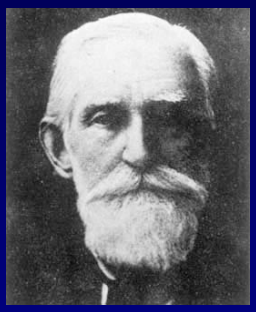
Faster Math Functions

(part 2)

Robin Green
R&D Programmer
Sony Computer Entertainment America

Minimax Polynomials

Pafnuty Chebyshev 1821-1894



- **Chebyshev described what the error for the best approximating polynomial should look like.**
- **An Order-N approximation error curve should:**
  a. Change sign N+1 times
  b. Touch the maximal error N+2 times
- **This function minimizes the maximal error, so it is called the *Minimax Polynomial***

Evgeny Remez 1896-1975

- **The algorithm to generate these polynomials is the *Remez Exchange Algorithm***
- **The algorithm requires:**
  - A function to be approximated
  - A range of approximation
  - The required order of approximation
  - A weighting function

Remez Exchange Algorithm

- **The algorithm works by generating a set of zero crossings for the error curve, e.g.**

$$\sin(x) - a + bx_0 + cx_0^2 = 0$$
$$\sin(x) - a + bx_1 + cx_1^2 = 0$$
$$\cdots$$
$$\sin(x) - a + bx_n + cx_n^2 = 0$$

- **These are solved for a,b and c as a matrix inversion. The maximal error is found and one x_n is altered. Repeat until minimized.**

Remez Exchange Algorithm

- **The algorithm is very sensitive to error and requires large number representations (e.g. 50 digits or more) and numerical solvers.**
  - The best place to find these tools is in a mathematical package like *Maple* or *Mathematica*.
  - In Maple, a raw call to `minimax()` will use all coefficients available, so we will have to set up the calculation…

Forming The Minimax Inequality

- **Start by looking at the Taylor Expansion of sine to get a feel for what the end result**
  - Every odd power, first coefficient is 1.0

$$\sin(x) \approx x - 0.16667x^3 + 0.0083333x^5 - 0.00019841x^7$$

- **Transform the problem into finding $P(x^2)$ in**

$$\sin(x) \approx x + x^3 P(x^2)$$

The Minimax Inequality

- **Next, we form the minimax inequality expressing our desire to minimize the relative error**

$$\left| \frac{\sin(x) - x - x^3 P(x^2)}{\sin(x)} \right| \leq error$$

The Minimax Inequality

- **Divide through by** x^3

$$\left| \frac{\frac{\sin(x)}{x^3} - \frac{1}{x^2} - P(x^2)}{\frac{\sin(x)}{x^3}} \right| \leq error$$

The Minimax Inequality

- **We want only every other power, so substitute** $y = x^2$

$$\left| \frac{\frac{\sin\left(\sqrt{y}\right)}{y^{3/2}} - \frac{1}{y} - P(y)}{\frac{\sin\left(\sqrt{y}\right)}{y^{3/2}}} \right| \leq error$$

The Function to Approximate

- **We solve for $P(y)$ giving us**

$$P(y) = \frac{\sin\left(\sqrt{y}\right)}{y^{3/2}} - \frac{1}{y}$$

- **With the weighting function**

$$w(y) = \frac{y^{3/2}}{\sin\left(\sqrt{y}\right)}$$

Final Details

- **Before we can run this through the minimax algorithm, we need to convert to a Taylor Series**
  - This stops the sin() function being evaluated and greatly speeds the calculation

$$P(y) = -\frac{1}{6} + \frac{y}{120} - \frac{y^2}{5040} + \frac{y^3}{362880} - \ldots$$

- **We also transform our approximation range**

$$x^2 \to y \Rightarrow \left[0..\frac{\pi}{2}\right] \to \left[0..\frac{\pi^2}{2^2}\right]$$

Calculating The Minimax Poly

- We run these values through the minimax algorithm:

```
minimax(p(y), y=0..Pi^2/4, [2,0], w(y));
```

- It returns (to 5 d.p.):

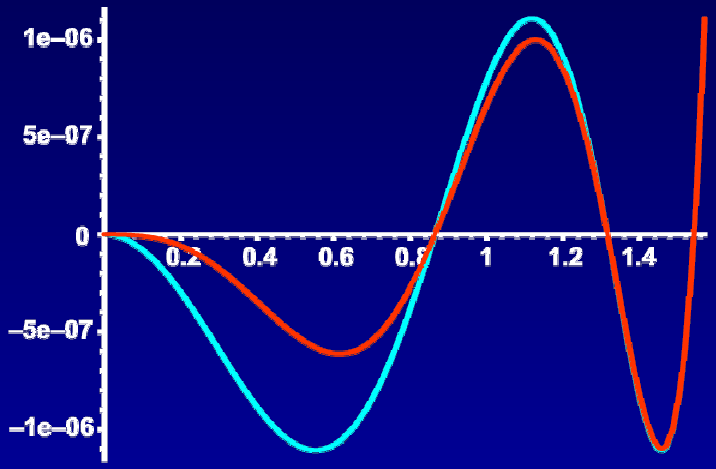
$$P(y) = -0.16666 + 0.0083143\,y - 0.00018542\,y^2$$

- Finally, we resubstitute back into the original equation

$$\sin(x) \approx x - 0.16666x^3 + 0.0083143x^5 - 0.00018542x^7$$

Minimax Polynomial Error

- Maximal error = 1.108e-6 at x=1.5707



Magic Numbers

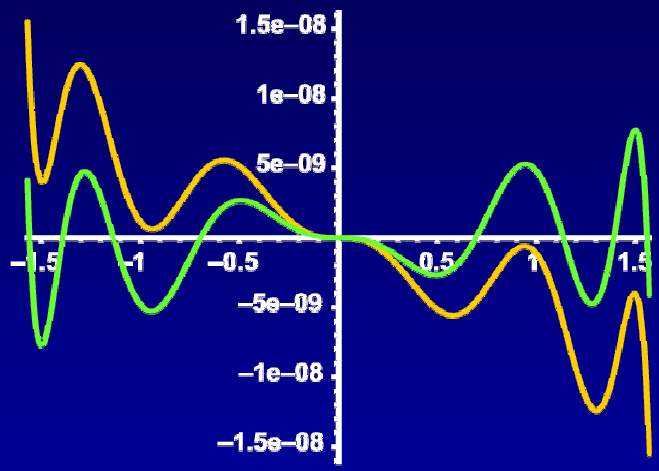
- **Comparing the results**

$$\text{taylor} = x - 0.16667x^3 + 0.0083333x^5 - 0.00019841x^7$$
$$\text{minimax} = x - 0.16666x^3 + 0.0083143x^5 - 0.00018542x^7$$

- **An enormous increase in accuracy with a tiny change in coefficients.**
  - We must be very sure of our coefficients.
  - We must pay attention to floating point rounding.

Side Note: PS2 ESIN Instruction

- **The manual entry for ESIN has a small error in it.**



Improving On Perfection

- **We can optimize polynomials for floating point representation**
  - This will guarantee better accuracy for small angles, as the first term has the most effect in these cases.
  - Requires a smaller range than Pi/2, so we will use 0..Pi/4
- **Using a different form of minimax inequality, we can specify the first coefficient:**

$$\sin(x) \approx x + kx^3 + x^5 P(x^2)$$

Optimizing For Floating Point

- **Now we have to specify the first coefficient as a single-precision float.**

$$k = -0.166666666\dot{6} = -1/6$$

- **Multiply k by 2^{24}, round to zero and divide by 2^{24} to create an exactly representable number.**

$$\frac{\lfloor k \times 2^{24} \rfloor}{2^{24}} = \frac{-2796203}{16777216}$$
$$= -0.166666686534881591796875000000$$

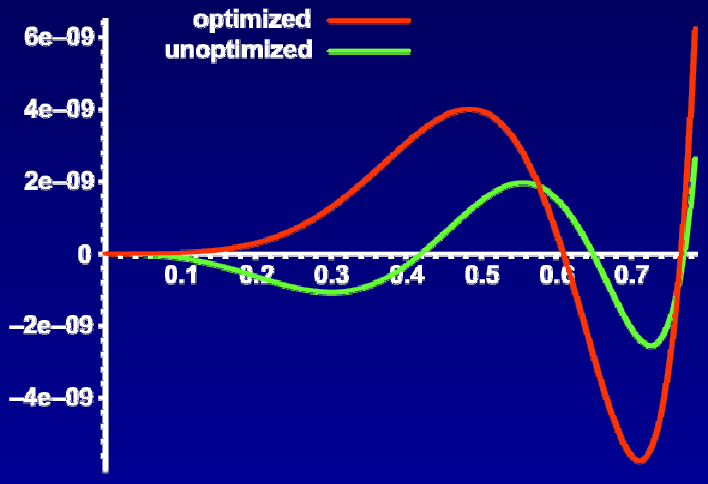
Optimizing For Floating Point

- **Solving the Minimax Inequality for P(x) and reinserting the answer gives us**

$$\sin(x) \approx x - \frac{2796203}{16777216}x^3 + 0.00833282412x^5 - 0.000195878418x^7$$

- **Maximal error = 6.275e-9 at x=0.7855**
  - About twice the error of the unoptimized version but with better accuracy for small angles.

