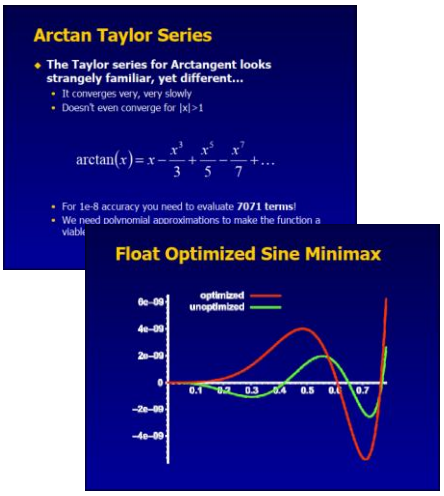

GDC
Even Faster Math Functions
Robin Green
Programmer, Pacific Light & Hologram
GAME DEVELOPERS CONFERENCE
MARCH 16–20, 2020 | #GDC20

Time for an update

- Since we last talked at GDC 2002, numerical research has progressed
- Error analysis has greatly improved, we now aim for “good to the last bit”
- Tools have improved, become open source and free
- Frankly, we are all quite a bit older now



https://basesandframes.wordpress.com/2016/05/17/faster-math-functions/

The last time we talked about implementing math functions at GDC it was 2002

Research has progressed in the last 10 years, especially “last bit” accuracy with the aim of standardizing math functions

Much of this research has happened as open source, available to anyone interested in using them

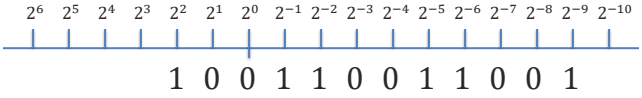
So it’s time for an update

And if you have the slides (oh hai!) check out the unobtrusive links at the bottom-right for the original sources and refences

Why care about FPGA?

- FPGA is the coming storm
  - FPGA enables wide parallelism and expects bit-level computation
  - Floating point is a compressed memory encoding for place-value bit strings
  - Even in FP code, once you have range reduced you're doing fixed point

$$x = \sum_{i=-p}^{N-1} x_i 2^i$$



Where there was no `libm`, we made our own

With FPGA you start with no operators

In the teaser for this course I mentioned FPGA as being an area of interest

Why on earth should game devs care about the world of hardware?

Because FPGA is the coming storm

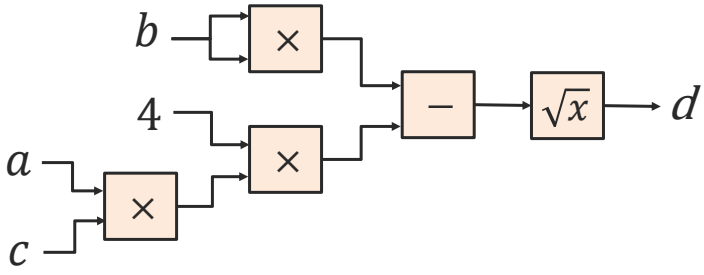
- You want to make magical things with your conference badge?
- Simulate 32-bit graphics hardware on-chip for a handheld?
- Costs are coming down, soft SOCs are being generated from open source projects

IEEE754 floats are a compressed memory storage format for bit strings

Fixed point being used for AI training has led to a resurgence in interest for math functions the cover the full dynamic range to the last bit

Thinking in gates

- Let's examine calculating the quadratic discriminant $d = \sqrt{b^2 - 4ac}$
- Converting CPU ops into circuitry is simple enough

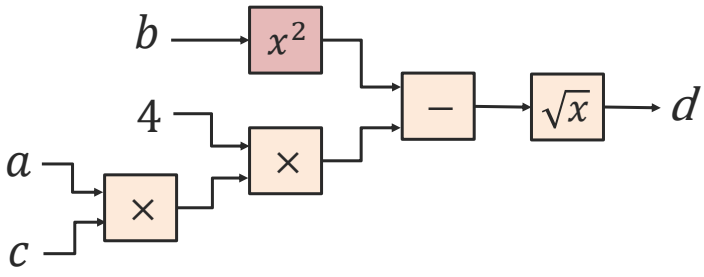


A quick thought experiment, a simple translation of the quadratic discriminant into FPGA would take the ops, fit to standard vendor libraries and gain a little parallelism.

But we can do a LOT better – stop thinking in CPU ops

Thinking in gates

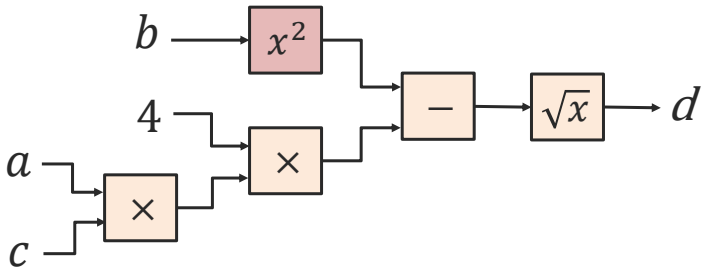
- Squaring is a special case of multiplication



Squaring is a special case of multiplication

Thinking in gates

- Squaring is a special case of multiplication

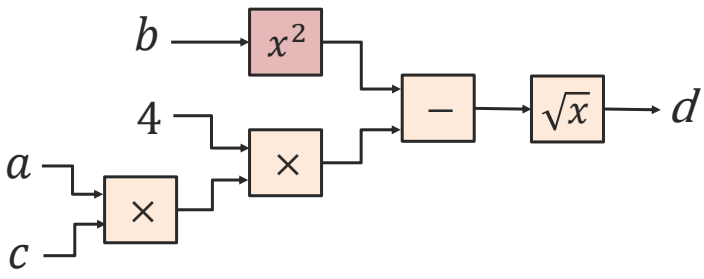



```
   2321
x  2321
   2321
  46420
 696300
4642000
5387041
```


Looking at the long form of the multiplication there is symmetry along this diagonal

Thinking in gates

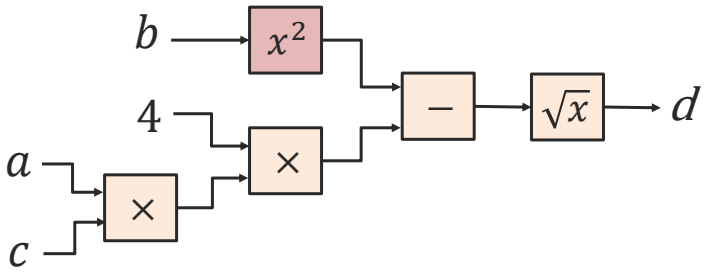
- Squaring is a special case of multiplication




```
      2321
x     2321
      2321
     46420
    696300
   4642000
   5387041
```

Thinking in gates

- Squaring is a special case of multiplication

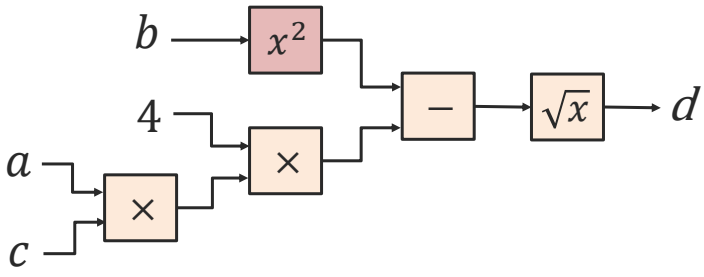



```
   2321
x  2321
   2321
  46400
 690000
4000000
```


We keep the first digit …

Thinking in gates

- Squaring is a special case of multiplication

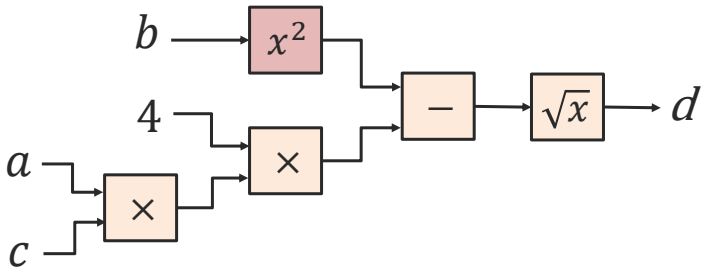



```
   2321
x  2321
   4641
  92400
1290000
4000000
```


Double the remaining digits ...

Thinking in gates

- Squaring is a special case of multiplication




```
    2321
x   2321
----------
    4641
   92400
  129000
 4000000
----------
 5387041
```


...then sum with carry

Computer Science books often show examples like this in base 10 to justify algorithms (HATE this)

So is algorithm true? Let's follow an on-paper experiment using bits

1. to illustrate the raw size of a naïve multiply unit
2. to get comfortable with this bit visualization
   it will come in handy later

Squaring in binary

```
1  ....................100100010001
0  ...................000000000000.
0  ..................000000000000..
0  .................000000000000...
1  ................100100010001....
0  ...............000000000000.....
0  ..............000000000000......
0  .............000000000000.......
1  ............100100010001........
0  ...........000000000000.........
0  ..........000000000000..........
1  .........100100010001...........
-----------------------------------
   000000000101001000110011001000001   = 5387041
```

Let’s do the calculation in binary

We all learned that multiplies are just stacks of shifted additions

1. Rows only appear when the multiplier has a 1-bit otherwise they are zero
2. We sum columns - with carry - into the result

Immediately you can see that a multiply is faster if you sort the value by number of 1-bits

Let’s check out this symmetry idea

Squaring in binary

```
1  ..................100100010001
0  .................00000000000.
0  ................0000000000...
0  ...............000000000.....
1  ..............10010001......
0  .............0000000.......
0  ............000000........
0  ...........00000.........
1  ..........1001..........
0  .........000............
0  ........00.............
1  .......1...............
-----------------------------------
```

Removing the duplicate bits below the diagonal line leaves us with a triangle of bits

Squaring in binary

```
1                    100100010001
0                   00000000000
0                  0000000000
0                 000000000
1                10010001
0               0000000
0              000000
0             00000
1            1001
0           000
0          00
1         1
---------------------------------
```

Squaring in binary

```
1 ..................100100010001
0 .................00000000000.
0 ................0000000000....
0 ...............000000000......
1 ..............10010001........
0 .............0000000..........
0 ............000000............
0 ...........00000..............
1 ..........1001................
0 .........000..................
0 ........00....................
1 .......1......................
-----------------------------------
```

So the algorithm says we keep the digits on the diagonal and double the remaining bits

Squaring in binary

```
1  ..................10010001000.1
0  .................0000000000.0..
0  ................000000000.0....
0  ...............00000000.0......
1  ..............1001000.1........
0  .............000000.0..........
0  ............00000.0............
0  ...........0000.0..............
1  ..........100.1................
0  .........00.0..................
0  ........0.0....................
1  .......1.......................
---------------------------------
```

Well, times two, that’s just a shift left

Squaring in binary

```
1  ..................1001000100001
0  .................000000000000.
0  ................00000000000....
0  ...............0000000000......
1  ..............100100001........
0  .............00000000..........
0  ............0000000.............
0  ...........000000................
1  ..........10001..................
0  ..........0000...................
0  .........000....................
1  ........01......................
-----------------------------------
```


And we fill the spaces with zeros

Squaring in binary

```
1  ..................1001000100001
0  .................000000000000..
0  ................00000000000....
0  ...............0000000000......
1  ..............100100001........
0  .............00000000..........
0  ............0000000............
0  ...........000000..............
1  ..........10001................
0  .........0000..................
0  ........000....................
1  .......01......................
-----------------------------------
   000000000101001000110011001000001  = 5387041
```

You can check that it gives the same result.

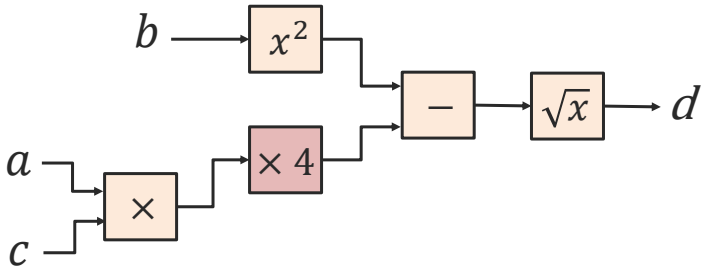
A more interesting test is to try the result with 2-s compliment signed values to show that squaring always produces a positive result - plus an overflow flag

We have seen that

- multiplication can be viewed as repeated addition
- looking at the bits can lead to smaller, faster results

Thinking in gates

- In fixed point multiplication by four is just wiring, not even a gate
- In floating point × 4 is simply adding 2 to the exponent



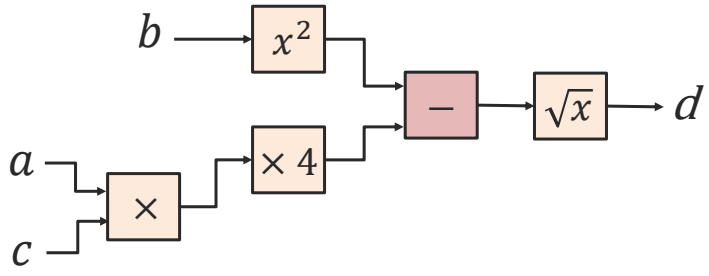
So with the new information

We can see that "times 4" is just a simple shift

- In fixed point it's not even a gate, it's just wiring
- In float it's a "plus-2" to the exponent

Thinking in gates

- If both operands of an ADD/SUB are the same sign, we can make the op smaller
  - The cancellation case never happens, saving 1 large leading-zero-counter + 1 shifter
  - For example $\sqrt{x^2 + y^2}$ guarantees this



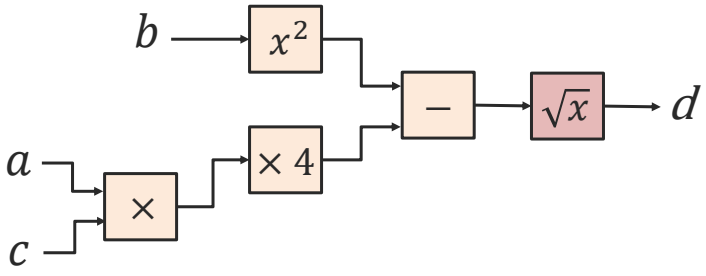
Even simple additions can be optimized if we can guarantee constraints on the operands

If they have the same sign we can remove the circuitry that supports the cancellation case

Not useful here, but comes into play for vector length calculations

Thinking in gates

- If the range of the input is small, we can approximate with a low degree poly
- New fixed point methods for SQRT we will look at later



If we know that the input range is low, we can approximate with a table or low degree polynomial

