707106781f + c*0.707106781f;
    case 3: return s*0.382683432f + c*0.923879533f;
    case 4: return c;
    ...
  }
  return 0;
}
```

Math Function Forms

- **Most math functions follow three phases of execution**

1. Range Reduction
2. Approximation
3. Reconstruction

- **This is a pattern you will see over and over**
  - Especially when we meet Polynomial Approximations

Polynomial Approximation

Infinite Series

- **Most people learn about approximating functions from Calculus and Taylor series**

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

- **If we had infinite time and infinite storage, this would be the end of the lecture.**

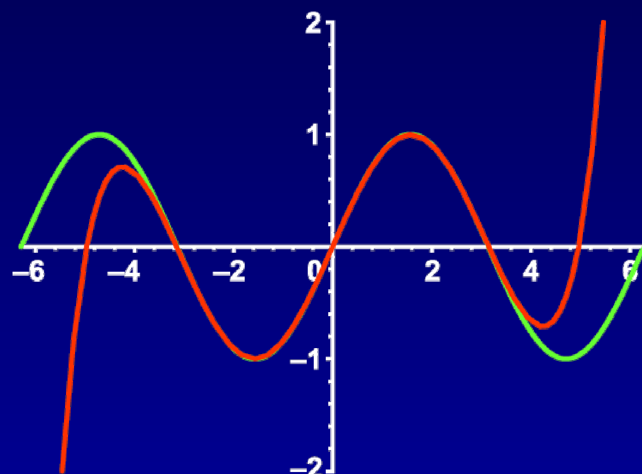
Taylor Series

- **Taylor series are generated by repeated differentiation**
  - More strictly, the Taylor Series around x=0 is called the Maclauren series

$$f(x) = f(0) + f'(0) + \frac{f''(0)}{2!} + \frac{f'''(0)}{3!} + \dots$$

- **Usually illustrated by graphs of successive approximations fitting to a sine curve.**

Taylor Approx of Sine



Properties Of Taylor Series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$$

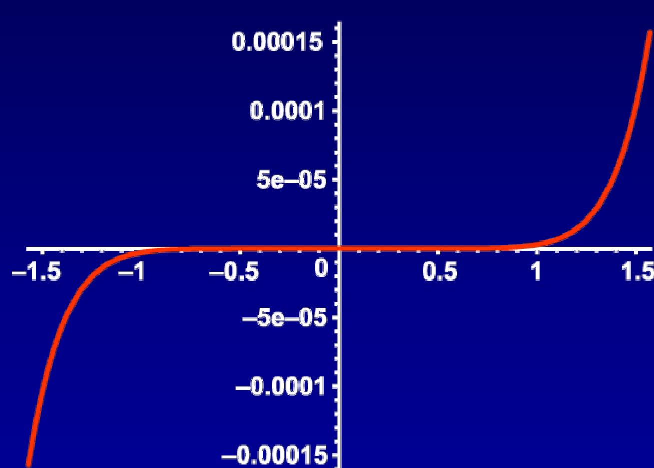
- **This series shows all the signs of convergence**
  - Alternating signs
  - Rapidly increasing divisor
- **If we truncate at the 7th order, we get:**

$$\sin(x) \approx x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7$$
$$= x - 0.16667x + 0.0083333x^5 - 0.00019841x^7$$

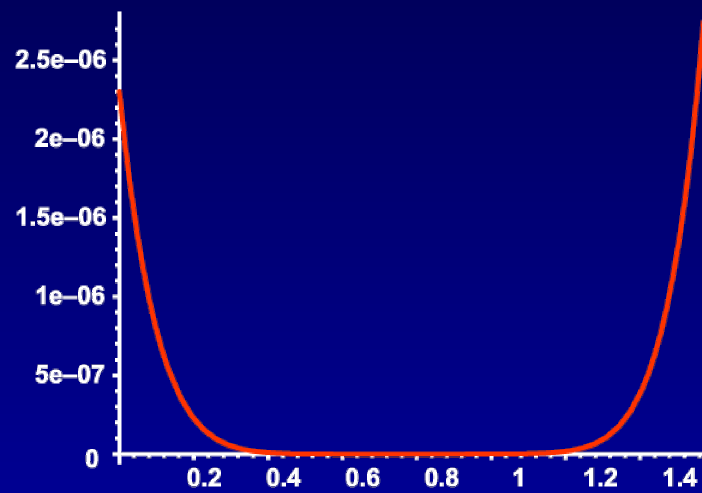
Graph of Taylor Series Error

- **The Taylor Series, however, has problems**
  - The problem lies in the error
  - Very accurate for small values but is exponentially bad for larger values.
- **So we just reduce the range, right?**
  - This improves the maximal error.