Float Optimized Sine Minimax



Optimizing for Small Angles

- **Most functions are bound by the accuracy of their first coefficient**
  - In the same way we split the range reduction into two coefficients, we can do the same for the first coefficient of a polynomial, e.g.

$$\ln(x) \approx a_0 x + a_1 x + x^2 P(x)$$

  - Where $a_0+a_1 = k$, the leading coefficient.

Fast Polynomial Evaluation

Polynomial Evaluation

- **Now we have our coefficients, we need to evaluate them efficiently.**
  - When presented with a polynomial of the form

$$y = cx^2 + bx + a$$

  - Most of us would produce code like

```
y = c*x*x + b*x + a;
```

Large Polynomials

- Faced with a larger polynomial…

$$y = Cx^7 + Bx^5 + Ax^3 + x$$

… you can see the need for better methods

```
y = C*x*x*x*x*x*x*x +
    B*x*x*x*x*x +
    A*x*x*x +
    x;
```

Horner Form

- Most good programmers will use Horner Form to encode the polynomial

```
z = x*x;
y = (((C*z + B)*z + A)*z + 1)*x;
```

- This also codes well if you have multiply-add instructions available
- Horner Form is used by nearly all libraries, with coefficients stored as arrays of values

The Problem With Horner Form

- **Horner Form leads to very serial computations**
  - Later instructions are waiting for earlier results.
  - A lot of instructions are stalled in the pipeline.
  - Makes no use of available SIMD instructions.
- **There is a better way – Estrin's Method.**
  - Comes from parallel computation research
  - Only documented in Knuth – nowhere online!

Estrin's Method

- **Stems from the observation that by bracketing, we can produce regular sub-expressions**

$$\begin{aligned} f(x) &= Dx^3 + Cx^2 + Bx + A \\ &= (Dx + C)x^2 + (Bx + A) \\ &= Px^2 + Q \end{aligned}$$

- **The trick is to keep a running copy of x^{2^N} as "glue" between each sub expression…**

Estrin's Method Table

$p_0(x) = A$

$p_1(x) = (Bx+A)$

$p_2(x) = (C)x^2+(Bx+A)$

$p_3(x) = (Dx+C)x^2+(Bx+A)$

$p_4(x) = (E)x^4+((Dx+C)x^2+(Bx+A))$

$p_5(x) = (Fx+E)x^4+((Dx+C)x^2+(Bx+A))$

$p_6(x) = ((G)x^2+(Fx+E))x^4+((Dx+C)x^2+(Bx+A))$

$p_7(x) = ((Hx+G)x^2+(Fx+E))x^4+((Dx+C)x^2+(Bx+A))$

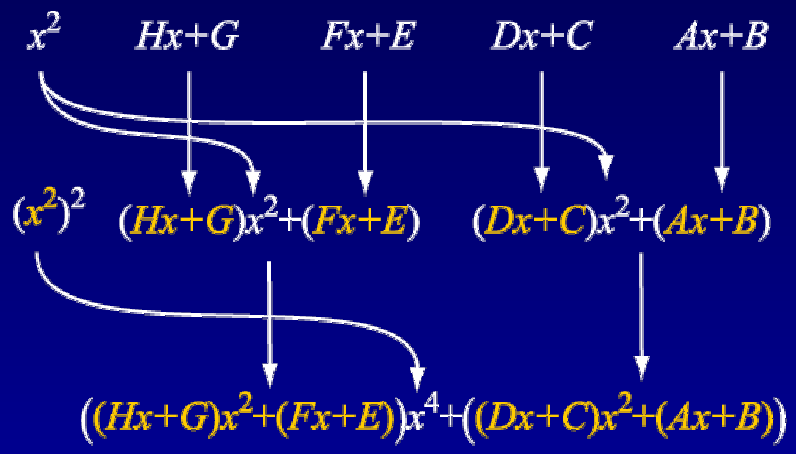
$p_8(x) = (I)x^8+(((Hx+G)x^2+(Fx+E))x^4+((Dx+C)x^2+(Bx+A)))$

$p_9(x) = (Jx+I)x^8+(((Hx+G)x^2+(Fx+E))x^4+((Dx+C)x^2+(Bx+A)))$

$p_{10}(x) = ((K)x^2+(Jx+I))x^8+(((Hx+G)x^2+(Fx+E))x^4+((Dx+C)x^2+(Bx+A)))$

$p_{11}(x) = ((Lx+K)x^2+(Jx+I))x^8+(((Hx+G)x^2+(Fx+E))x^4+((Dx+C)x^2+(Bx+A)))$

Estrin's Method Chart



Packing Estrin's Into SIMD

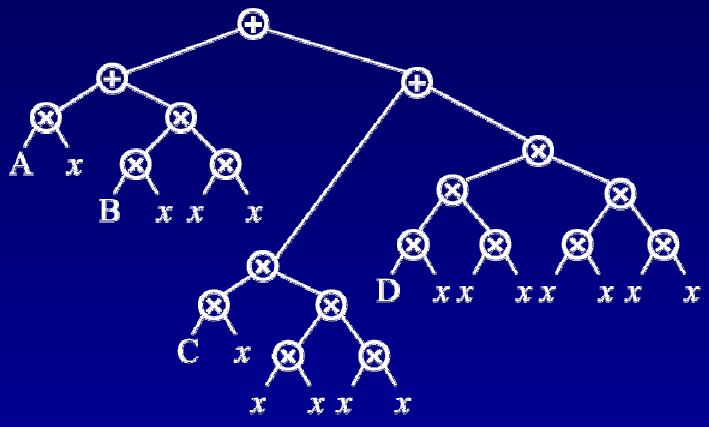
- **Evaluating the polynomial takes O(log N) time**
  - Proves that you can improve on Horner Form
- **Each row should be evaluated in parallel**
  - Estrin's assumes N processing units
  - We only have four-way SIMD instructions
  - We will still have serialization stalls
- **Many architectures don't allow interaction between vector elements**
  - We have to split calculations across SIMD registers
  - We can often use redundant calculations to speed evaluation

Brute Force Searching

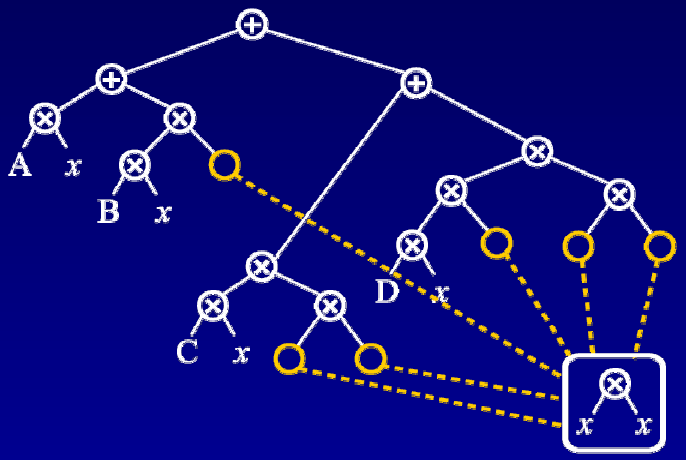
- **One tool can find the perfect evaluation order for every architecture – Brute Force**
  - We generate trees of expressions
  - Flatten the tree into an ordered sequence of instructions
  - Evaluate the sequence w.r.t a particular architecture
- **Each run can generate millions of trees to test**
  - Run the program overnight, keeping the best result so far
  - Distribute the calculation over many machines