TAKEAWAY: Arithmetic in the FPGA world is very different

Some of these ideas can directly convert to fixed-point techniques on CPU, which we will be looking at

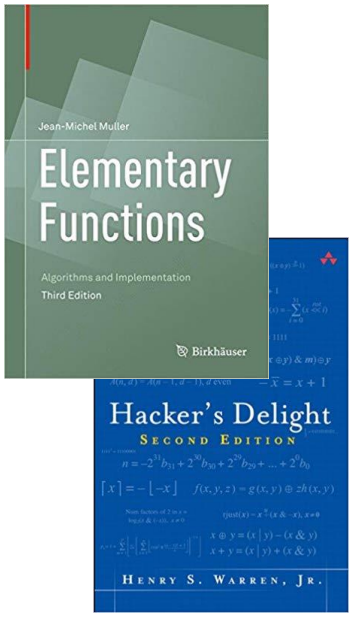
1. Function Approximation
2. Range Reduction
3. Function Approximation
4. Recip, Sqrt, InvSqrt
5. Bitheaps and FPGA
6. Lookup Tables
7. Periodic Evaluation
8. Summary

Here’s the running order of the talk

I went overboard but there are so many interesting things

Resources

- Cephes libm
  https://github.com/jeremybarnes/cephes/blob/master/single/tanf.c
- Sun MUSL libm
  https://github.com/runtimejs/musl-libc/blob/master/src/math/atanf.c
- Apple libm
  https://opensource.apple.com/source/Libm/Libm-315/Source/ARM/tanf.h.auto.html
- Intel Math Kernel Library (MKL)
  https://software.intel.com/en-us/mkl-developer-reference-c-performance-enhancements



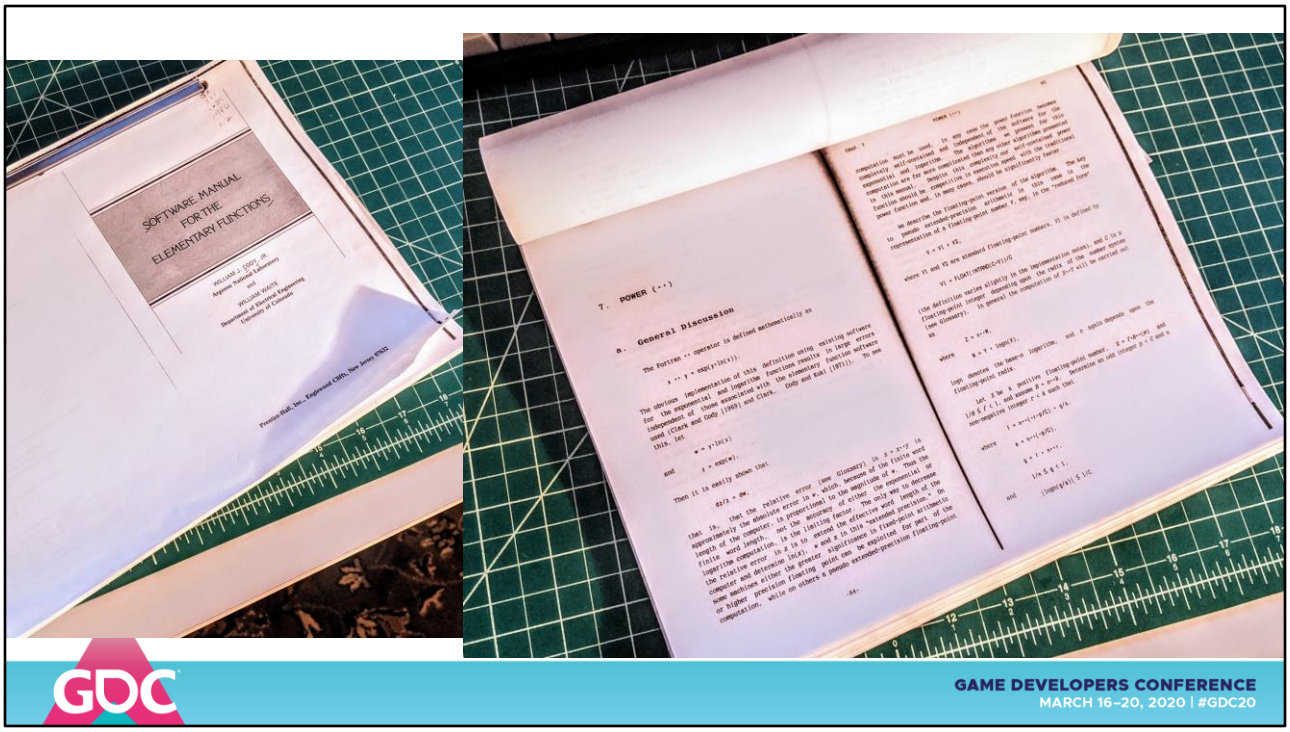
- Jean-Michel Muller "*Elementary Functions, Algorithms and Implementation, 3rd Ed*", Birkhauser, 2016
- Henry S Warren Jr, "*Hacker's Delight, 2nd Ed*", Addison Wesley, 2013
- W. Cody & W. Waite "*Software Manual for the Elementary Functions*", Prentice-Hall, 1980

In researching this, I finally bought "*Elementary Functions*" and it contained overviews of almost everything I had been researching.

So don't be me and buy a copy first.

Also, the "*Hacker's Delight*" is a timeless, invaluable exploration of bit twiddling and hackery

- All options are explored and caveats are explained
- It's a masterpiece of bit-level techniques



Often you'll see references to "*Cody & Waite*" – an out of print, hand typeset book from the 1980s.

Almost everyone I know who has a copy is using a pile of photocopied pages.

The advice inside is starting to look a little dated, but it's a fascinating historical artifact of the pre-IEEE754 mainframe days

Function Approximation

- There's a pretty standard model of function approximation
  1. **First Range Reduction**

     e.g. $\sin(x) = \sin\left(y + k\frac{\pi}{2}\right)$ or $\exp(x) = \exp\left(y + k\ln\left(\frac{\pi}{2}\right)\right)$
  2. **Second Range Reduction**

     e.g. a table lookup like $C = -\cos\left(\frac{\pi}{2}\right)$ if $k \bmod 4 = 2$
  3. **Approximation**

     e.g. a polynomial like $\sin(z) = z - 0.166665739\,z^3 + 0.008298676\,z^5$
  4. **Reconstruction**

     e.g. $\sin(x) = \sin(z)C - \cos(z)S$

We can EVALUATE a square root, but we can only APPROXIMATE arcsine

Quick recap of the previous "*Faster Math Functions*" talk

Approximation for trig, exponential and logarithm functions take a pretty standard form:

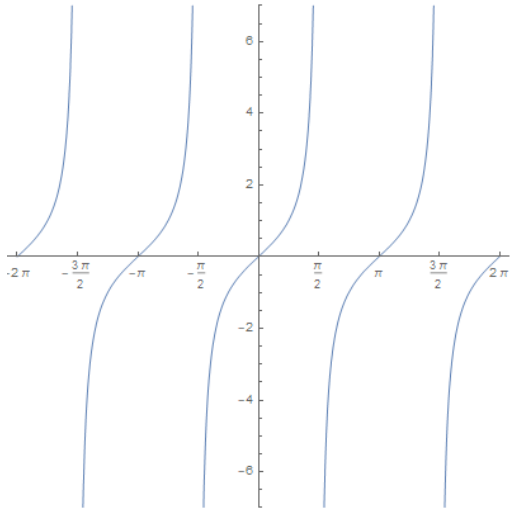
- **First range reduction** – reduces arbitrary arguments to something periodic like 0..2PI or 1..2

- **Second range Reduction** – often reduces the range to incorporate precomputed elements from a table

- **Approximation** – fill in the tiny remaining range with an optimized polynomial

- **Reconstruction** – assembles the approximated and precomputed pieces along with sign and quadrant values

Approximating `tanf()`



All the beginner texts start with Sine and Cosine

Both well behaved functions, let’s try something different – TAN

Approximating `tanf()`

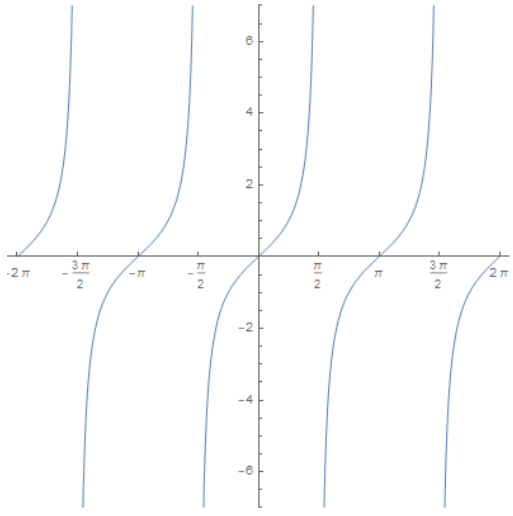
- This is not how it's done.

```
169
170  static inline float
171  fastertan (float x)
172  {
173    return fastersin (x) / fastercos (x);
174  }
175
176  static inline float
```


This piece of code is not ... optimal

But at least it was inline for extra speed

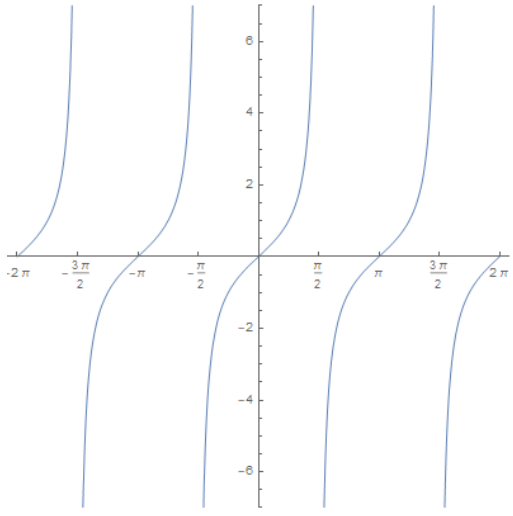
Approximating `tanf()`



Lets look at some of the identities we can exploit to reduce the amount (range) of function we need to approximate

Approximating `tanf()`

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

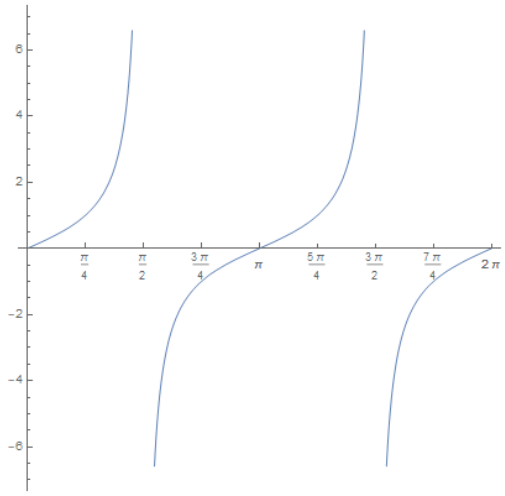


All the negative values can be constructed from the positive side

We can use an ABS to remove half of the number line and reconstruct it later

Approximating `tanf()`

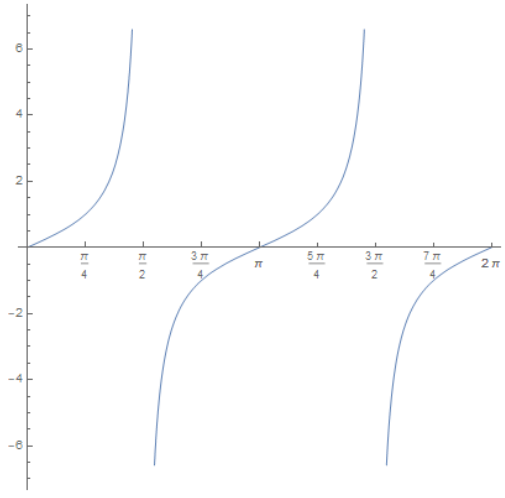
- $\tan(-x) = -\tan(x)$



Saved 50% of the number line, BOOM

Approximating `tanf()`

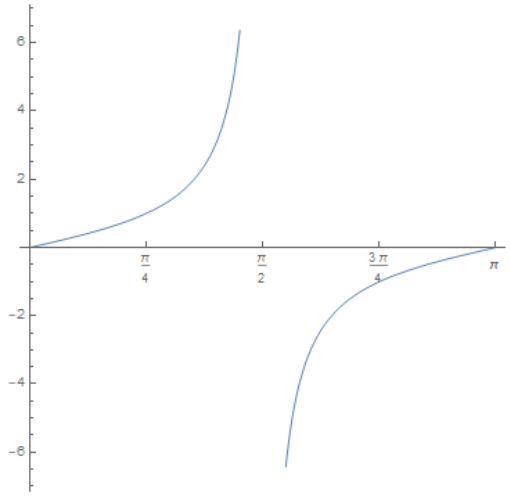
- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$



Next the pattern repeats every PI units

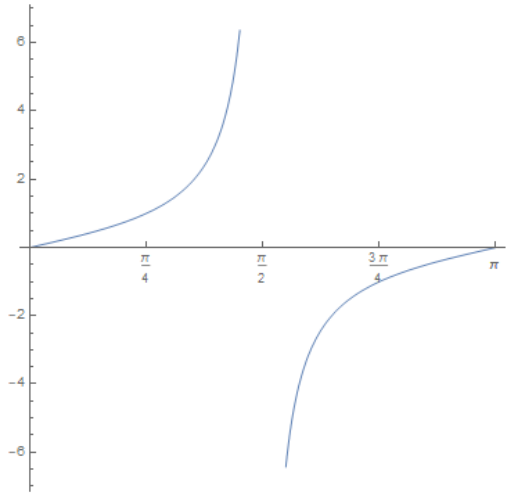
Approximating `tanf()`

- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$



Approximating `tanf()`

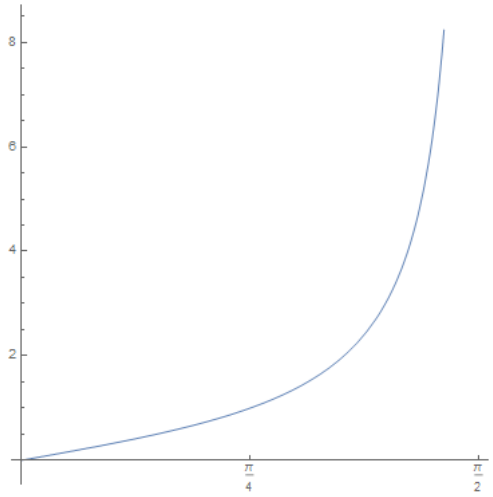
- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$
- A simple reverse and negate at $\pi/2$