  - Bigger reconstruction cost, large errors at boundaries.
  - The distribution of error remain the same.
- **How about generating series about x=Pi/4**
  - Improves the maximal error.
  - Now you have twice as many coefficients.

Taylor 7th Order for –Pi/2..Pi/2



Taylor 7th Order for 0..Pi/2



Taylor 7th Order for 0..Pi/2

- And now the bad news.

$$
\begin{aligned}
\sin(x) \approx\ & -0.0000023014110 + \\
& 1.000023121x + \\
& -0.00010117322x^2 + \\
& -0.1664154429x^3 + \\
& -0.00038530806x^4 + \\
& 0.008703147018x^5 + \\
& -0.0002107589082x^6 + \\
& -0.0001402989645x^7
\end{aligned}
$$

Taylor Series Conclusion

- **For our purposes a Taylor series is next to useless**
  - Wherever you squash error it pops back up somewhere else.
  - Sine is a well behaved function, the general case is much worse.
- **We need a better technique.**
  - Make the worst case nearly as good as the best case.

Orthogonal Polynomials

Orthogonal Polynomials

- **Families of polynomials with interesting properties.**
  - Named after the mathematicians who discovered them
  - Chebyshev, Laguerre, Jacobi, Legendre, etc.

- **Integrating the product of two O.P.s returns zero if the two functions are different.**

$$\int w(x) P_i(x) P_j(x)\, dx = \begin{cases} c_j & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

  - Where w(x) is a weighting function.

Orthogonal Polynomials

- **Why should we care?**
  - If we replace $P_i(x)$ an arbitrary function f(x), we end up with a scalar value that states how similar f(x) is to $P_j(x)$.
  - This process is called projection and is often notated as

$$\langle f \,|\, P_j \rangle = \langle f | w | P_j \rangle = \int f(x) P_j(x) w(x)\, dx$$

- **Orthogonal polynomials can be used to approximate functions**
  - Much like a Fourier Transform, they can break functions into approximating components.

Chebyshev Polynomials

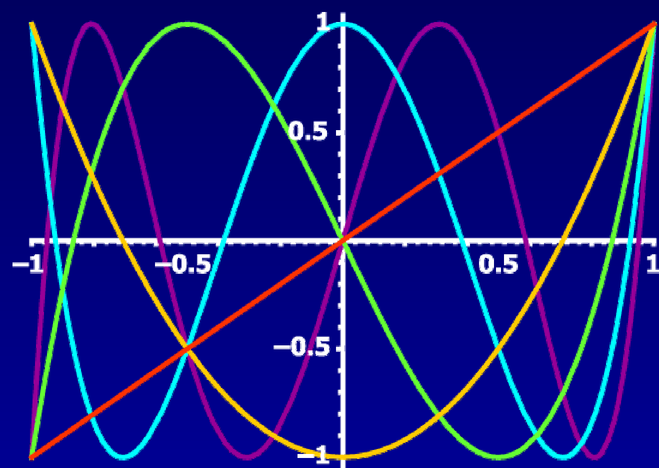
- **Lets take a concrete example**
  - The Chebyshev Polynomials $T_n(x)$

$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_2(x) = 2x^2 - 1$$
$$T_3(x) = 4x^3 - 3x$$
$$T_4(x) = 8x^4 - 8x^2 - 1$$
$$T_4(x) = 16x^5 - 20x^3 + 5x$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

Chebyshev Plots

- **The first five Chebyshev polynomials**



Chebyshev Approximation

- **A worked example.**
  - Let's approximate f(x) = sin(x) over [-π..π] using Chebyshev Polynomials.