Expression Tree

$$f(x) = Ax + Bx^3 + Cx^5 + Dx^7$$



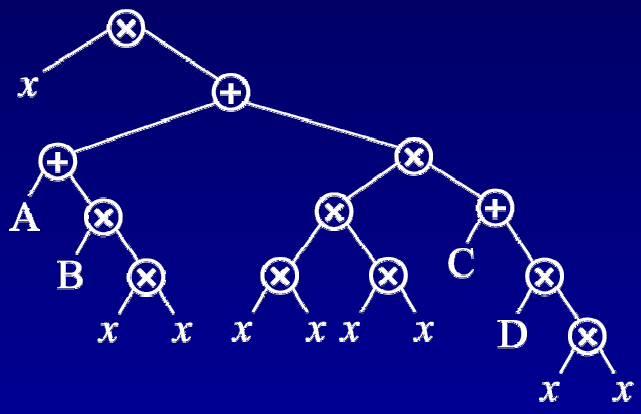
Common Expression Removal



Tree After Rotations

$$f(x) = x\Big(\big(A + Bz\big) + z^2\big(C + Dz\big)\Big)$$

where $z = x^2$



Coding Using SIMD

- **Using SIMD instructions to code this**
  - Temptation is to gather all coefficients into a 4-vector
  - Calculate [x, x^3, x^5, x^7] and do one big multiply
  - Finally sum the result

```
...
calculate [ x, x^3, x^5, x^7 ]
...
N :=    [ A, B , C , D  ]
     * [ x, x^3, x^5, x^7 ]
...
sum the elements of N
...
```

Coding With SIMD

- **This turns out to be very slow**
  - You are only using one parallel instruction!
  - Better to think of calculating in two pairs

```
...
calculate [ x, x², x, x² ]
...
N :=    [ A, B, C, D ]
     * [ x, x², x, x² ]

N :=    [ Ax, Bx², Cx, Dx² ]
     * [  -,  -,  x²,  x² ]
...
sum the elements of N
...
```

Summing Elements

- **Summing the elements of a vector is very serial if you don't have a special instruction**
  - Better to split the problem over two registers.

```
A.x = A.x + A.y
              .
              .
              .
A.x = A.x + A.z
              .
              .
              .
A.x = A.x + A.w
              .
              .
              .
      Finished.
```

```
A.x = A.x + A.y
B.x = B.x + B.y
.
.
.
A.x = A.x + B.x
.
.
.
Finished.
```

SIMD Insights

- **The more work you have to do, the more opportunities for parallelisation**
  - Easier to code higher power polynomials as O(log n) starts to be more and more effective
  - Use redundant elements as "free duplicate values" to avoid moving elements around
  - You end up distributing coefficients in small groups
- **SinCos a good example**
  - More efficient to do both sine and cosine at the same time

```
Coeff1 := [A, A, E, E]
Coeff2 := [B, B, F, F]
Coeff3 := [C, C, G, G]
Coeff4 := [D, D, H, H]
```

Higher Order Functions

More Transcendental Functions

- **What about the other Transcendentals?**
  - Tangent, Arctangent, Logarithm, Exponent, Roots, Powers
- **Let's take a look at how well they are approximated by polynomials**
  - For each power, make an approximation (e.g. minimax between [0..1])
  - Calculate the maximum error within the range
  - Find the negative base-2 log of the error
- **This value tells us how many bits of accuracy we have for each approximation**

Table Of Error In Bits

	2	3	4	5	6	7	8
Exp(x)	6.8	10.8	15.1	19.8	24.6	29.6	34.7
Sin(x)	7.8	12.7	16.1	21.6	25.5	31.3	35.7
Ln(1+x)	8.2	11.1	14.0	16.8	19.6	22.3	25.0
Atan(x)	8.7	9.8	13.2	15.5	17.2	21.2	22.3
Tan(x)	4.8	6.9	8.9	10.9	12.9	14.9	16.9
Asin(x)	3.4	4.0	4.4	4.7	4.9	5.1	5.3
Sqrt(x)	3.9	4.4	4.8	5.2	5.4	5.6	5.8

Interpretation

- **Sine and Exponent give us 4 bits per power**
  - Will only need a 7^{th} order approximation for single floats
- **Why are tangent and arcsine so badly behaved?**
  - We will find out shortly…
- **Square Root will never be approximated by a polynomial.**
  - Less than half a bit of accuracy per power
  - We must use Newton's Method with good initial guesses

Hardware Square Root

- **You will never out perform a hardware square root**
  - To avoid serializing the Newtons Iterations hardware designs use a redundant *signed-digit* number system.

$$
\begin{aligned}
637 &= 6\times10^2 + 3\times10^1 + 7\times10^0 \\
&= 1\times10^3 - 4\times10^2 + 4\times10^1 - 3\times10^0 = 1\bar{4}4\bar{3} \\
&= 1\times10^3 - 4\times10^2 + 3\times10^1 + 7\times10^0 = 1\bar{4}37
\end{aligned}
$$

  - Carry propagation happens only once, at the end of the calculation

Tangent

Tangent Definitions

- **Tangent is defined as the ratio of sine over cosine**

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

- **For infrequent use this calculation is good enough**
  - Remember to test for cos(x)=0 and return BIGNUM
  - Has twice the inaccuracy of sin or cosine

Using Trig Identities

- **We must range reduce as much as possible**
  - Using trigonometric identities to help us

$$\tan(-x) = -\tan(x)$$

- **We record the sign, force x to positive**
  - We will replace the sign during reconstruction
  - We have halved our input range

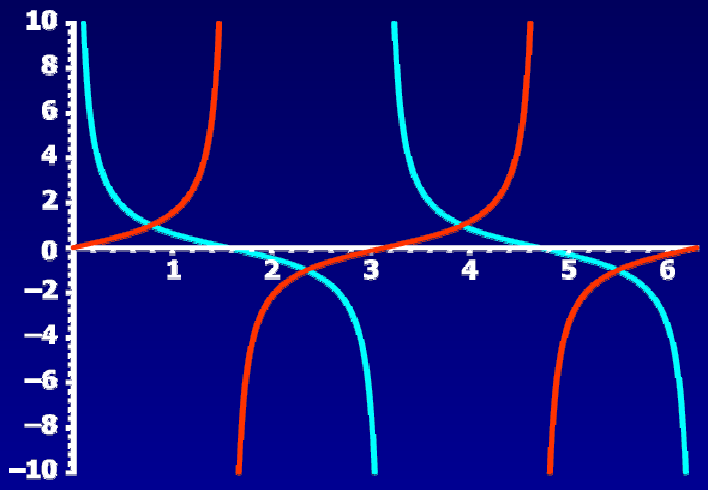
Range Reduction

- **The next identity is the fun one**

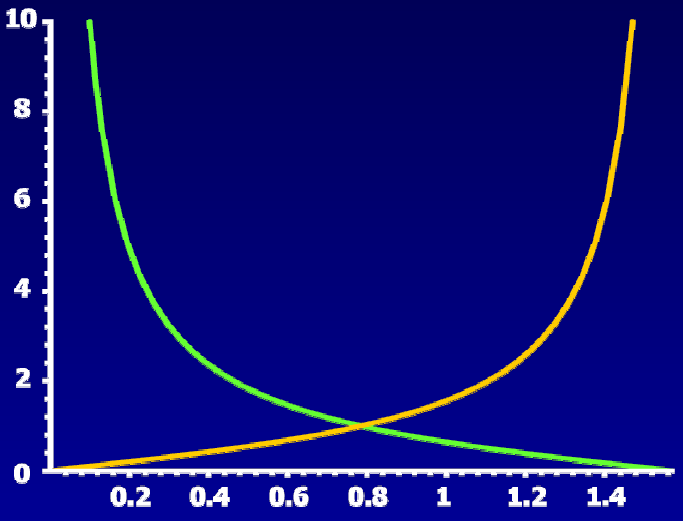
$$\tan(x) = \frac{1}{\tan\left(\frac{\pi}{2} - x\right)}$$
$$= \cot(\pi/2 - x)$$

- **This tells us two things**
  1. The range 0..Pi/2 is repeated over and over.
  2. There is a sweet spot at Pi/4 where both sides are equal.