The range at PI/2 is a reverse and negation of the first PI/2

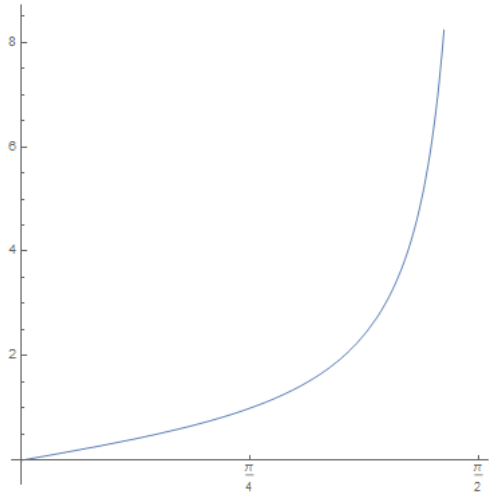
Approximating `tanf()`

- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$
- A simple reverse and negate at $\pi/2$



Approximating `tanf()`

- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$
- A simple reverse and negate at $\pi/2$
- The range $[\pi/4\,, \pi/2]$ is $\frac{1}{\tan(\pi/4 - x)}$

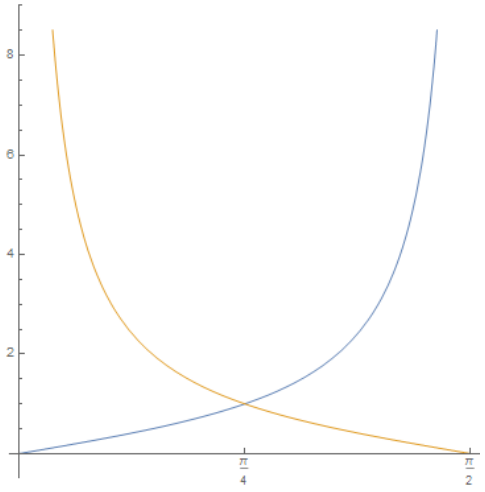


Math identities tell us that the range [PI/4, PI/2]

is the reciprocal of [0, PI/4]

Approximating `tanf()`

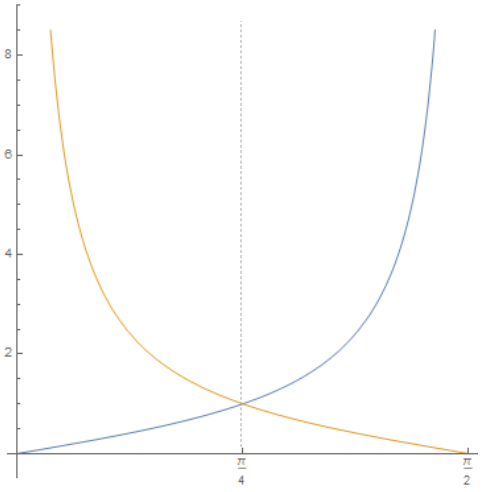
- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$
- A simple reverse and negate at $\pi/2$
- The function $\left[\frac{\pi}{4}, \frac{\pi}{2}\right]$ is $\frac{1}{\tan\left(\frac{\pi}{4} - x\right)}$



This is about the only time math books mention cotangent COT

Approximating tanf()

- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$
- A simple reverse and negate at $\pi/2$
- The function $\left[\frac{\pi}{4}, \frac{\pi}{2}\right]$ is $\frac{1}{\tan\left(\frac{\pi}{4} - x\right)}$
- If we have a reciprocal,
  we only need to approximate $\left[0, \frac{\pi}{4}\right]$

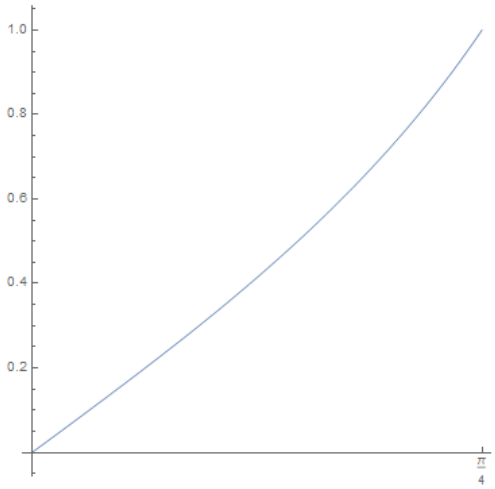


So we only need to approximate the first [0, PI/4] of the function

We can reconstruct the rest from just that

Approximating `tanf()`

- $\tan(-x) = -\tan(x)$
- We have a repeating pattern over $[0, \pi]$
- A simple reverse and negate at $\pi/2$
- The function $\left[\frac{\pi}{4}, \frac{\pi}{2}\right]$ is $\frac{1}{\tan\left(\frac{\pi}{4} - x\right)}$
- If we have a reciprocal,
  we only need to approximate $\left[0, \frac{\pi}{4}\right]$



It's also an uncomplicated function that we can fit a polynomial to

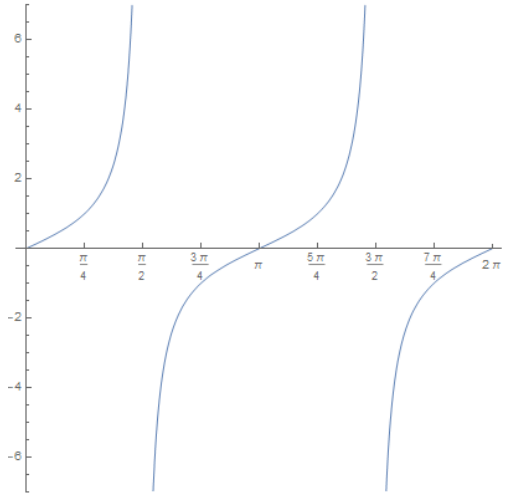
We can go further using precomputed tables, but this is deep enough for this simple example

Let's apply the range-reduce-approximate-reconstruct model we talked about

First Range Reduction

- Copy the sign and set $x = |x|$
- Break the parameter x into repeating chunks of $\left[0, \frac{\pi}{4}\right]$

$$k = \left\lfloor x \frac{4}{\pi} \right\rfloor$$
$$y = x - k\frac{\pi}{4}$$



This is the Additive Range Reduction method

K is the INTEGER FLOOR of X-times-4-over-PI

which tells us how many copies of Pi/4 are inside X

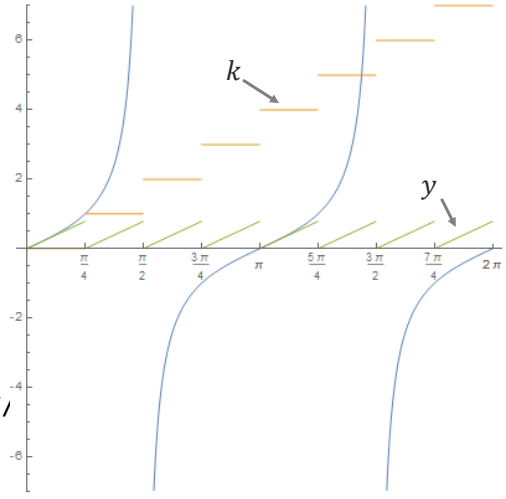
We then subtract that many copies from X to get a range reduced value Y

First Range Reduction

- Copy the sign and set $x = |x|$
- Break the parameter x into repeating chunks of $\left[0, \frac{\pi}{4}\right]$

$$k = \left\lfloor x \frac{4}{\pi} \right\rfloor$$
$$y = x - k\frac{\pi}{4}$$

- Subtract k copies of $\pi/4$ from x until $x < \pi/$
- k is an integer, handy for quadrant tests



Plotting Y and K shows us the driving functions we can use to construct our approximation

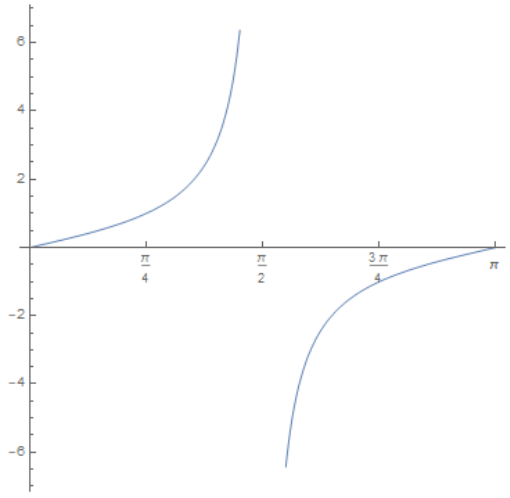
K is an integer that can be used to tell us which quadrant we are in

Y is a parameter we can send to our polynomial for [0, Pi/4]

Second Range Reduction

- Some functions use tables, here we use bithacks and symmetry

$$k \bmod 4 = \{0,1,2,3\}$$



Second Range Reduction

Since we wimped out of table-based range reduction here (which would go here)

We will exploit symmetry

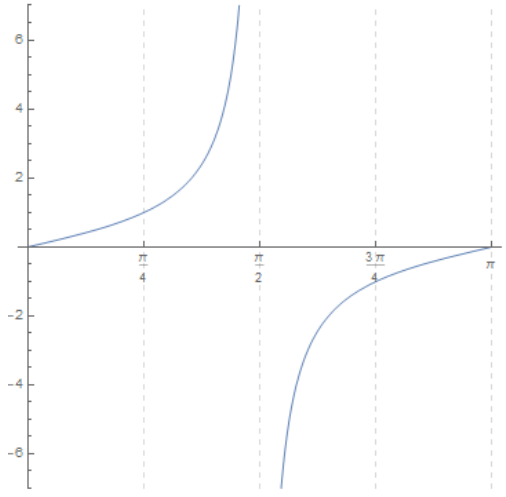
Using the K variable we can codify the repeating pattern

Second Range Reduction

- Some functions use tables, here we use bithacks and symmetry

$$k \bmod 4 = \{0,1,2,3\}$$

$$z = \begin{cases} y & k \bmod 4 = 0 \\ (\pi/4) - y & k \bmod 4 = 1 \\ -y & k \bmod 4 = 2 \\ y - (\pi/4) & k \bmod 4 = 3 \end{cases}$$



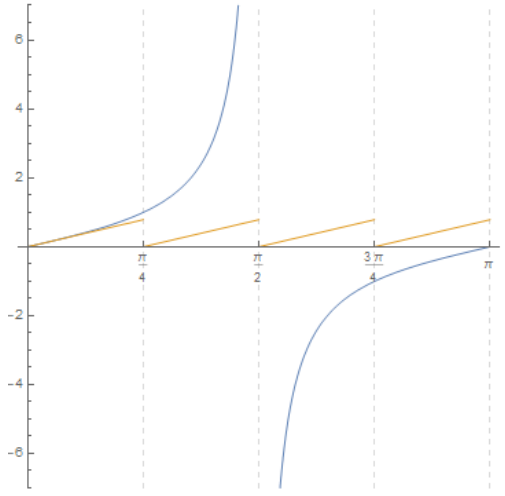
K MOD 4 is the same as a bitwise K AND 3

Second Range Reduction

- Some functions use tables, here we use bithacks and symmetry

$$k \bmod 4 = \{0,1,2,3\}$$

$$z = \begin{cases} y & k \bmod 4 = 0 \\ (\pi/4) - y & k \bmod 4 = 1 \\ -y & k \bmod 4 = 2 \\ y - (\pi/4) & k \bmod 4 = 3 \end{cases}$$



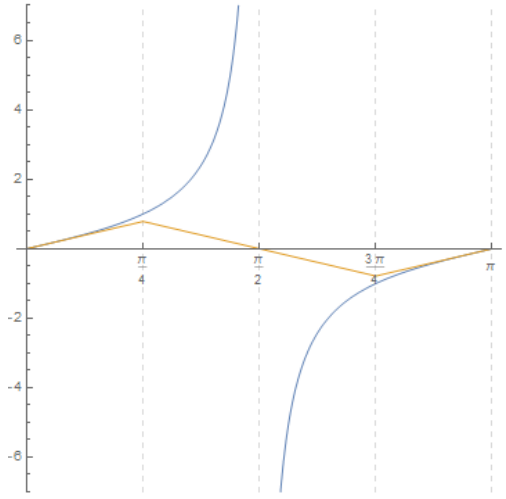
Each section is reversed or negated separately...

Second Range Reduction

- Some functions use tables, here we use bithacks and symmetry

$$k \bmod 4 = \{0,1,2,3\}$$

$$z = \begin{cases} y & k \bmod 4 = 0 \\ (\pi/4) - y & k \bmod 4 = 1 \\ -y & k \bmod 4 = 2 \\ y - (\pi/4) & k \bmod 4 = 3 \end{cases}$$

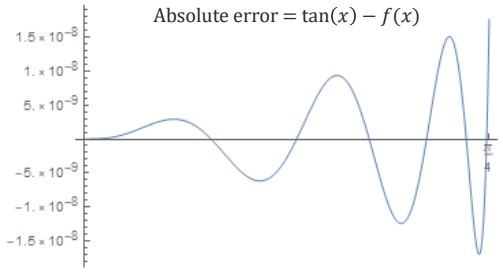


...which reverses and reflects the driving functions for the polynomial

Approximation

- Using the driver variable z we can approximate $\tan(x)$ over $\left[0, \frac{\pi}{4}\right]$ using a minimax polynomial

$$\begin{aligned} result \; = z &+ 0.333331568548\, z^3 + \\ &0.133387994085\, z^5 + \\ &0.0534112807\, z^7 + \\ &0.0244301354525\, z^9 + \\ &0.00311992232697\, z^{11} + \\ &0.00938540185543\, z^{13} \end{aligned}$$



Here is an approximation to tan(x) over [0, Pi/4] using a minimax polynomial.

We'll be looking closer at those later.

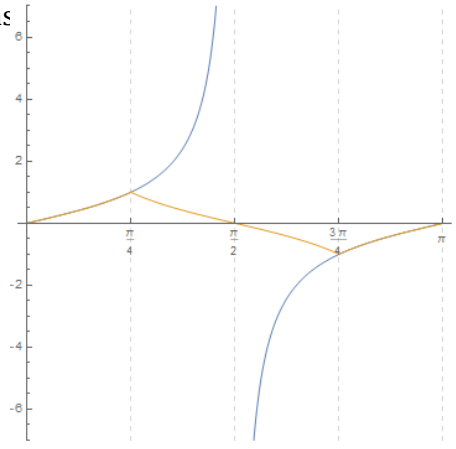
Polynomials can be evaluated way more efficiently than this raw expression.