  - First, transform the domain into [-1..1]

$$a = -\pi$$

$$b = \pi$$

$$g(x) = f\left(\frac{a-b}{2}x + \frac{a+b}{2}\right)$$

$$= \sin(\pi x)$$

Chebyshev Approximation

- **Calculate coefficient k_n for each $T_n(x)$**

$$k_n = \frac{\int_{-1}^{1} g(x) T_n(x) w(x) dx}{c_n}$$

Where the constant c_n and weighting function w(x) are

$$c_n = \begin{cases} \pi & \text{if } n = 0 \\ \pi/2 & \text{otherwise} \end{cases} \qquad w(x) = \frac{1}{\sqrt{1-x^2}}$$

Chebyshev Coefficients

- **The resulting coefficients**

$$k_0 = 0.0$$
$$k_1 = 0.5692306864$$
$$k_2 = 0.0$$
$$k_2 = -0.666916672$$
$$k_4 = 0.0$$
$$k_5 = 0.104282369$$
$$k_6 = \dots$$

- This is an infinite series, but we truncate it to produce an approximation to g(x)

Chebyshev Reconstruction

- **Reconstruct the polynomial in x**
  - Multiply through using the coefficients k_n

$$
\begin{aligned}
g(x) \approx k_0(1) &+ \\
k_1(x) &+ \\
k_2(2x^2 - 1) &+ \\
k_3(4x^3 - 3x) &+ \\
k_3(4x^3 - 3x) &+ \\
k_4(8x^4 - 8x^2 - 1) &+ \\
k_5(16x^5 - 20x^3 + 5x) &
\end{aligned}
$$

Chebyshev Result

- **Finally rescale the domain back to [-π..π]**

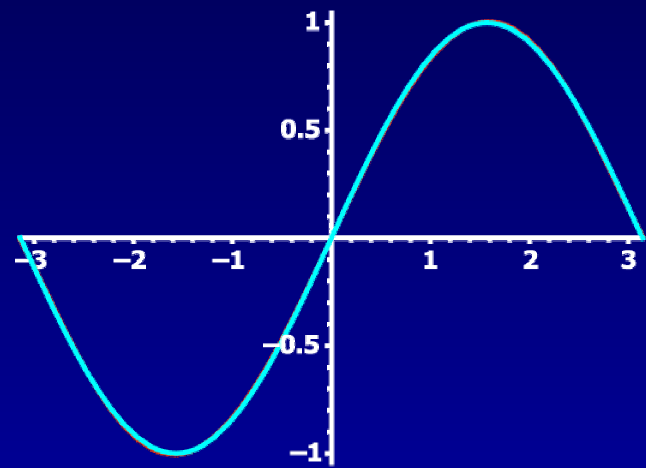
$$f(x) \leftarrow g\left(\frac{2}{b-a}x - \frac{a+b}{b-a}\right)$$

  - Giving us the polynomial approximation

$$
\begin{aligned}
f(x) \approx\ & 0.984020813\, x + \\
& -0.153301672\, x^3 + \\
& 0.00545232216\, x^5
\end{aligned}
$$

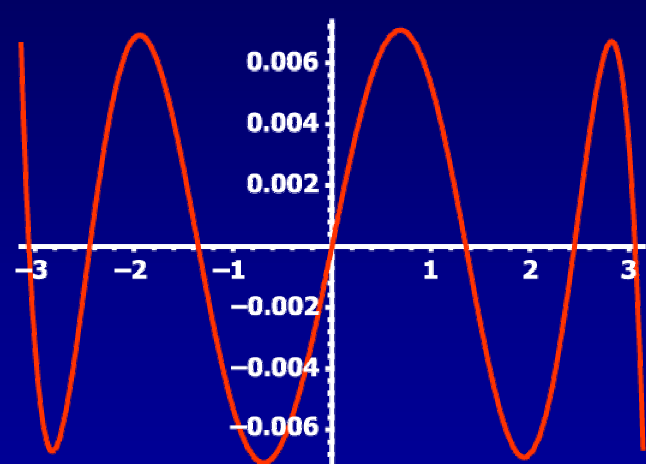
Chebyshev Result

- The approximated function f(x)



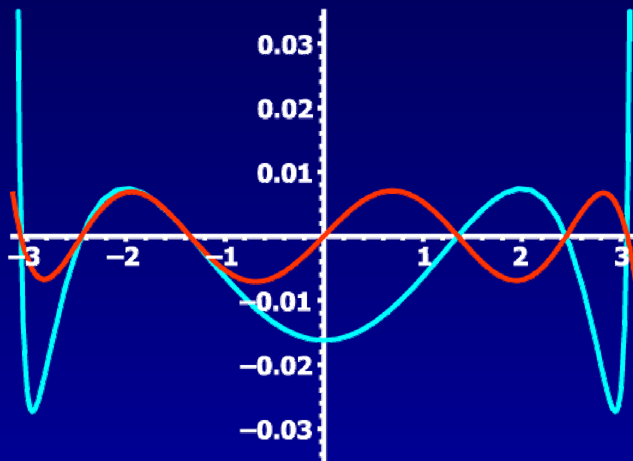
Chebyshev Absolute Error

- The absolute error sin(x)-f(x)



Chebyshev Relative Error

- The relative error tells a different story…



Chebyshev Approximation

- **Good points**
  - Approximates an explicit, fixed range
  - Uses easy to generate polynomials
  - Integration is numerically straightforward
  - Orthogonal Polynomials used as basis for new techniques
    - E.g. Spherical Harmonic Lighting
- **Bad points**
  - Imprecise control of error
  - No clear way of deciding where to truncate series
  - Poor relative error performance

[Continued in part 2]