Tangent and Cotangent



Tangent and Cotangent Detail



Minimax Approximations

- **Tangent between 0..Pi/4 is simple enough**
  - Looking at the Taylor Series, it uses odd powers.

$$\tan(x) \approx x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \ldots$$

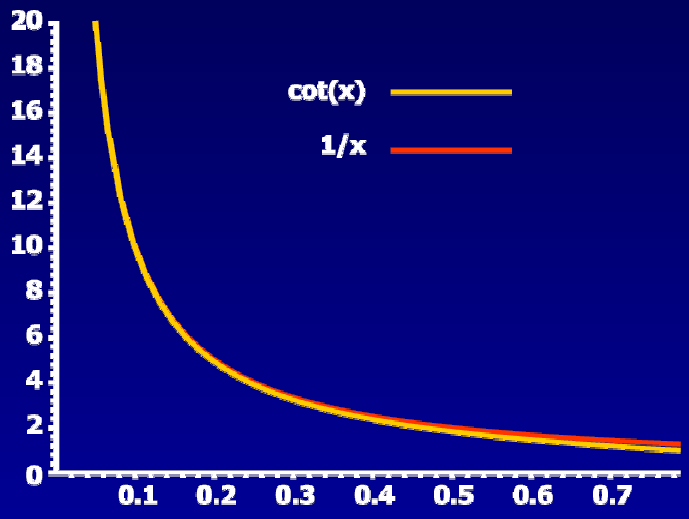
  - Apart from being strangely resistant to polynomial approximation, it is a straightforward minimax.

$$\tan(x) \approx x + x^3 P\left(x^2\right)$$

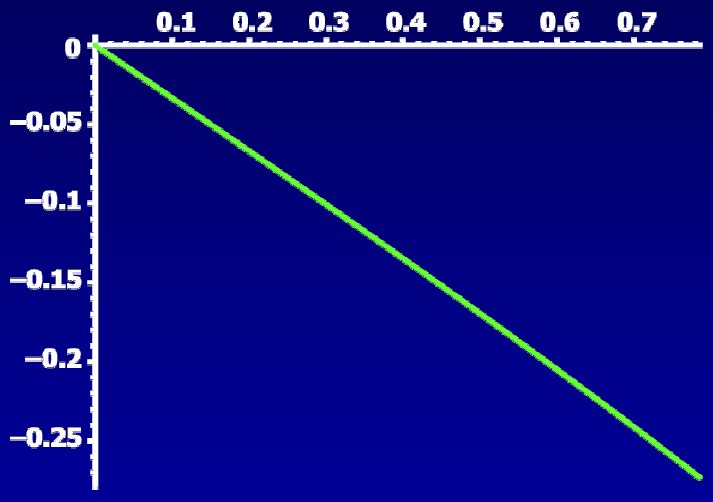
Cotangent Minimax

- **But how to approximate Cotangent?**
  - It has infinite values and high curvature
- **There is a trick using small angles**
  - The arctangent is just cosine/sine
  - As $x \to 0$ the sine $\to x$ and cosine $\to 1$
  - Meaning that as $x \to 0$ then cotangent $\to 1/x$

Cotangent and 1/x



Cotangent Minus 1/x



Cotangent Minimax

- **We can use a different minimax form**

$$\cot(x) \approx \frac{1}{x} + xP\left(x^2\right)$$

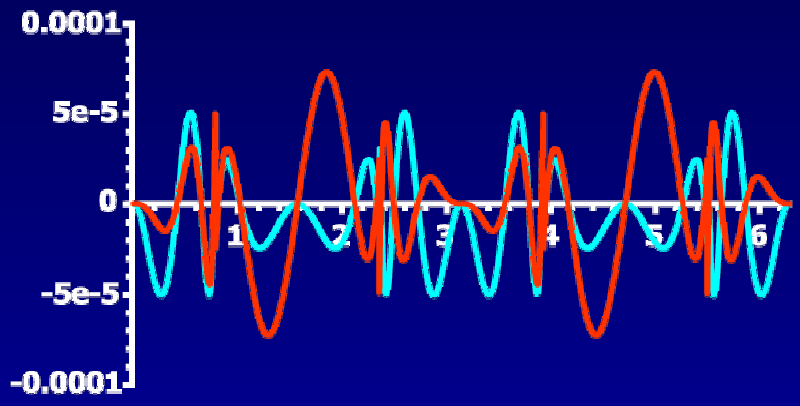
- **This has an interesting property**
  - The reciprocal is based on nothing but x
  - We can start a reciprocal instruction way up the pipeline and use the result after calculating the difference polynomial

Final Tangent Form

- **The final form of the tangent function**

$$\tan(x) = \begin{cases} \tan(x) & 0 \le x < \frac{\pi}{4} \\ \cot(x - \pi/4) & \frac{\pi}{4} \le x < \frac{\pi}{2} \\ \cot(x) & \frac{\pi}{2} \le x < \frac{3\pi}{4} \\ \tan(x - \pi/4) & \frac{3\pi}{4} \le x < \pi \end{cases}$$

Tangent Error Plot 0..2Pi



Rational Polynomials

- **Minimax can also produce rational polynomials**
  - A pair of polynomials that are divided by each other to produce the final result

$$\tan(x) \approx \frac{xP(x^2)}{Q(x^2)}$$

- **Rational Polynomials can be very accurate for low powers**
  - If you can take the cost of a divide or reciprocal at the end of the approximation process
  - Popular for high accuracy functions

Rational Tangent

- **Rational approximation to tangent**
  - Gives 1.3e-8 accuracy using fewer operations

$$\frac{1.15070768x - 0.110238971x^3}{1.15070770 - 0.49380897x^2 + 0.0111809593x^4}$$

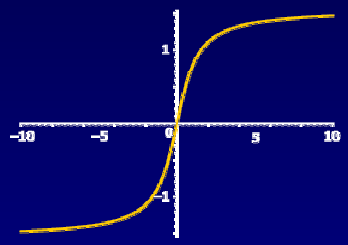
  - Can improve this by dividing through by denominator constant, saving us a load-constant instruction

$$\frac{0.999999986x - 0.0958010197x^3}{1 - 0.429135022x^2 + 0.00971659383x^4}$$

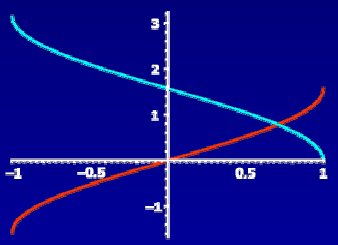
Arctangent

Inverse Trig Functions

- **Arctangent**
  - The inverse of a tangent
  - Maps [–inf..+inf] to [–π/2..π/2]



- **Arcsine and Arccosine**
  - Asin maps [-1..+1] to [–π/2..π/2]
  - Acos maps [-1..+1] to [π..0]



Why only Arctangent?

- **Atan, Acos and Asin are closely related**

$$\arccos(x) = \arctan\left(\frac{\sqrt{1-x^2}}{x}\right)$$

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x)$$

$$= 2\arcsin\left(\sqrt{\frac{1-x}{2}}\right)$$

Range Reduction

- **Arctan has these interesting properties**

$$\arctan(-x) = -\arctan(x)$$

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$$

- **We only need to define arctan between 0..1**
  - That is one huge range reduction

Arctan Taylor Series

- **The Taylor series for Arctangent looks strangely familiar, yet different…**
  - It converges very, very slowly
  - Doesn't even converge for |x|>1

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

  - For 1e-8 accuracy you need to evaluate **7071 terms**!
  - We need polynomial approximations to make the function a viable calculation

Minimax Inequality

- **The minimax form is very straightforward**

$$\arctan(x) = x + x^3 P(x^2)$$

- **But gives us very poor accuracy**
  - A 13th power minimax approximation over 0..1 gives a maximal error of 0.9!
  - We need to reduce the range a lot more

Double Angle Formulation

- **Arctangent is the inverse of a tangent**
  - So let's assume x = tan(a+b) so that arctan(x) = a+b. What are a and b ?