This poly has ~10e-8 accuracy which is OK for a approximation to binary32 float accuracy (24-bit mantissa)

Not good enough to guarantee final bit correctness, but this is a basic demo. We can get into the weeds later

Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$

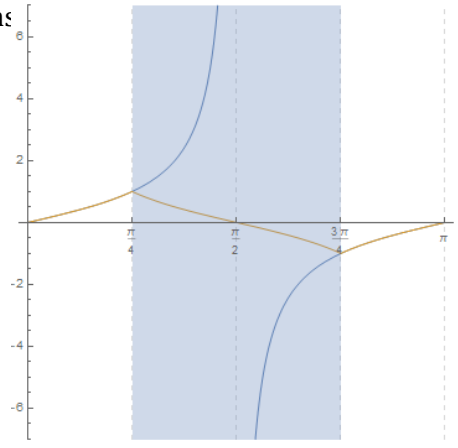


Here is the result of driving the Y function through the polynomial

We still need to apply the reciprocals

Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$

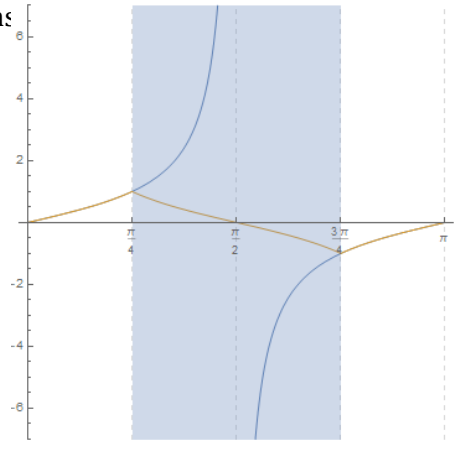


These two sections are where we reciprocate

Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$
- We can use a bit test on k

$$if\ (k+1) \bmod 2 = 2$$
$$then\ result = 1/result$$



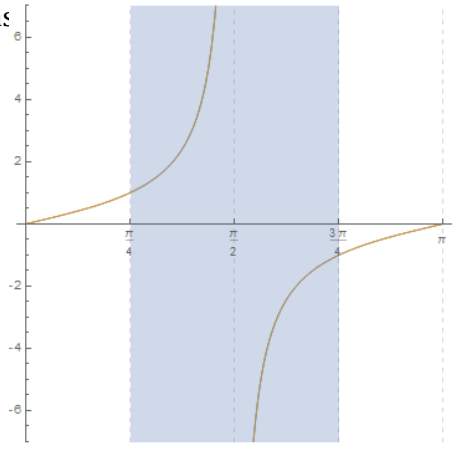
We can pick them out using a bithack on the K variable

Approximating `tanf()`

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$

- We can use a bit test on k

$$if\ (k+1) \bmod 2 = 2$$
$$then\ result = 1/result$$

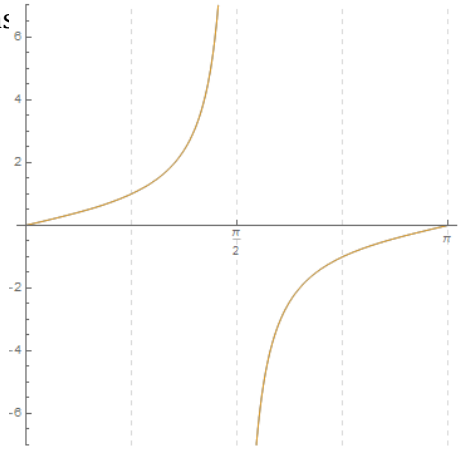


After applying the reciprocal...

Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$
- We can use a bit test on k

$$if\ (k+1) \bmod 2 = 2$$
$$then\ result = 1/result$$

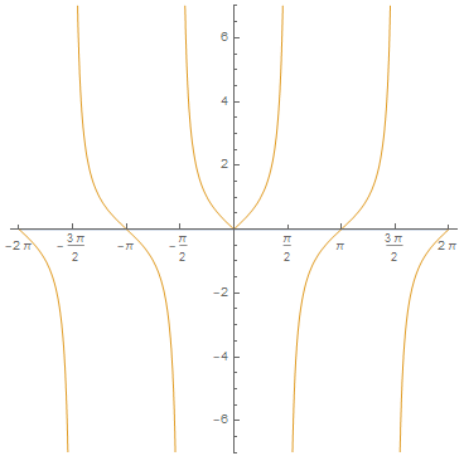


We have one cycle of TAN correctly approximated

Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$
- We can use a bit test on k

$$if\ (k+1) \bmod 2 = 2$$
$$then\ result = 1/result$$



However, negative numbers are still messed up

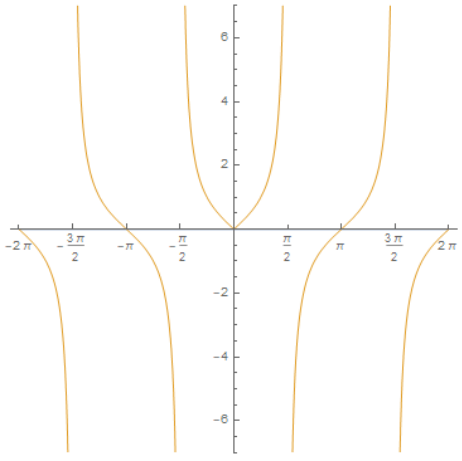
Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$
- We can use a bit test on k

$$if\ (k+1) \bmod 2 = 2$$
$$then\ result = 1/result$$

- And finally, restore the sign

$$result = sign \times result$$



If we restore the sign bit we copied off earlier...

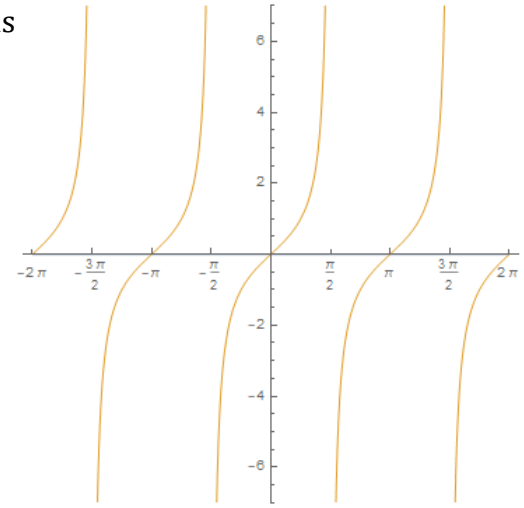
Reconstruction

- We need to apply reciprocals in the sections where $k \bmod 4 = \{1, 2\}$
- We can use a bit test on k

$$if\ (k+1) \bmod 2 = 2$$
$$then\ \ result = 1/result$$

- And finally, restore the sign

$$result = sign \times result$$

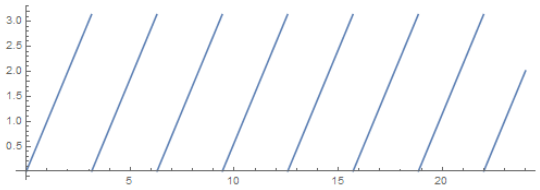


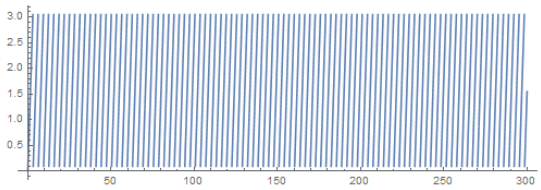
We get the function correct across the whole numberline

Additive Range Reduction

- Taking an input x and reducing it to a smaller range, e.g. $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ by subtracting k copies of the whole range C

$$R = x - kC \qquad \text{where } k = \left\lfloor x \frac{1}{C} \right\rfloor$$





- Note the cancellation problem when x is close to kC, e.g. $x = 22$

In the TANF example we just used additive range reduction

There is also multiplicative range reductions used for exp() and log() functions

If you plot what range reduction does, it generates extremely detailed features from values on the number line

Under floating point, this detail is going to break down the further you get from zero

Cody & Waite Additive Range Reduction

- To get additional accuracy, we can break C into multiple values

$$C = C_1 + C_2$$

- The first value should be exactly machine representable, e.g.

$$C_1 = \frac{201}{64} = 3.140625$$
$$C_2 = \pi - C_1 = 9.67653589793 \times 10^{-4}$$

- The reduction calculation is then

$$R = (x - k \times C_1) - k \times C_2$$

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.9012&rep=rep1&type=pdf

For additional accuracy, we can split the constants into two floating point values and subtract them from largest to smallest

Called Cody & Waite reduction

Two stage range reduction will take you up to 2^64 = 1.8*10^18

For values larger than 2^64 there are large number techniques

(e.g. binary64 and binary128 libraries will require these)

Double Residue Modular Range Reduction

- Using an analogy, let's say we need 372 mod 7
- We can break 372 into bits

$$372 = 101110100_2$$
$$= 256 + 64 + 32 + 16 + 4$$

- Make a table of $i \bmod M = 7$ as a *signed residuals*

$$m_i^+ = 2^i \bmod M$$

$$m_i^- = 2^i \bmod M - M$$

0	1	-6	1
1	2	-5	2
2	4	-3	4
3	8	-6	1
4	16	-5	2
5	32	-3	4
6	64	-6	1
7	128	-5	2
8	256	-3	4
9	512	-6	1
10	1024	-5	2

There is a more interesting idea "Double Residue Range Reduction"

We can inspect the parameter and break it into bits

We can tabulate what each power-of-2 is mod 2PI

These are the residuals

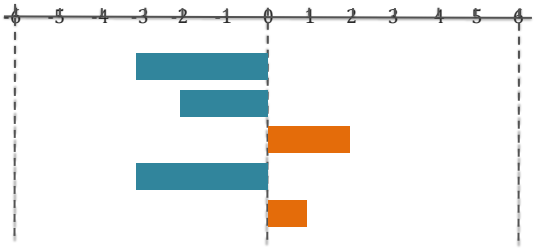
Double Residue Modular Range Reduction

- Starting with $R = 0$ we visit each non-zero bit in the table summing

$$R_{i+1} = R_i + m_i^* \qquad \text{where } m_i^* = \begin{cases} m_i^+ & \text{if } R_i < 0 \\ m_i^- & \text{if } R_i \geq 0 \end{cases}$$

- For the example of $372 = 101110100_2 \bmod 7 = 1$

$$\begin{array}{lll} bit\ 8 = 1 & R = 0 & m_7^- = -3 \\ bit\ 6 = 1 & R = -3 & m_6^+ = 1 \\ bit\ 5 = 1 & R = -2 & m_5^+ = 4 \\ bit\ 4 = 1 & R = 2 & m_4^- = -5 \\ bit\ 2 = 1 & R = -3 & m_2^+ = 4 \\ & R = 1 & \end{array}$$



Initialize R = 0

Visit each non-zero bit in the parameter

Look up the residual in the table (based on sign of R)

Sum into R

Following this algorithm, the value of R is guaranteed to stay within the modular bounds, in this case +/- 2PI

Double Residue Modular Range Reduction

- We can tabulate $x \bmod \pi$ in the same way producing fractional entries
- No matter the size of x, the value of R is guaranteed to stay within $\pm M$

0	1	-2.14159265359	1.00000000
1	2	-1.14159265359	2.00000000
2	4	-2.28318530718	0.85840734641
3	8	-1.42477796077	1.71681469282
4	16	-2.84955592154	0.292036732051
5	32	-2.55751918949	0.584073464102
6	64	-1.97344572539	1.1681469282
7	128	-0.805298797182	2.33629385641
8	256	-1.61059759436	1.53099505923
9	512	-0.0796025351362	3.06199011845
10	1024	-0.159205070272	2.98238758332

- One additional iteration used to correct the sign of the final result

http://rnc7.loria.fr/villalba.pdf

We can do the same calculation with floats

Summing the FP residuals into R

One additional iteration to ensure the correct sign.

Wait, what?

- You transformed a single multiply-subtract into a loop with table lookups?!

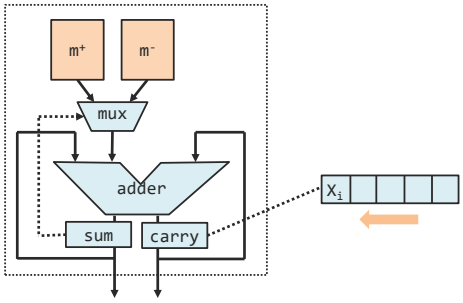
Wait.

You just turned a simple multiply-subtract into a loop?

With table lookups?

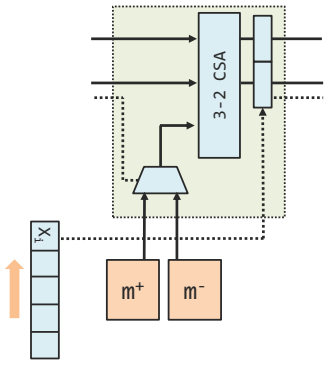
Wait, what?

- You transformed a single multiply-subtract into a loop with table lookups?!
- Remember, in FPGA land multiplies are repeated additions.



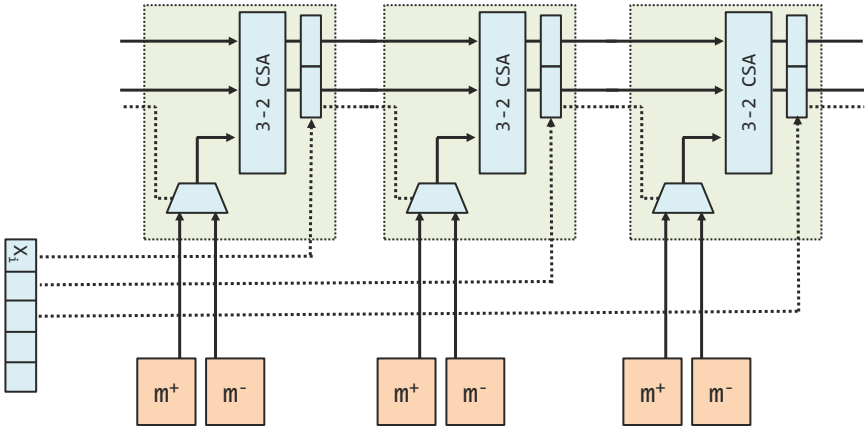
FPGA land doesn’t think of loops as a problem

Wait, what?



We can take the basic idea and unroll the loop...

Loop unrolling

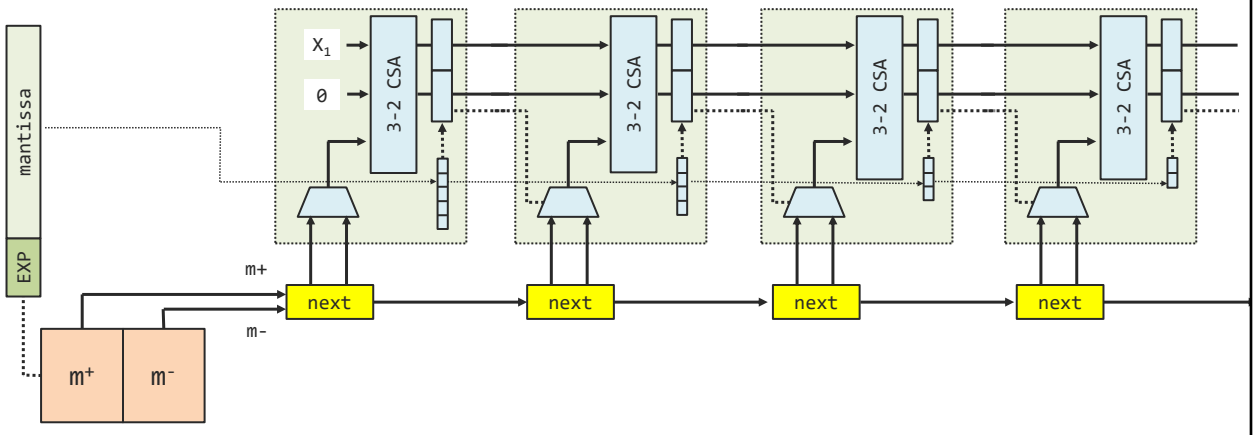


3-2 CSA – Carry Save Adder that sums three inputs into two

But now we've have too many copies of the tables

The paper shows how to calculate the next table value from the previous one ...

Loop unrolling



So we only need one table

This unrolled model gives us last-bit range reduction in about the same tree depth as a large multiply-add (but more area)

TAKEAWAY: Things that look dumb and expensive in CPU land may turn out way cheaper in FPGA land.

Payne & Hanek Reduction

- To reduce extremely large angles $> 2^{64} = 1.8 \times 10^{18}$ into $[0, 2\pi]$
- Think of a floating point number as a bit string
- The question becomes, how many bits do we need more?

CUT FOR TIME

https://www.csee.umbc.edu/~phatak/645/supl/Ng-ArgReduction.pdf

These techniques work for angles less than 2^64 = 1.8*10^18

There is a cutoff point where additive range reduction stops working

To get past these points, there is Payne & Hanek reduction

Uses a precalculated table with hundreds of bits of PI but only uses a small number of them

The proof is worth reading for it's bit-centric thinking

Taylor series suck

- I want you to be deeply suspicious of any tutorial using Taylor series
  - Taylor series are approximations around a point, not a range

- Chebyshev described the optimal polynomial approximation

> The error of an order N polynomial approximation will:
>
> 1. Cross the $x = 0$ line $N + 1$ times
> 2. Approach the $\pm$ maximum error $N + 2$ times

Be suspicious of any ADVANCED tutorial using Taylor series

They suck for numerical accuracy for all but the smallest delta

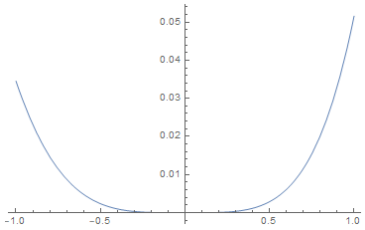
Instead, demand accuracy across the whole range

MINIMIZE the MAXIMUM error

Taylor series suck

- e.g. Approximating e^x where $x \in [-1,1]$ with an order 3 Taylor series:

$$t(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$



Approximating exp with an order-3 Taylor series

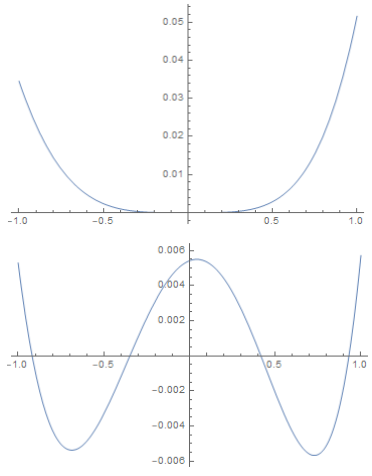
Shows that it's accurate at a point, not at across a range

Exponentially inaccurate any distance from the point.

Taylor series suck

- e.g. Approximating e^x where $x \in [-1,1]$ with an order 3 Taylor series:

$$t(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$



- The same with a order 3 minimax polynomial:

$$m(x) = 0.994572447541 + 0.996302027975\,x + 0.542986901612\,x^2 + 0.178688719292\,x^3$$

Using exactly the same number of terms, we can get 10x improvement in accuracy over the range

The optimal order-N polynomial approximation:

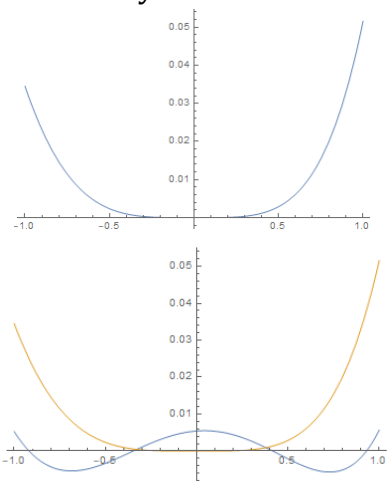
Crosses the x=0 axis 4 times (N+1)

Approaches max error 5 times (N+2)

Taylor series suck

- e.g. Approximating e^x where $x \in [-1,1]$ with an order 3 Taylor series:

$$t(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$



- The same with a order 3 minimax polynomial:

$$m(x) = 0.994572447541 + 0.996302027975\,x + 0.542986901612\,x^2 + 0.178688719292\,x^3$$

Here is the abs error plot on the same scale

Minimax Polynomials

- How to calculate the magic polynomial coefficients?

- **Mathematica** includes an approximations package not included by default
  - Add `Needs["FunctionApproximations`"]` to your workbook

- **Maple** has the **`numapprox`** package that provides
  - Add `using(numapprox);` to import the whole library

- **Sollya** contains an interactive environment for numerical approximation
  - Build and install under Linux or WSL for Windows 10 Pro, requires additional libraries
  - Start on the bash command line using `rlwrap -A sollya` for an interactive UI

https://sollya.gforge.inria.fr

To calculate accurate polynomial, you need special software designed to go beyond machine accuracy

Mathematica, Maple and the open source Sollya all have the necessary code for generating coefficients to arbitrary accuracy

So you want to make a minimax poly

- You get the package, you dive right in and make an approximation

```
p = RationalInterpolation[Sin[x], {x,5,0}, {x,0,2Pi}]
```

Let me give you a taste of the process

This really is an art more than a mechanical process
And it can be frustrating as hell

You want a 5^{th} order polynomial approximation of sine

We use the RationalInterpolation function to optimize absolute error
(Minimax optimizes for relative error)

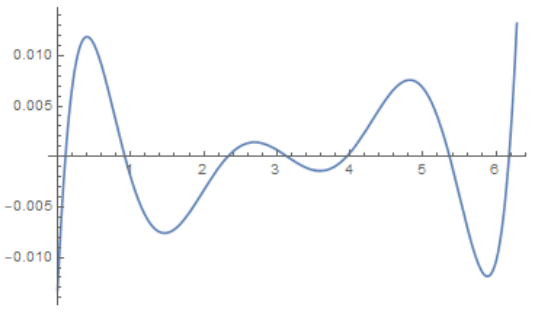
You press the button and ...

The results are... not great

- You get the package, you dive right in and make an approximation

```
p = RationalInterpolation[Sin[x], {x,5,0}, {x,0,2Pi}]
```

```
 0.013192765375372996047607 00 +
 0.845809402596108346252474 4 x +
 0.314734001696293465470935 3 x^2
-0.415675630350305455624733 9 x^3 +
 0.091262941543546380195816 89 x^4
-0.005809979307104837925728 709 x^5
```



- And it's not very good

... it's not very good

1% accuracy over the range

Where are the magical numbers this is supposed to produce?

Reduce the range

- So we reduce the range to $\pm\pi/2$ and try again

```
p = RationalInterpolation[Sin[x], {x,5,0}, {x,-Pi/2,Pi/2}]
```

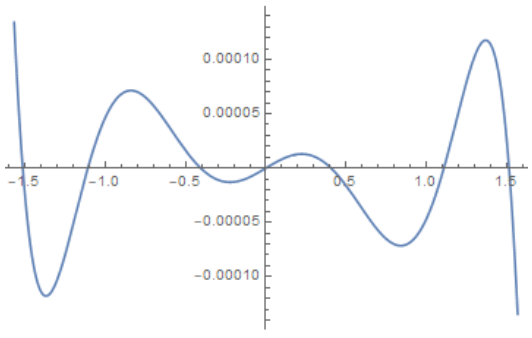
Let’s try reducing the range to a quarter of a cycle

Weird tiny coefficients

- So we reduce the range to $\pm\pi/2$ and try again

```
p = RationalInterpolation[Sin[x], {x,5,0}, {x,-Pi/2,Pi/2}]
```

$$-1.6168575796572830719119 \times 10^{-43} + 0.99991152837960383597075 54\, x + 2.0127791340503065826147 05 \times 10^{-43}\, x^2 - 0.16602000425894697326719 30\, x^3 - 5.6911328646816887056226 63 \times 10^{-44}\, x^4 + 0.0076266621511777583121647 98\, x^5$$



- Better, abs error = 0.0001342

Meh, better

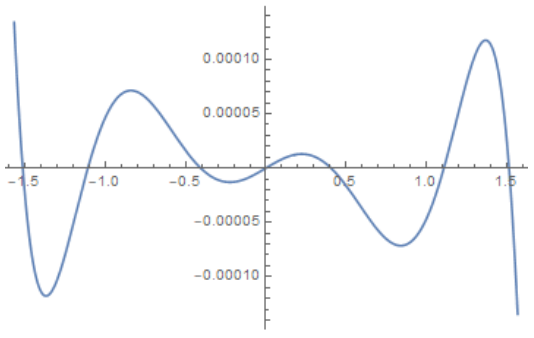
Still not magical – we can match that accuracy with a LERP table

Weird tiny coefficients

- So we reduce the range to $\pm\pi/2$ and try again

```
p = RationalInterpolation[Sin[x], {x,5,0}, {x,-Pi/2,Pi/2}]
```

$$-1.6168575796572830719119 \times 10^{-43} + 0.99991152837960383597075 54\, x + 2.0127791340503065826147 05 \times 10^{-43}\, x^2 - 0.16602000425894697326719 30\, x^3 - 5.6911328646816887056226 63 \times 10^{-44}\, x^4 + 0.00762666215117775831216 4798\, x^5$$



- Better, abs error = 0.0001342
- But we have weird tiny coefficients

… plus now we have these weird tiny constants

Every second power

- Looking at the Taylor series for $\sin(x)$ it has odd powers

$$\sin x \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} + \cdots$$

- So we need a minimax polynomial of the form

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

- Solving for P with a single variable gives us

$$P(y) = -\frac{1}{y} + \frac{\sin(\sqrt{y})}{y^{3/2}} \qquad \text{where } y = \sqrt{x}$$

Check the Taylor series to get some insight

Sine starts with a 1.0 * X and uses every other power

So we need to create a polynomial of the form P(X^2)

Substitute Y = SQRT(X) and solve for P(Y) we get this weird rational

Every second power

- We code a solution with a post reconstruction

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = RationalInterpolation[d, {y,1,0}, {y, -(Pi/2)^2, (Pi/2)^2 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

We plug this into the solver ...

Every second power

- We code a solution with a post reconstruction

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = RationalInterpolation[d, {y,1,0}, {y, -(Pi/2)^2, (Pi/2)^2 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

- NOTE: Sollya is purpose designed for this kind of approximation
  - Allows you to control the powers explicitly in the minimax parameters

```
> remez(sin(x), [|0,2,4|], [|-pi/2, pi/2|], absolute);
```

NOTE: Sollya has specific controls for generating arbitrary powers

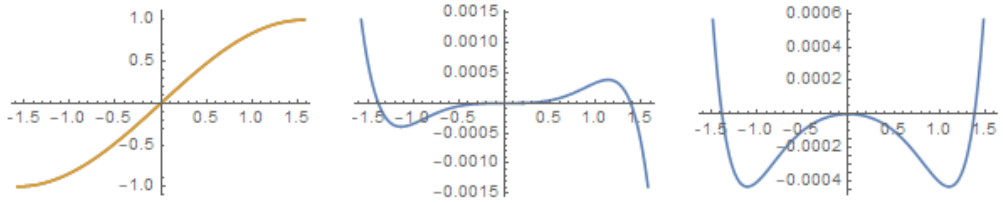
The tools in Mathematica and Maple are starting to look old

Looks good, performs bad

- We code a solution with a post reconstruction

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = RationalInterpolation[d, {y,1,0}, {y, -(Pi/2)^2, (Pi/2)^2 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

```
x - 0.16701898465403695152 x^3 + 0.008148259912957998312 x^5
```



Polynomial looks good!

Performance looks bad

A tenth of the accuracy of the every-power version.

Reduce the range again

- Let's try a smaller range, this time $\pi/8$

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = RationalInterpolation[d, {y,1,0}, {y, -(Pi/8)^2, (Pi/8)^2 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

OK, deep breath, we reduce the range further

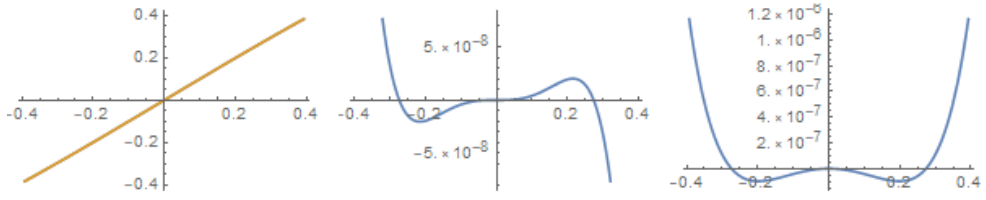
We can make up the difference with a larger table in Second Range Reduction

Reduce the range again

- Let's try a smaller range, this time $\pm\pi/8$

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = RationalInterpolation[d, {y,1,0}, {y, -(Pi/8)^2, (Pi/8)^2 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

```
x - 0.16667131080141501286 x^3 + 0.008380604102404438736O x^5
```



Looking better, getting somewhere

Abs Error around 0.6 X 10^-8

Relative error looking pretty biased positive

Reduce the range some more!