$$x = \tan(a+b) = \frac{\tan(a)+\tan(b)}{1-\tan(a)\tan(b)}$$

$$z = \tan(a) \Rightarrow a = \arctan(z)$$

$$k = \tan(b) \Rightarrow b = \arctan(k)$$

$$x = \frac{z+k}{1-zk} \Rightarrow z = \frac{x-k}{1+kx}$$

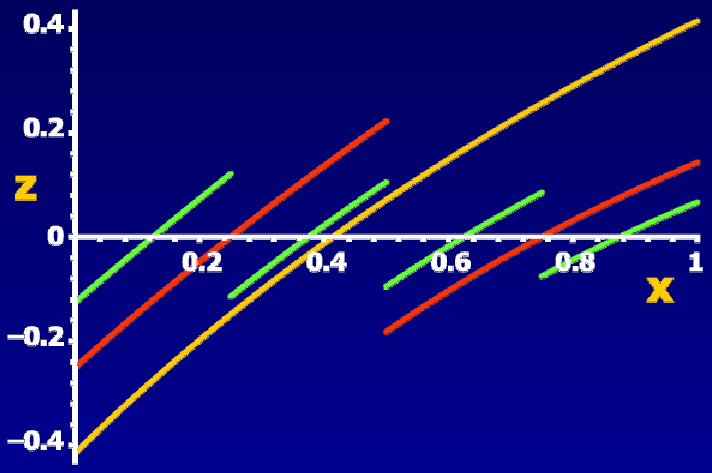
$$\begin{aligned}\arctan(x) &= \arctan(k)+\arctan(z)\\ &= b+\arctan(z)\end{aligned}$$

The X-Z Mapping

$$\arctan(x) = b + \arctan\left(\frac{x-k}{1+kx}\right)$$

- **How do we use this?**
  - It remaps input x into the smaller z-range
  - We can subdivide 0..1 into, say, 2 segments, each with their own b offset and k parameter.
- **What value of k to choose?**
  - If z(0) = -z(1) we have equal distribution either side of the graph, which happens when k = sqrt(2)-1
  - Or divide by equal input values – ¼, ½, ¾

Some X-Z Mappings



Designing Your Function

- **Now you have a set of tools to play with:**
  - Range reduction to 0..1
  - Compliment if x>1.0
  - Choice of X-Z map
    - Equal-Angle maps are most accurate
    - Equal-Input maps are faster to range reduce
  - Choice of minimax order

```
float arctan(x)
{
  float k,b,sign = +1.0f;
  bool compl = false;
  // remove sign
  if(x<0.0) { s = -s; y = -y; }
  // compliment if x>1.0
  if(x>1.0) { compl = true;
              x = 1/x;}
  // x-z range reduce
  if(x<0.5) { k = 0.25f;
              b = arctan(k); }
  else { k = 0.75f;
         b = arctan(k); }
  y = (y-k)/(1+k*y);
  // poly approx
  y = b + arctan_poly(x);
  // finish compliment
  if(compl) y:=Pi/2 - y;
  // reinstate sign
  return y*sign;
}
```

Using Arctan for Arccosine

- **The ideal example for using IEEE754 flags**
  - To convert from arctan to arccosine we need to cope as x tends towards zero

$$\arccos(x) = \arctan\left(\frac{\sqrt{1-x^2}}{x}\right)$$

  - Arccos(0.0) = Arctan(+inf) = Pi/2
  - We must remember to reset the divide-by-zero flag before returning from arccos(0.0)

Exponent

Exponent: The Anti-Log

- **All exponents can be expressed as powers of the Euler Constant.**

$$10 = e^{\ln 10} \quad \Rightarrow \quad 10^x = \left(e^{\ln 10}\right)^x$$

$$= e^{x \ln 10}$$

$$2 = e^{\ln 2} \quad \Rightarrow \quad 2^x = e^{x \ln 2}$$

Range Reduction

- **Exponent also uses *additive range reduction*, much like sine and cosine**

$$x = N\ln(2)/2 + r$$
$$= NC + r$$

- **This means we break x into two parts**
  - 2 raised to some exponent plus a fractional residual.
  - Which floating point numbers already do!

$$\exp(x) = 2^{\frac{N}{2}} e^{r}$$

Polynomial Approximation

- **Looking at the Taylor series**

$$\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots$$

- **We can subtract the 1 and minimax it over the range [0..ln(2)/2] with**

$$\exp(x) - 1 \approx x + x^2 P(x)$$

The Exponent Function

- **Two methods of coding**

  1. Breaking your input into exponent and mantissa and reassembling at the end

  2. Using tables for 2^N to keep all work in float registers, but only supporting a small range of input

```
float fast_exp(x)
{
  const float C = 2.f/ln(2);
  const float iC = ln(2)/2.f;
  // range reduction
  int N = (int)(x*C);
  float y = x-N*invC;
  // approximation
  y = exp_poly(y) + 1.0f;
  // reconstruction
  return power2[N>>1]*y;
}
```

Semi-Table Based Reconstruction

- **New research has better methods for reconstructing the power of two**
  - There are a family of algorithms by Tang where K > 1 in

$$\exp(x) = 2^{\frac{N}{2^K}} e^r$$

  - To reconstruct, say, $2^{N/128}$ we calculate

$$N = 128M + 16K + J$$

$$2^{\frac{N}{128}} = 2^M \times 2^{\frac{K}{8}} \times 2^{\frac{J}{128}}$$

  - Where the last two terms are from 8 and 16 entry tables

Fast Float Exponent

- **How about just constructing a float directly from an integer power?**
  - Remember how the IEEE exponent works.

```
01011100001000000000000000000000
```

  - float = sign * $2^{(\text{exponent} + 127)}$ * 1.mantissa
  - Note also that :

$$e^x = 2^{\frac{x}{\ln(2)}} = 2^{\frac{1}{\ln(2)}x}$$
$$= 2^{kx}$$

Fast Float Exponent

- Take an integer in [-126..127], e.g. 12

```
00000000000000000000000000001100
```

- Integer multiply it by integer 2^{23} / ln(2)

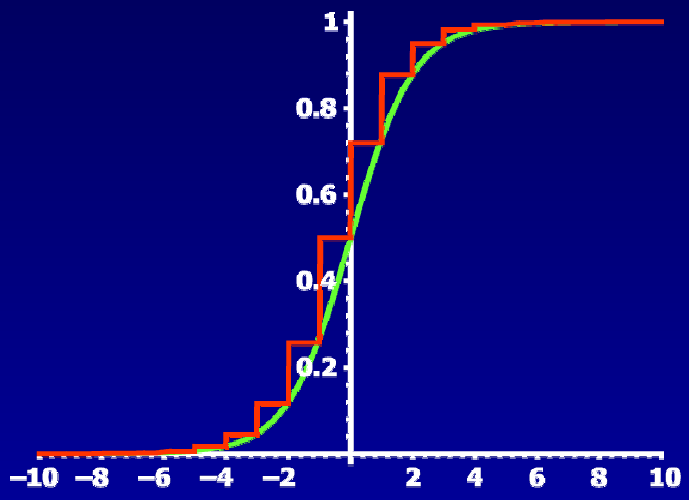
```
* 00000000101110001010101000111011
= 00001000101001111111101011000101
```

- Add in the bias offset integer 127 * 2^{23}

```
+ 00111111100000000000000000000000
= 01001000001001111111101011000110
= 172011.09375
```

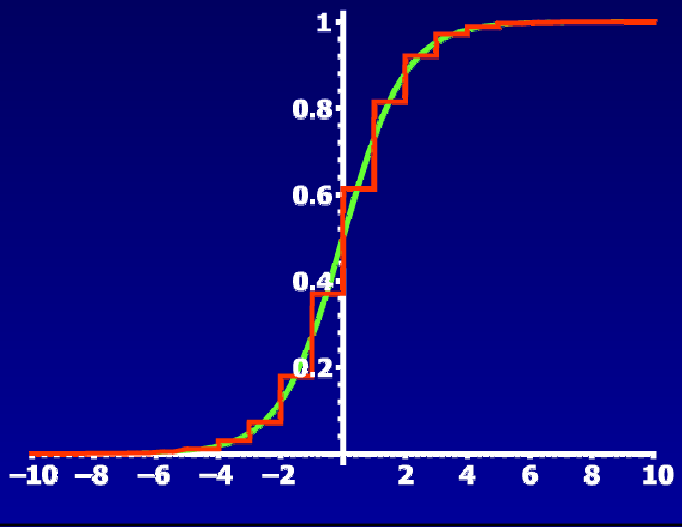
Fast Float Exponent

- Plotting the function 1/(1-exp(-x)) shows an offset:



Fast Exponent Fixed

- Subtracting $0.7*2^{23} = 5872025$ removes the offset



Increasing Resolution

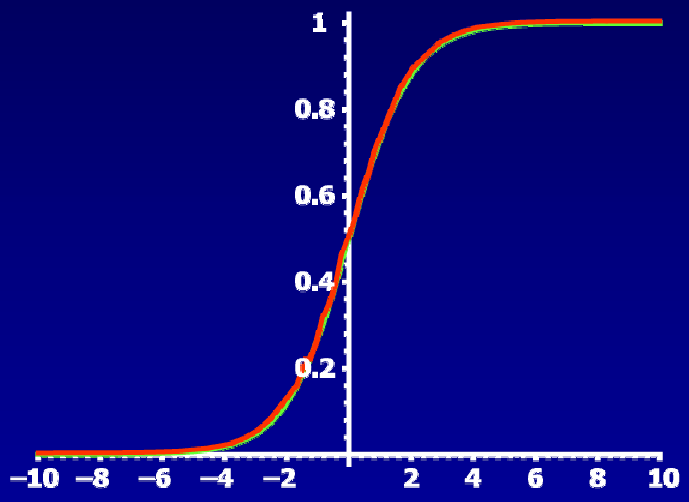
- **The problem with single precision floats**
  - there are only 255 exponent values we can use hence the staircase effect.

- **Solution:**
  - Allow the input power to have some fractional bits.
  - Take float, multiply it by $2^3 = 8$ and floor it, e.g. 12.125 = 97

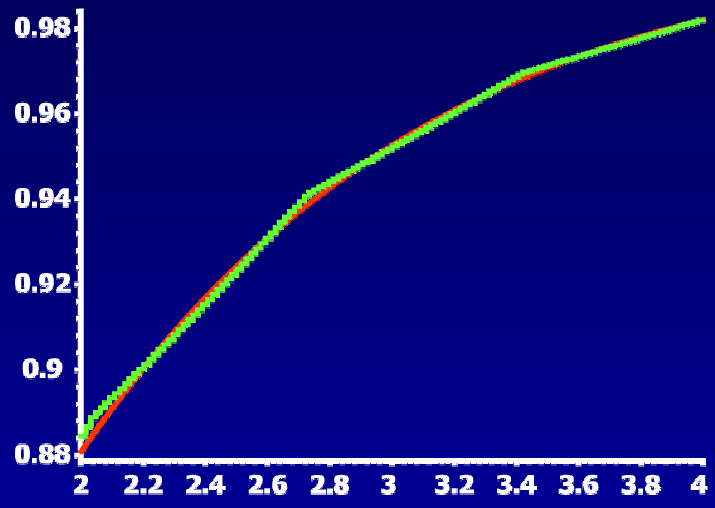
    00000000000000000000**00001100001**

  - Multiply by $2^{(23-3)}$ / ln(2), add IEEE bias $127*2^{23}$
  - Pick how many fractional bits you want, but the bias changes each time
    - e.g. if bits = 6 then bias = $3*2^{17}$

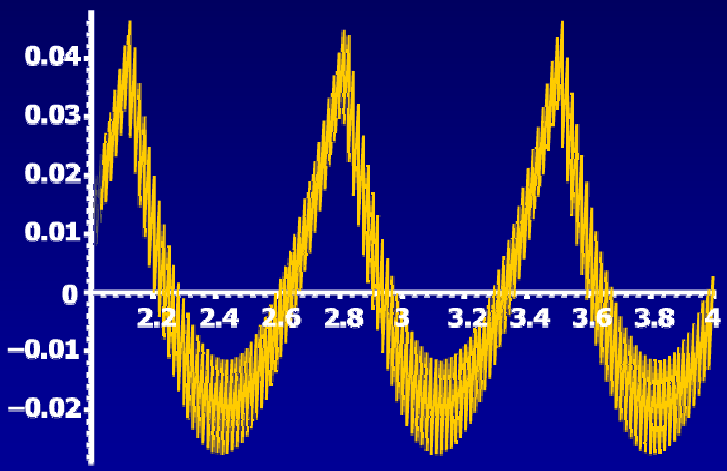
High Resolution Fast Exponent



High Res Fast Exponent Detail



High Res Relative Error



Logarithm

Logarithms

- **Ln(x) returns exponents**
  - Defined in the range [0..+infinity]
- **There are two main logarithm functions**
  - log(x) refers to the logarithm base 10
  - ln(x) refers to the *Natural Logarithm* base *e*
- **They are freely convertible**

$$\log(x) = \frac{\ln(x)}{\ln(10)}$$

Taylor Series

- **The Taylor Series comes in two forms**
  - First is closely related to the arctangent series

$$\ln(x) = 2\left(z + \frac{z^3}{3} + \frac{z^5}{5} + \frac{z^7}{7} + \frac{z^{11}}{11}\ldots\right) \quad \text{where } z = \frac{x-1}{x+1}$$

  - Second is simpler but converges as slowly

$$\ln(x+1) = x - \frac{z^2}{2} + \frac{z^3}{3} - \frac{z^4}{4} + \frac{z^5}{5} - \frac{z^6}{6}\ldots$$

Range Reduction

- **Range reduction for ln uses *multiplicative range reduction***
  - Given that we store floating point numbers as exponent and mantissa

$$x = sign \times 2^n \times f \quad \text{where } 1.0 \le f < 2.0$$

  - we find that the logarithm of such a number is

$$\begin{aligned} \ln(x) &= \ln\left(2^n \times f\right) \\ &= \ln\left(2^n\right) + \ln f \\ &= n \ln 2 + \ln f \end{aligned}$$

Dissecting Floats

- **How do we split a float into exponent and mantissa?**
  - On a PC it's easy, you just mask and shift bits.
  - Remember to subtract 127 from the exponent.
- **On the PS2 Vector Units, we have a problem.**
  - Only 16 bit integer registers – not enough to hold even the entire mantissa.
  - No way to shift the exponent into lower bits
  - Float-to-int conversion loses the exponent
- **But there is a trick…**

Extracting The Exponent

- **What happens if you convert a float to a float?**
  - Pretend the contents of a floating-point register is an integer value and convert this "integer" into a float.
  - For example, take the value **2^{57}*1.125**
  - sign = 0
    exponent = 57 + 127 = 184
    mantissa = 1.125

`01011100001000000000000000000000`

  - This bit pattern represents the integer 1544552448

Extracting The Exponent

- Multiply 1544552448 by $1/2^{23}$ to rescale the decimal point

$$1544552448 \times \frac{1}{2^{23}} = 184.125$$

- Subtract the exponent bias of 127

$$184.125 - 127 = 57.125$$

- Convert to integer to obtain the true exponent

A Mathematical Oddity: Bitlog

- **A real mathematical oddity**
  - The integer $\log_2$ of a 16 bit integer
  - Given an N-bit value, locate the leftmost nonzero bit.
  - b = the bitwise position of this bit, where 0 = LSB.
  - n = the **NEXT** three bits (ignoring the highest 1)

$$\text{bitlog}(x) = 8 \times (b - 1) + n$$

- **Bitlog is exactly 8 times larger than $\log_2(x)$-1**

Bitlog Example

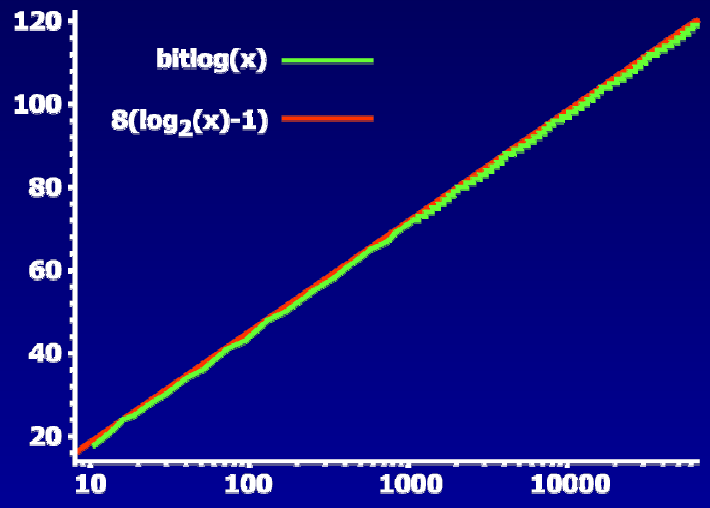
- **For example take the number 88**

88 = 1011000_2
b = 6^{th} bit
n = 011_2 = 3
bitlog(88) = 8*(6-1)+3
= **43**

- (43/8)+1 = 6.375
- Log2(88) = 6.4594…

- **This relationship holds down to bitlog(8)**

Bitlog Graph



X To The Power N

Integer Powers

- **Raising to an integer power**
  - multiply all powers of x where N has a bit set

$$N = 23 = 10111_2$$

$$x^{23} = x^{16} \times x^4 \times x^2 \times x$$

- **If N<0 then calculate 1/result**
- **If x<0 then record sign and reinstate later**
  - Note that odd powers of –ve number lose the sign