- Let's try the half range $[0, \pi/8]$

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = MiniMaxApproximation[d, {y, { 0, (Pi/8)^2 }, 1,0 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

Switching to Minimax Approximation to optimize for relative error

Let's only approximate the positive half of the function

Reduce the range some more!

- Let's try the half range $[0, \pi/8]$

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = MiniMaxApproximation[d, {y, { 0, (Pi/8)^2 }, 1,0 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

Power: Infinite expression $\frac{1}{0}$ encountered.

- Minimax/Remez can't handle zeros in the range

KABOOM

Minimax can't handle zeros in the range

We could manipulate the polynomial to remove zero crossings and add them back at reconstruction, or...

Skip the zero crossing

- Starting at 1e-10 instead of zero

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = MiniMaxApproximation[d, {y, { 1*^-10, (Pi/8)^2 }, 1,0 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

... offset the range a tiny fraction past the zero

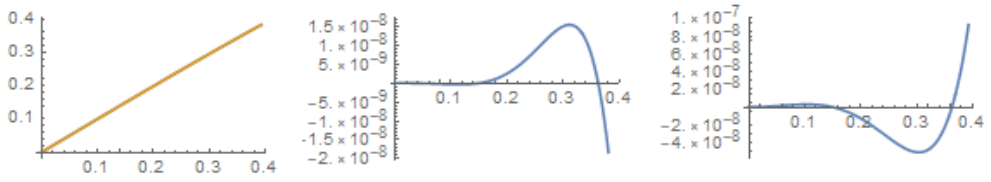
(Stupid hacks)

Skip the zero crossing

- Starting at 1e-10 instead of zero

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = MiniMaxApproximation[d, {y, { 1*^-10, (Pi/8)^2 }, 1,0 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

```
x -0.16666607646888027215 x^3 + 0.008302771643344929 9798 x^5
```



Looking good!

Well formed polynomial, relative and absolute errors about the same

Moar accuracy!

- How about adding another power

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = MiniMaxApproximation[d, {y, { 1*^-10, (Pi/8)^2 }, 2,0 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

Let’s crank up the powers

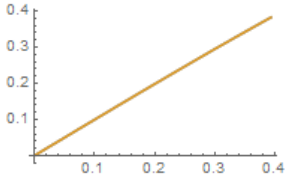
A 7^{th} order polynomial (after reconstruction)

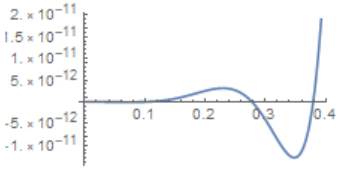
Moar accuracy!

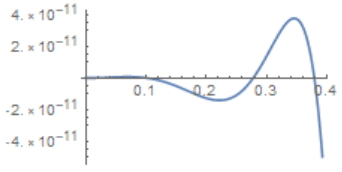
- How about adding another power

```
a = Sin[x] - x-x^3*P[x^2];
b = Expand[Simplify[a/x^3]];
c = ReplaceAll[b, {x^2->y, x->Sqrt[y]}];
d = Expand[c + P[y]];
f = MiniMaxApproximation[d, {y, { 1*^-10,(Pi/8)^2}, 2,0 }];
g = ReplaceAll[f, {y->x^2}];
h = Expand[x + x^3 * g];
```

```
x -0.16666666635050979363 x^3 + 0.008333296492623903734 9 x^5 -0.000197775915790683179 32 x^7
```







Finally where we want to be

Machine coefficients

- We have exquisite coefficients, now we need to turn them into code

> Q: What happens when we take these finely balanced coefficients and truncate them to machine numbers?
>
> A: It gets pretty ugly

So we have a set of beautifully optimal coefficients and now we need to use them in a program

Just truncating and rounding them can undo all the work you have done (see the ESIN instruction in PS2 in the previous talk)

Machine coefficients

- We can reform the minimax inequality to solve with machine constants

$$\sin x \approx C_1 x + C_2 x^3 + x^5 P(x^2)$$

- If Mathematica says the best approximation to $\cos(x)$ over $[0, \pi/4\ ]$ is

$$p = 0.9998864206 + 0.00469021603x - 0.5303088665x^2 + 0.06304636099x^3$$

with an absolute error $\varepsilon = \|cos - p\|_{[0,\pi/4]} = 0.0001135879\ldots$

- We can optimize the polynomial as integers-over-powers-of-2

We can try to round the first few, most significant values to machine numbers

Then use MINIMAX to re-fit the rest of the coefficients

This assumes our coefficients cannot be improved upon

Let's look at the coefficients for COS

Machine coefficients

- Search for $a_0, a_1, a_2, a_3 \in \mathbb{Z}$ such that we minimize

$$\max_{0 \le x \le \pi/4} \left| \cos x - \left(\frac{a_0}{2^{12}} + \frac{a_1}{2^{10}} x + \frac{a_2}{2^6} x^2 + \frac{a_3}{2^4} x^3 \right) \right|$$

- A naïve approach directly translates the coefficients, $\varepsilon = 0.00069397\ldots$

$$\hat{p} = \frac{2^{12}}{2^{12}} + \frac{5}{2^{10}} x - \frac{34}{2^6} x^2 + \frac{1}{2^4} x^3$$

- After optimization, $\varepsilon = 0.0002441406250$

$$p^* = \frac{4096}{2^{12}} + \frac{6}{2^{10}} x - \frac{34}{2^6} x^2 + \frac{1}{2^4} x^3$$

We can try different representations of them as integers over powers-of-2

If you directly translate the values you get about 6x error

After optimization we can find a solution that only has 2x error

Machine Optimized Minimax

- Sollya implements this and more with `fpminimax`
- Uses lattice basis reduction, NP-Hard solution to the Shortest Vector Problem

```
> t = double(pi/64);
> P = fpminimax(sin(x), [|0,1,3,5|], [|single...|], [-t;t], fixed, absolute);
> print(P);
8.3420276641845703125e-3 * x^5 + -0.16666668653488159179687 5 * x^3 + x
> printexpansion(P);
x * (0x3ff0000000000000 + x^2 * (0xbfc5555580000000 + x^2 * 0x3f8115a000000000))
> display=powers!;
> print(P);
34989 * 2^(-22) * x^5 + -2796203 * 2^(-24) * x^3 + x
> display=default!;
> dirtyinfnorm(sin(x)-P(x), [-pi/64;pi/64]);
3.858769943513002373023938267504879412874156507 99e-13
>
```

http://sollya.gforge.inria.fr/sollya-3.0/help.php?name=fpminimax
https://calcul.math.cnrs.fr/attachments/spip/Documents/Journees/nov2010/N-Brisebarre.pdf

This is an area where Mathematica and Maple are falling behind

The state of the art, open source Sollya package can do this for free

Uses an NP-Hard algorithm called “LLL” after LENSTRA, LENSTRA & LOVASZ to optimize the problem over a “lattice basis”, which I do not pretend to understand yet.

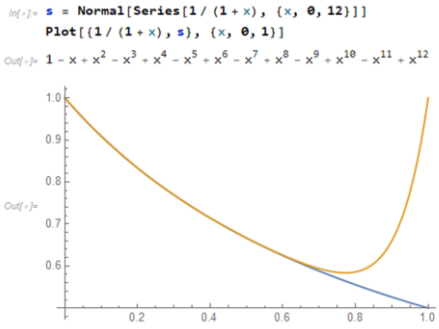
They claim to have applied this to Intel’s “ERF” function and replaced an order 19 polynomial with just 2 coefficients with the same accuracy.

Recip, Sqrt, InvSqrt

- Three closely related functions

- Let's look at reciprocal $1/(1+X)$
- Taylor series converges *super slowly*
- Minimax converges not much better

- Let's calculate 1/10.378 using signed 16.16 fixed point values



New subject, reciprocals and INVERSE SQRT

Taylor series converges way too slowly

Not a good fit for polynomial approximation

We can EVALUATE it though

A worked example, 1/10.378 using 16.16 fixed point values

Recip, Sqrt, InvSqrt

- For an n-bit result we use $k = n/4$ bit chunks
- The input X reduced to $1 \leq Y < 2$ by counting leading zeros (`_lzcnt`)
- Same as $Y = X/2^{\lfloor \mathrm{Log}_2(X) \rfloor}$
- Shift until the leading $1.nnn$ is at the decimal point

0000000000001010.0110000011000100 $= 10.378$

$15 - 12 = 3$

$Y \gg 3 = 1.29725$

0000000000000001.0100110000011000 $= 1.29725$

We want an N-bit result, we break it into N/4-bit chunks

Reduce the number to [1,2] by counting leading zeros and shifting (floats already have the mantissa stored like this)

Recip, Sqrt, InvSqrt

- We use a table to approximate the first k-bits of the reciprocal

$$A = Y \times \hat{R} - 1$$

- $\hat{R}$ is the function $1/Y$ truncated to the k-th bit

```
{ 1.0000000000000000, 0.9411764705882350, 0.8888888888888889, 0.8421052631578946,
  0.8000000000000000, 0.7619047619047619, 0.7272727272727273, 0.6956521739130435,
  0.6666666666666666, 0.6400000000000000, 0.6153846153846154, 0.5925925925925926,
  0.5714285714285714, 0.5517241379310345, 0.5333333333333333, 0.5161290322580645 }
```

- The table index comes from the first k-bits after the decimal

$$Y = \boxed{0000000000000001.0100101101001000}$$

$$\text{index} = 4$$
$$\hat{R} = \text{table}[4] = 0.8000$$

We can use a table to remove the first K-bits

A multiply by 1/K, looked up using k-bits as the index

Recip, Sqrt, InvSqrt

- $A = Y \times \hat{R} - 1 = 1.29725 \times 0.8 - 1.0 = 0.0378$

$$A = \boxed{0000000000000000.0000100110101101}$$

- Next the function $B = f(A)$ will approximate the Taylor series $1/Y$

$$f(A) = C_o + C_1 A + C_2 A^2 + C_3 A^3 + C_4 A^4 + \cdots$$

- But we know that $-2^{-k} < A < 2^{-k}$ (we cancelled the bits) so

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4 + \cdots \qquad \text{where } z = 2^{-k}$$

Next we break the remaining bits into K-bit chunks

Treat them as integers

We're going to Taylor series gives us a simple sum of powers (I know I know)

We re-express A as the sum of small integers times 2^-k

We know that the first A is cancelled, leaving only A2, A3, A4 ...

Recip, Sqrt, InvSqrt

- Inserting A into $f(A)$ gives us

$$\begin{aligned} f(A) = C_0 &+ C_1(A_2z^2 + A_3z^3 + A_4z^4) \\ &+ C_2(A_2z^2 + A_3z^3 + A_4z^4) \\ &+ C_3(A_2z^2 + A_3z^3 + A_4z^4) \\ &+ C_4(A_2z^2 + A_3z^3 + A_4z^4) + \cdots \end{aligned}$$

- Expanding and removing all terms less than 2^{-4k}

$$f(A) \approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6$$

http://ftp.cs.ucla.edu/pub/milos/RECIP/paper.pdf

Inserting this re-expressed A into the Taylor series for 1/X

Multiply through

Remove all values < 2^N (we only want an N-bit approximation)

We are left with a simple sum-of-terms

Each of the Z terms is just a shift of K-bits down the bit string

Recip, Sqrt, InvSqrt

- What are the C_n coefficients? From the Taylor series…

$$\frac{1}{1+A} \approx 1 - A + A_2^2 z^4 + 2A_2 A_3 z^5 - A_2^3 z^6$$

- For Sqrt we get

$$\sqrt{1+A} \approx 1 + \frac{1}{2}A - \frac{1}{8}A_2^2 z^4 - \frac{1}{4}A_2 A_3 z^5 + \frac{1}{16}A_2^3 z^6$$

- Inverse Sqrt we get

$$\frac{1}{\sqrt{1+A}} \approx 1 - \frac{1}{2}A + \frac{3}{8}A_2^2 z^4 + \frac{3}{4}A_2 A_3 z^5 - \frac{5}{16}A_2^3 z^6$$

So what are the C constants?

For RECIP, they are all +1 or -1

For SQRT we have powers-of-2

For INVSQRT we have multiples of 1/16

Recip, Sqrt, InvSqrt

- The values for A_2 and A_3 are the next k-bit chunks of A

$$A = \boxed{0000000000000000.0000100110101101}$$

$A_2 = 9$ $A_3 = 10$

$$f(A) = 1 - A + A_2^2 z^4 + 2A_2 A_3 z^5 - A_2^3 z^6$$

$$= 1 - 0.0378 + 9^2 \times 2^{-4k} + 9 \times 10 \times 2 \times 2^{-5k} - 9^3 \times 2^{-6k}$$

Back to the reciprocal

We have three terms to calculate
- A2 squared
- A2 A3
- A2 cubed

For reciprocal, the constants are just a sign flip of the A2 CUBED term

Recip, Sqrt, InvSqrt

- Looking at the bits and summing the subexpressions

$A =$ 0000000000000000.0000100110100100

$1 - A =$ 0000000000000000.1111011001011011

$9^2 \times 2^{-4k} =$ 0000000000000000.0000000001010001

$2 \times 9 \times 10 \times 2^{-5k} =$ 0000000000000000.0000000000001011

$9^3 \times 2^{-6k} =$ 0000000000000000.0000000000000010

$(1 - A) + f_2 + f_3 - f_4 =$ 0000000000000000.1111011010101100

Working through each subexpression

Notice how the A2 CUBED is shifted so far down that only the top few bits are required

We can short circuit the full multiply here to get only the top bits

We then sum each subexpression to get the bitstring at the bottom

Recip, Sqrt, InvSqrt

- To reconstruct, calculate $g(Y) = M \times f(Y)$
  - For reciprocal, $M = \hat{R}$
  - For square root, $M = 1/\sqrt{\hat{R}}$
  - For inverse sqrt, $M = \sqrt{\hat{R}}$

$$f(A) = 0000000000000000.1111011010101100$$
$$= 0.963562011719$$

$$\hat{R} \times f(A) = 0.8000 \times 0.963562011719$$
$$= 0.770843505859$$

Final reconstruction requires a multiply by a constant

For reciprocal, this is just the value R = 1/Y we looked up in the table earlier.

For the other functions, it's a lookup into a separate table indexed from the original 4-bits of A

Multiply through and …

Recip, Sqrt, InvSqrt

- Final step is to reverse the initial count-leading-zeros shift
  - Note that $bitcount$ can be a negative shift
  - For Sqrt and InvSqrt shift by $2^{-bitcount/2}$
  - This requires an additional multiply by $\sqrt{2}$ if $bitcount$ is odd

$$\hat{R} \times f(A) = 0.770843505859 \quad \boxed{0000.1100010101010101}$$

$$\hat{R} \times f(A) \times 2^{-3} = 0.0963439941406 \quad \boxed{0000.0001100010101010}$$

$$1/10.378 = 0.0963576797071 \quad \boxed{0000.0001100010101010101110011}$$

$$\Delta = 0.0000136855664477$$

The final reconstruction step is to shift the number back to where we started (remember the Leading-Zero-Count?)

Congratulations, we reached the end!

And got last-digit accuracy for a 16-bit fractional value.

TAKEAWAY: You can't look at decimal error values and intuit just how well an approximation is working

What about the Magic Number InvSqrt?

- Hacker's Delight 2nd Ed has a chapter on the Quake style InvSqrt
- Gives a rel. error of 0.0017513304

```
float rsqrt(float x0)
{
  union { int ix; float x; };
  x = x0;
  float xhalf = 0.5f * x;
  ix = 0x5F375A82 - (ix >> 1);
  x = x * (1.5f - xhalf*x*x);
  return x;
}
```

http://www.lomont.org/papers/2003/InvSqrt.pdf

What about the Quake INVSQRT trick?

It can be improved upon

Magic Number InvSqrt

- Changing the 1.5f in the Newton–Raphson iteration to 1.5008909f improves rel. error from 0.00175 to 0.000892

```
float rsqrt(float x0)
{
  union { int ix; float x; };
  x = x0;
  float xhalf = 0.5f * x;
  ix = 0x5F375A82 - (ix >> 1);
  x = x * (1.5008909f - xhalf*x*x);
  return x;
}
```

Changing the constant in the Newton Raphson iteration from 2.0 to this can halve the error

Magic Number InvSqrt

- Adding a second iteration reduces rel. error to -0.0000047
  - This requires a different starting constant 0x5F37599E

```
float rsqrt(float x0)
{
  union { int ix; float x; };
  x = x0;
  float xhalf = 0.5f * x;
  ix = 0x5F37599E - (ix >> 1);
  x = x * (1.5f - xhalf*x*x);
  x = x * (1.5f - xhalf*x*x);
  return x;
}
```

Adding a second NR iteration can give you really low error (in the normal range).

Magic Number InvSqrt

- Removing the Newton-Raphson step altogether gives rel. error of 0.035
- Substantially faster, only two instructions plus a constant load for 0x5F37624F

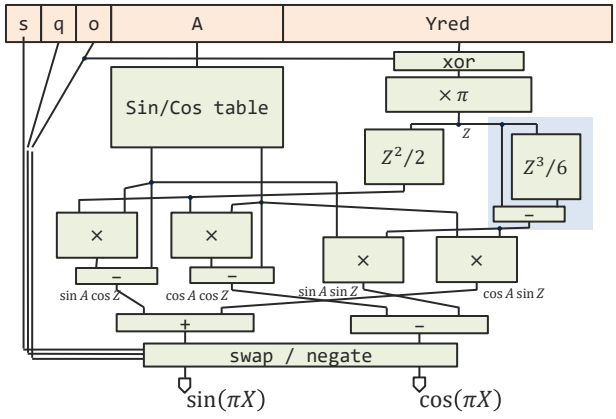
```
float rsqrt(float x0)
{
  union { int ix; float x; };
  x = x0;
  ix = 0x5F37624F - (ix >> 1);
  return x;
}
```

If you want a dirty hack, removing the NR iteration gets you 3.5% error

Substantially faster

FPGA sincospi

- Table-based sincospi for fixed point subdivides to $\pi/64$ or more
  - Approximation using Taylor series



We've approximated some functions, but expressing them in FPGA sometimes involves very different thinking

Not just using parallelism, some of the latest work changes the game.

This circuit is for a SIN-COS-PI function where the input is a value from [-1, 1] in fixed point

Individual bits are taken off the top

S = SIGN

Q = QUADRANT

O = OCTANT

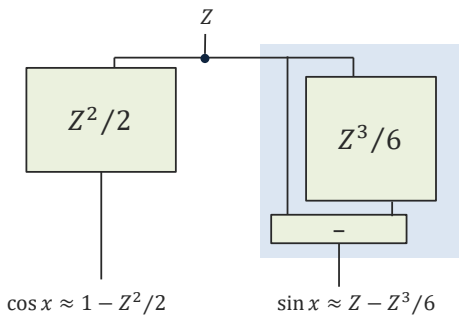
There is an "MULTIPLY-BY-PI" unit

And two units that directly calculate the Taylor sub-expressions

Z^2/2 and Z-Z^3/2

Black box sub-expressions

- Computing $Z^2/2$ we covered earlier, using a squarer and a shift in the output
- But how to calculate $Z - Z^3/6$ without serializing everything?
- Remember building the squarer?



So the problem here is to calculate these subexpressions efficiently

We know how to SQUARE, and a DIVIDE-BY-2 is just a shift, which is just wiring.

How to calculate Z MINUS Z^3 DIVIDED BY 6

Let's look at the bits

Bitheaps

- If you think of the multiply as a column-of-sums

```
1 ..................100100010001
0 .................000000000000.
0 ................000000000000..
0 ...............000000000000...
1 ..............100100010001....
0 .............000000000000.....
0 ............000000000000......
0 ...........000000000000.......
1 ..........100100010001........
0 ........000000000000..........
0 .......000000000000...........
1 .....100100010001.............
-----------------------------------
  000000000101001000110011001000 01
```


Remember the SQUARER

A bit of the result is the sum of the column of bit above

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed

If our multiplier is constant ...

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns

We can TETRIS style collapse the bits

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns

Removes the zeros

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns

One

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns

by

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns

one

Bitheaps

- If you think of the multiply as a column-of-sums
- And number you are multiplying by is fixed
- If we collapse the zeros, we only need to sum the bits in the remaining columns
- This is a bitheap

https://hal-ens-lyon.archives-ouvertes.fr/ensl-00738412v2

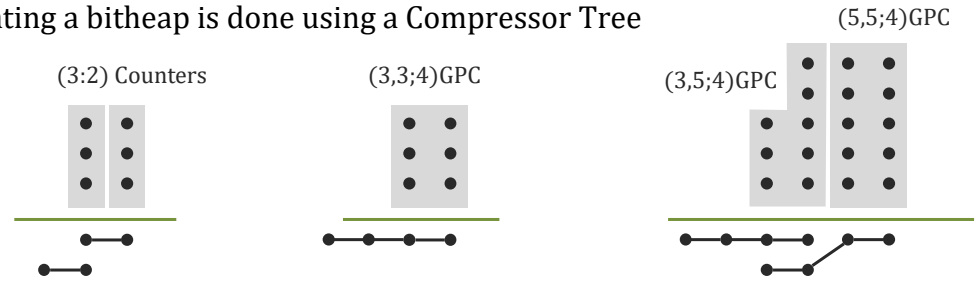
To make us a BITHEAP

If we label each bit with where it came from, we have everything we need to evaluate the expression

We can also label each bit with timings. because some bits may appear later than others

Bitheap Arithmetic and Evaluation

- The sum or product of two bitheaps is another bitheap
- By notating each bit with it's place value position, we can collapse complex expressions into a single bitheap
- Evaluating a bitheap is done using a Compressor Tree



https://www.epfl.ch/labs/lap/wp-content/uploads/2018/05/ParandehAfsharJan08_EfficientSynthesisOfCompressorTreesOnFpgas_ASPDAC08.pdf

Bitheaps have an arithmetic – there are rules for sum and product that produce bitheaps

The result is evaluated using a Compressor Tree made from General Purpose Counters (GPCs)

Simple counters will sum a single column and produce N-bits of output

GPCs can sum multiple columns and produce N-bit sums with place value (e.g. the MSBs are counted as 2)

Layers of GPCs collapse the bitheap into a final sum – see the paper for the algorithm

Flopoco

- Flopoco, an open source app for generating VHDL for math operations
  - Now >10 years old, active research platform (e.g. includes Posit generators!)
  - Install stable v4.1.3, as the latest v5.0 under development is a ground-up rewrite
  - Installs under Linux or WSL in Windows 10
    - The usual Linux find-and-compile-an-academic-library nightmare
    - But if I can do it, you can do it

```
$ ./flopoco FixSinCos
FixSinCos: Computes (1-2^(-w)) sin(pi*x) and (1-2^(-w)) cos(pi*x) for x in -[1,1[, using
tables and multipliers.
  lsb (int): weight of the LSB of the input and outputs
  method (int): 0 for table- and mult-based, 1 for traditional CORDIC, 2 for reduced-iteration
     CORDIC (optional, default value is 0)
```

https://hal.inria.fr/hal-02161527/document

Flopoco is an actively developed FPGA math generator

Contains routines that generate optimized VHDL for your design – you can specify how many bits and which algorithms to use

Generates tables, entities, processes, timings and public interfaces

One of the many operators is exactly this SinCosPi generator for any number of bits.

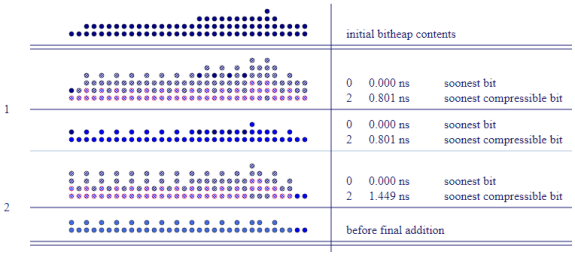
TAKEAWAY: If you are just plugging together vendor supplied operators, you can upgrade to modern methods today

Flopoco SinCosPi

- To generate the SinCosPi function described above:

```
$ ./flopoco  frequency=300 target=Virtex6 FixSinCos lsb=-32 method=0
             name=MyFixedPointSinCos generatefigures=1 outputfile=MyFixSinCosPi.vhdl
```

- Generates VHDL for the component
- Outputs the bitheaps for $Z^2/2$ and $Z^3/6$
  - Bits arrive at different times
  - Hover mouse over bits in .svg figures for more information
- 32-bit SinCosPi in 2 clocks at 300MHz



http://flopoco.gforge.inria.fr/bib/flopoco.html

http://perso.citi-lab.fr/fdedinec/recherche/publis/2013-HEART-SinCos.pdf

Generates a VHDL package to any number of bits of accuracy

Can generate .svg images – hover your mouse over the bits to see where they came from and their timings

TAKEAWAY: If you are just plugging together vendor supplied operators, upgrade to modern methods

Bipartite tables

- Bipartite tables is a method for fitting more accuracy into less storage
- First we split a [0.5,1.0] mantissa into three sections of k-bits (where $k = n/3$)

$$x = \boxed{x_1 \mid x_2 \mid x_3}$$

$$x \approx x_1 + x_2 2^{-k} + x_3 2^{-2k}$$

- Then we take the order-1 Taylor expansion of a function $f(x)$

$$f(x) = f\left(x_1 + x_2 2^{-k}\right) + x_3 2^{-2k} f'\left(x_1 + x_2 2^{-k}\right) + \varepsilon_1$$

- And substitute the order-0 Taylor expansion of $f'\left(x_1 + x_2 2^{-2k}\right)$

$$f(x) = f\left(x_1 + x_2 2^{-k}\right) + x_3 2^{-2k} f'(x_1) + \varepsilon_1 + \varepsilon_2$$

Next we're going to have another look into lookup tables.

For low accuracy functions, tabulating every entry in the n-bit range is totally possible.

On FPGA you are balancing the table storage vs. the number of gates for a full, last bit accurate calculation, and sometimes the full table wins.

Bipartite tables were invented to squeeze fully tabulated functions into a smaller space

Designed for parameters in [0.5, 1.0]

We divide the parameter into three K-bit chunks

Taylor expand an arbitrary function F(X)

Then substitute the derivitave term with a simpler approximation

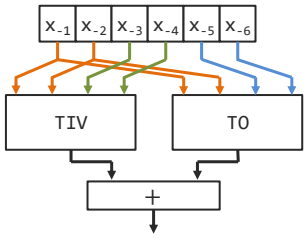
Bipartite tables

- This gives us the **Bipartite formula**

$$f(x) = \alpha(x_1, x_2) + \beta(x_1, x_3) + \varepsilon$$

where

$$\alpha(x_1, x_2) = f\left(x_1 + x_2 2^{-k}\right)$$
$$\beta(x_1, x_3) = x_3 2^{-2k} f'(x_1)$$



$$\text{and } \varepsilon \leq \left(\tfrac{1}{2}2^{-4k} + 2^{-3k}\right) \max f'' \approx 2^{-3k} \max f''$$

- The error proves that a function can be approximated with n-bits of accuracy by the sum of 2 terms looked up using $2n/3$-bit table indexes

http://perso.ens-lyon.fr/jean-michel.muller/MullerDevReliableComputing99.pdf

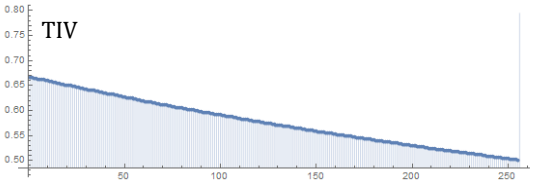
Doing this we can guarantee that the error is less than 2^-3K so it never appears in the output

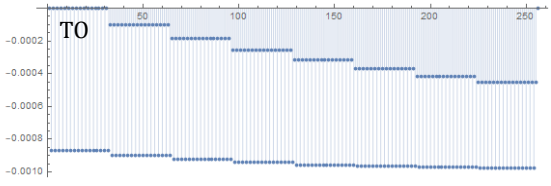
We then look up the values in two tables of size 2N/3 instead of 2^N

Sum the result

Bipartite tables

- Applying this to the reciprocal $f(x) = 1/(1+x)$ over $[0.5, 1.0]$
- For a 12-bit table:
  - Set $k = 4$ to produce two tables of 256 entries = 8-bit index





- The α table (TIV) must store 12-bit values
- The β table (TO) has values where the most significant $\sim 2k = 8$-bits are zero