Integer Power Function

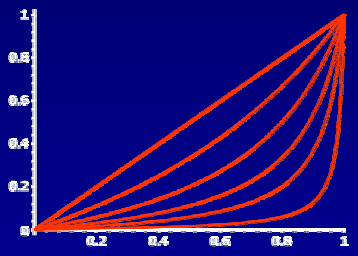
- **As simple as it sounds**
  - Loop through bits of N
  - Keep squaring x each loop
- **Careful noting and reinstating signs**

```
float powi(float x,int N)
{
  float y,w;
  bool ns = false,xs = false;
  // record signs
  if(N<0) ns = true, N = -N;
  if(x<0.0f) xs = true, x = -x;
  // setup initial powers
  if(N&1) y = x;
  else y = 1.0f, xs = false;
  w = x; N >>= 1;
  while(N) {
    w = w * w;
    if(N&1) y *= w;
    N >>= 1;
  }
  // reinstate signs
  if(xs) y = -y;
  if(ns) y = 1.0f/y;
}
```

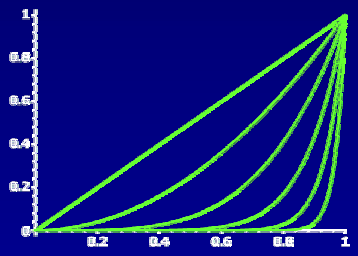
X^N Approximation

- **From Graphics Gems 4**

$$pow(x,n) \approx \frac{x}{n - nx + x} \qquad \text{for } 0 \le x \le 1$$



approximation



actual

Raising to a Power

- **The canonical method is to do the calculation in two parts – log and exponent**

$$\begin{aligned}\mathrm{pow}(x,N) &= e^{N\ln x} \\ &= 2^{N\log_2 x}\end{aligned}$$

  - Range reduction only needs to be done once.
  - MUST use extended accuracy in internal workings.
    - Log2(x) must have 8+ extra bits of accuracy
  - The most expensive operation in the toolbox.
  - Also the most difficult to write.
    - e.g. calculating 56^{156} naively gives an error of 1052 ULPS!

Floating Point Power Hack

- **The log and power can be faked**
  - If we force a float through the int-to-float conversion, the result is approximately logarithmic.

$$\mathrm{int\_to\_float}(x) \approx A\log_2 x + B$$

  - And conversely the inverse operation approximates

$$\mathrm{float\_to\_int}(x) \approx 2^{\frac{x-B}{A}}$$

- **Credited to Matthew Jones, Infogrames**

Expanding Out

- **Combine these two operations to approximate the power function**

$$x^N \approx 2^{N\frac{(A\log_2 x+B)-B}{A}}$$

$$= 2^{N\log_2 x+\frac{(N-1)B}{A}}$$

  - So we have an extra term to subtract before we can use the approximate power.

Floating Point Power Hack

- **So, the algorithm goes like this**
  - First, convert our float into logarithmic space

```
float a = float_as_int(x);
```

  - We multiply this value by the power we want to raise it to.

```
a = a * power;
```

  - Then we subtract the **magic constant**

```
magic = float_as_int(1.0f) * (power-1.0f);
a = a - magic;
```

  - And finally reconvert back to a float

```
a = int_as_float(x);
```

Scaling the Result

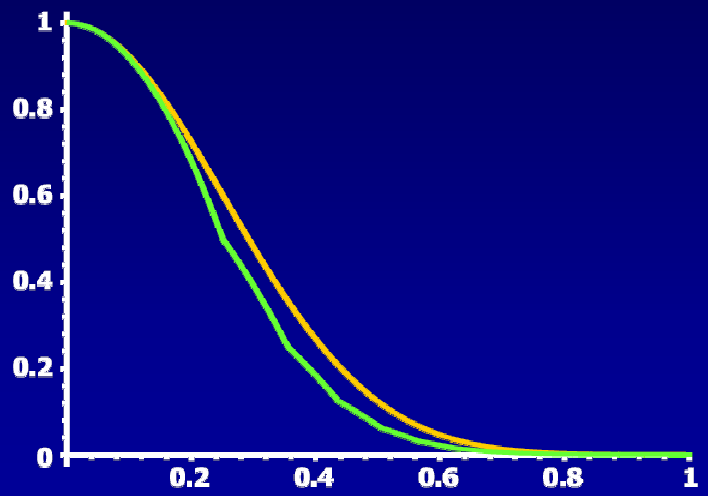
- **We can scale the resulting value**
  - by reformulating the magic number

```
magic = float_as_int(1.0f) * (power-1.0f);
a = a - magic;
```

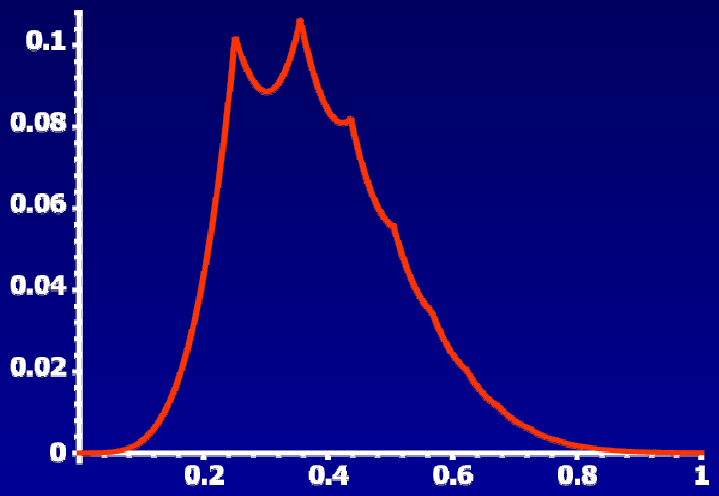
  - Can be rewritten to give us a 255 * x^N for no additional work

```
magic = float_as_int(255.0f) -
        float_as_int(1.0f) * power;
a = a + magic;
```

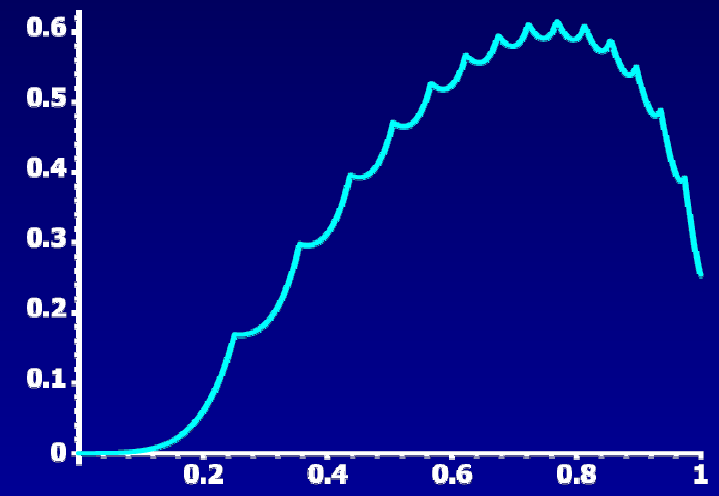
$Cos^{16}(x)$ Graph



$Cos^{16}(x)$ Absolute Error

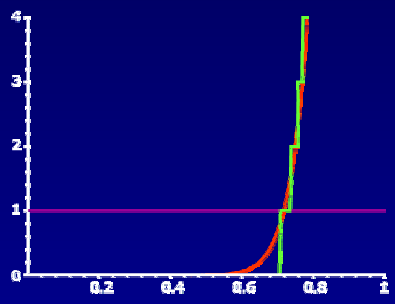


$Cos^{16}(x)$ Relative Error



Another Approach

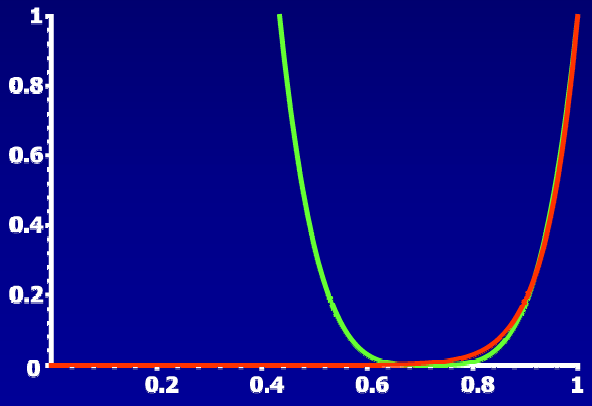
- **An interesting effect**
  - When raising pixel intensities in the range [0..255] to a power, for most of the range the value is less than one intensity level



  - As noted by Beaudoin & Guardado in "Direct3D ShaderX: Vertex and Pixel Tips and Tricks"

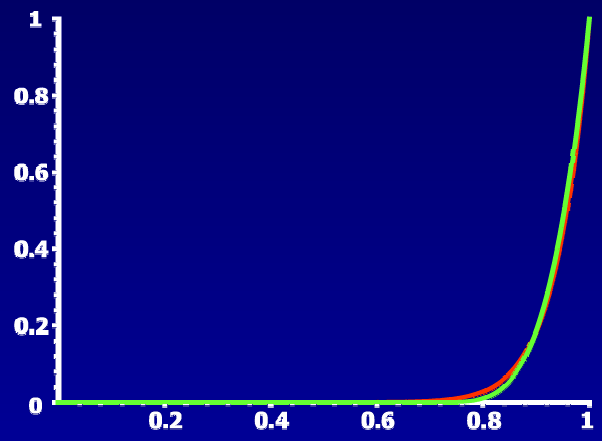
Low Power Approximation

- **Higher powers look a bit like lower powers scaled and offset**
  - Example: Using x^4 to approximate x^{16}



Low Power Approximation

- Our function is zero until it is greater than one part in 255



Low Power Approximation

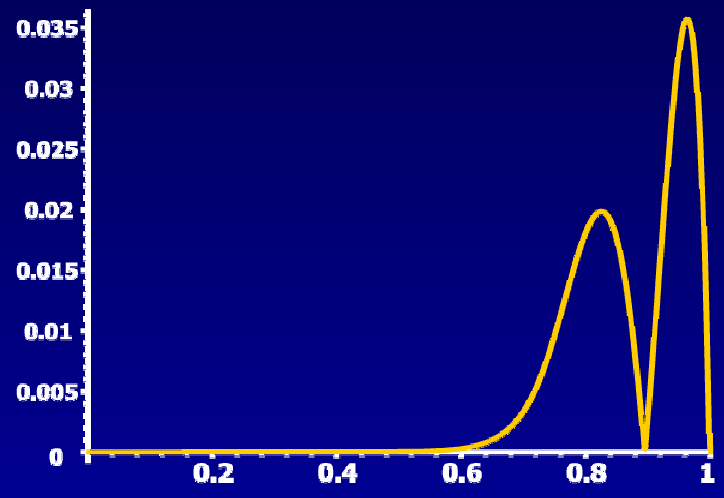
- New power approximation is:

$$x^m \approx \max(ax+b,0)^n$$

  - Where n<<m and a and b are:

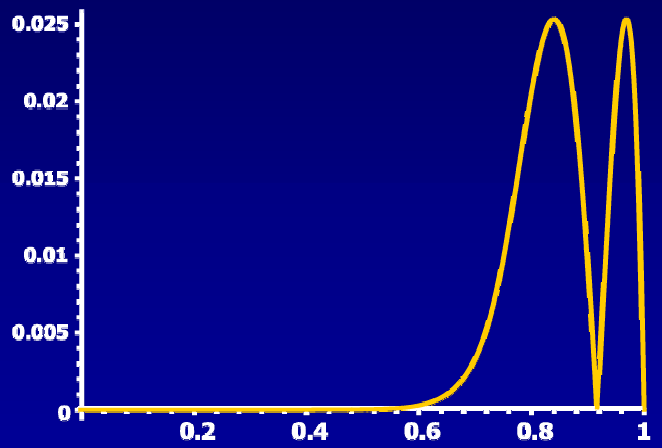
$$a = \frac{1}{1-256^{1/n}}$$
$$b = 1-a$$

Low Power Approx Error



Improved Error

- Using a binary search, improved values of a and b can minimize the maximal error

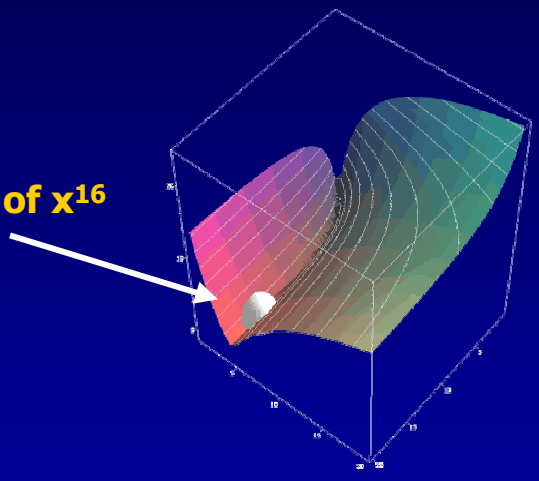


Low Power Approx Problem…

- **There is however a problem**
  - The paper doesn't seem to have obvious "best" values for a and b.
    - The results are just presented as a table of values with errors. The choice is already made for you.
  - Closer powers don't seem to reduce the error
    - so there is no obvious way to control error, relative or absolute.
- **What is going on?**
  - Let's plot the maximal error for every power approximating every other as a surface…

Low Power Approx Problem…

x^4 approx of x^{16}



- The original example was a global minimum!

Conclusions

Conclusions

- **Get to know IEEE754**
  - It's being updated as we speak
  - If your implementation differs, you will understand it's weaknesses
- **Always test both absolute and relative error**
  - Functions may look accurate on a graph
  - Relative error will show the tough weaknesses

Conclusions

- **Incremental Algorithms**
  - Used for generating sequences of trig operations
  - Some of the fastest techniques around
- **Table Based Algorithms**
  - Don't discount them just yet
  - Tables are usually far too big – check the size
  - Use range reduction and reconstruction on tables

Conclusions

- **Polynomial Approximations**
  - Some of the most flexible techniques around
  - Range Reduce, Approximate, Reconstruct
  - Choose between
    - Aggressive or simple range reduction
    - Polynomial order
    - Parallel polynomial evaluation techniques
    - Expensive or simple reconstruction
    - Semi table-based reconstruction
  - Use Maple or Mathematica to your advantage

Conclusions

- **New techniques**
  - Floating Point Hacks
    - An exciting new set of techniques
  - Gaining support form hardware designers
    - IA64 architecture uses 8-bit mantissa lookup for 1/x
  - Combine with semi table-based reconstruction
    - Accuracy over a limited range
    - Research to be done on accurate approximations

References

- **Many of the original books are out of print**
  - Researchers work with piles of photocopies!

- Hart, J.F., ***Computer Approximations***, John Wiley & Sons, 1968
- Cody & Waite, ***Software Manual for the Elementary Functions***, Prentice Hall, 1980
- Goldberg, Steve, ***What Every Computer Scientist Should Know About Floating Point Arithmetic***, ACM Computing Surveys, March, 1991
- Muller, J.M., ***Elementary Functions: Algorithms and Implementations***, Birkhaüser, 1997
- Tang, Ping Tak Peter, ***Table Lookup Algorithms for Elementary Functions and Their Error Analysis***, Proceedings of 10th Symposium on Computer Arithmetic, 1991

Website

- **Notes available from:**

http://research.scea.com/

robin_green@playstation.sony.com

Questions?