Apply this to the reciprocal for a 12-bit table

K = 4, producing two 8-bit tables with 256 entries

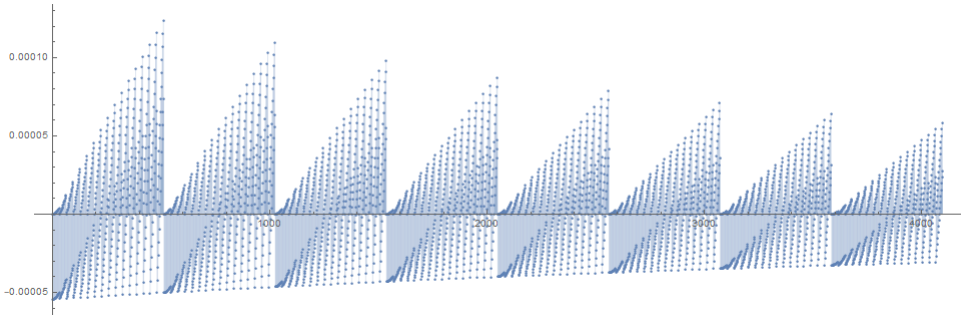
Also, the leading bits of the BETA table are all zeros, do not need to store them

Terminology:

- TIV = Table of Initial Values
- TO = Table of Offsets

Bipartite tables

- Reconstruction produces a full tabulation of $1/(1+x)$ over 4096 entries
- The abs error is guaranteed to be below $2^{-3k} \max f'' = 0.0001447$

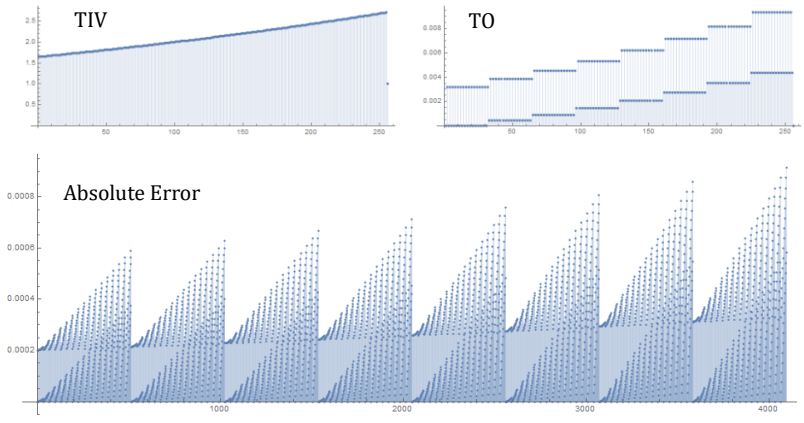


Using the two 256-entry tables we can reconstruct the full 4096 entries

The reconstruction error is guaranteed below 0.0001447

Bipartite tables

- The same exercise with $f(x) = e^x$ gives us tables with error < 0.004025



Applying the same to EXP function

We get an error below 0.004 from the two tables

(Yeah, not what's diagrammed, I'm doing something wrong – but it works! No really!)

Tripartite tables

- This process can generate Tripartite tables with $k = n/5$ bits per section

$$x = \boxed{x_1}\boxed{x_2}\boxed{x_3}\boxed{x_4}\boxed{x_5}$$

$$f(x) = \gamma(x_1, x_2, x_3) + \delta(x_1, x_2, x_4) + \theta(x_1, x_5) + \varepsilon$$

where

$$\gamma(x_1, x_2, x_3) = f\left(x_1 + x_2 2^{-k} + x_3 2^{-2k}\right)$$
$$\delta(x_1, x_2, x_4) = x_4 2^{-3k} f'\left(x_1 + x_2 2^{-k}\right)$$
$$\theta(x_1, x_5) = x_5 2^{-4k} f'(x_1)$$

$$\text{and } \varepsilon \leq \left(\tfrac{1}{2}2^{-6k} + 2 \times 2^{-5k}\right) \max f'' \approx 2^{-5k+1} \max f''$$

- Giving us a largest table size of $3n/5$ address bits

We can subdivide the mantissa further and produce Tripartite tables using k = n/5 bits per section

Lower error guarantees, slightly larger tables

- Tables an OK size for binary32 functions
- Tables too large for binary64 evaluation

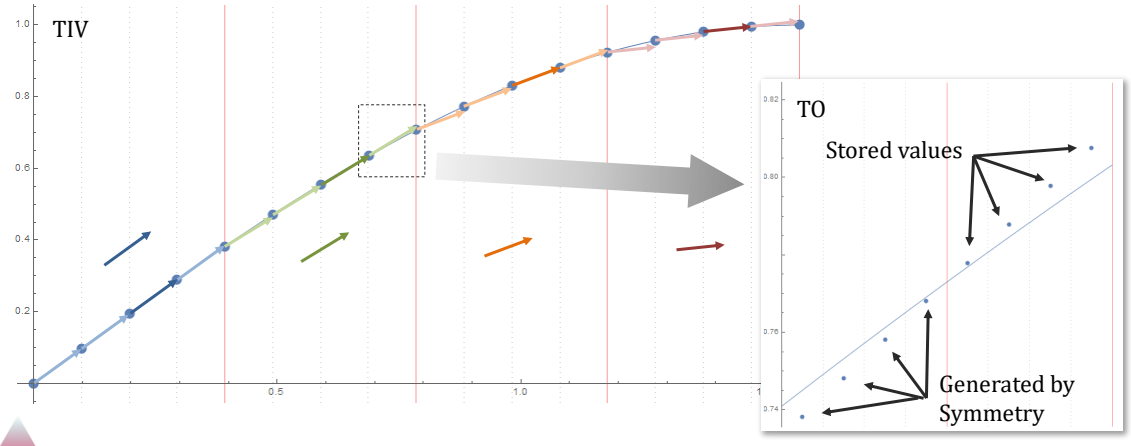
There is of course a generalization of the Multipartite idea to any number of divisions, with the usual diminishing returns for the complexity.

There is also a generalization to higher orders of Taylor series, producing a cascaded tree of tables to be summed in parallel.

However, producing many additions to avoid a single multiplication quickly becomes silly.

Multipartite tables

- The theory has been extended and generalized, e.g. here $\alpha = 4,\ \gamma = 2$



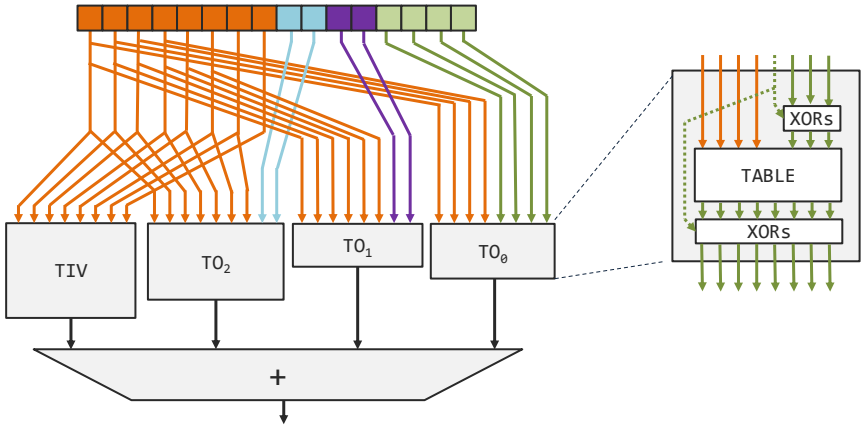
Only one gradient is chosen per block of the TIV and used for each subsection, so chose a representative one

This is SIN over [0, Pi/4] with 16 INITIAL VALUES and 4 OFFSETS, but using symmetry to double that number

This halves the size of the TO (Table of Offsets) and only takes an XOR on the address bits

General Multipartite tables

- General Multipartite architectures can get very interesting



https://hal-ens-lyon.archives-ouvertes.fr/ensl-00542210/file/2005-TC-Multipartite.pdf

General Multipartite Tables can get wild

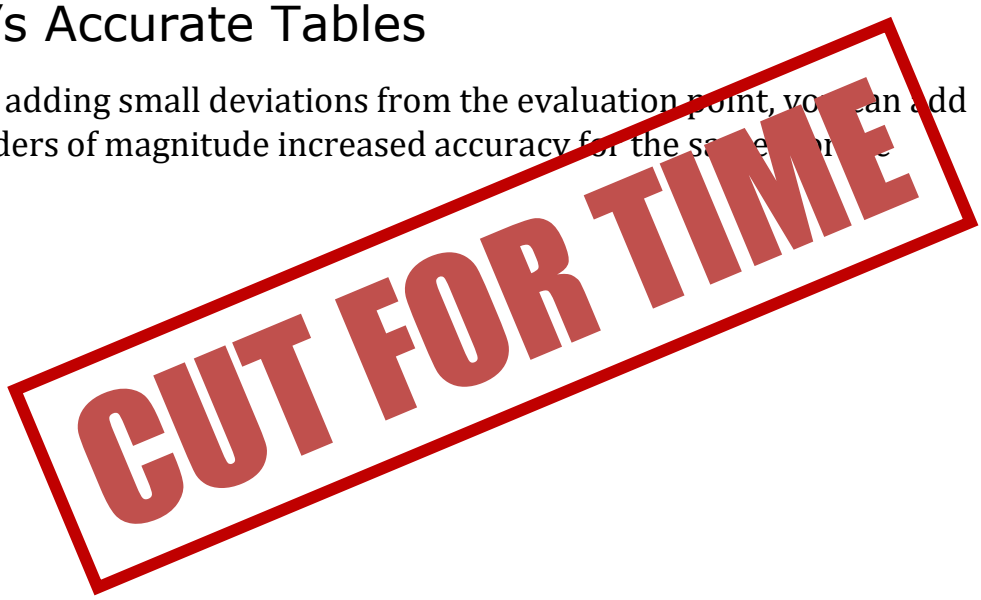
Here the first bit of the green integer is treated as a sign bit that negates a section of the address and output bits to reflect the TO points

The wins from halving the table size more than make up for the increased circuitry

NOTE: FloPoCo can generate multipartite tables of arbitrary functions using Sollya internally to evaluate the tables

Gal's Accurate Tables

- By adding small deviations from the evaluation point, you can add orders of magnitude increased accuracy for the same error



Probably the most difficult part of this talk to explain as it's not very visual

When using table lookups at non-power of 2 points (e.g. multiples of PI/64)

We approximate the function at that value and encode it into a stored value, e.g. a float

If we were allowed to diverge slightly from the evaluation point, we may be able to find a much better machine value to encode

Store both the new argument and the encoded value in the table

Finding these points involves a massive search and optimization problem, but for mission critical tables the work can be valuable

Evaluating Trig at Regular Steps

- We know that $\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$
- To evaluate this at regular angle steps of θ

Precalculate $S = \sin(\theta)$
$C = \cos(\theta)$
Set $x_0 = 0, y_0 = 1$
Iterate:
$x_n = x_{n-1} * C + y_{n-1} * S$
$y_n = y_{n-1} * C - x_{n-1} * S$

- This is the "Coupled" or "Standard Quadrature" oscillator
- It takes 4 multiplies and two add/subs. Can we do this quicker?

Sometimes you need to evaluate sines on a grid – making sounds, waves, modeling, etc.

There are fast ways to do this without using trig

The DSP world knows these are Discrete Oscillators

The canonical one is the Coupled Quadrature model

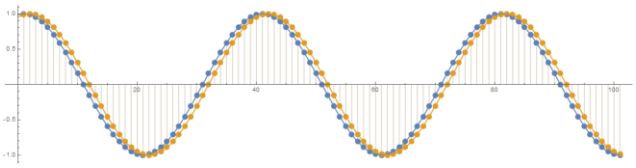
Biquad Oscillator

- We can use a single multiply-add:

$$\cos(\phi + \theta) = 2\cos(\theta)\cos(\phi) - \cos(\phi - \theta)$$

$$x_n = 2\cos(b)\, x_{n-1} - y_{n-1}$$
$$y_n = x_{n-1}$$

- Produces two copies of the cosine, phase shifted by θ (e.g. one step)



The Biquad oscillator takes a single MULTIPLY-ADD and a swap

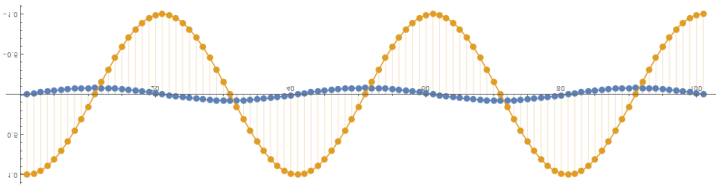
It generates a sine wave and another offset by one step

Digital Waveguide

- What if you want to produce both sine and cosine?

$$t = \cos(\theta)\,(x_{n-1} + y_{n-1})$$
$$x_n = t - y_{n-1}$$
$$y_n = t + x_{n-1}$$

- Produces both signals in quadrature, but at different magnitudes



Want to produce both SIN and COS at the same time?

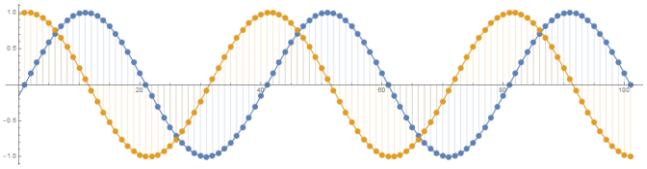
If you don't mind them being at different amplitudes, you can get quadrature

Staggered Update

- To fix the magnitudes we can loosen phase preconditions

$$k = 2\sin(\theta/2)$$
$$y_n = y_{n-1} - kx_{n-1}$$
$$x_n = x_{n-1} + ky_n$$

- Produces unit magnitude waves offset by $\theta/2$ from quadrature



You need the SIN and COS to be the same amplitude?

OK, we can get them cheaper but not at exactly 90 DEGREES offset

The Underlying Theory

- Each of these can be written in matrix form

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}$$

- The preconditions for stable oscillation are:

$ad - cb = 1$ which implies $\det(M) = 1$, the gain is unity

$|a + d| < 2$ which implies M has complex eigenvalues

C.Turner, "Digital Resonators", comp.dsp Conference, Apr 2010

This menagerie of functions has an underlying theory

This paper from C Turner lays it all out

Every Discrete Oscillator can be rewritten in matrix form

There are two preconditions for oscillation:

1. the matrix determinant is exactly ONE

2. the matrix has complex Eigenvalues

The Underlying Theory

- Each of these can be written in matrix form

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}$$

- M can be analyzed as the product of three complex matrices $\mathrm{A} = SDS^{-1}$

$$S = \begin{bmatrix} 1 & 1 \\ \psi e^{i\theta} & \psi e^{-i\theta} \end{bmatrix} \qquad \mathrm{D} = \begin{bmatrix} e^{i\theta} & 0 \\ 0 & e^{-i\theta} \end{bmatrix}$$

- Successive rotation raises $A^n = (SDS^{-1})(SDS^{-1})(SDS^{-1}) \ldots$

$$= SD^nS^{-1}$$

C.Turner, "Recursive Discrete-Time Sinusoidal Oscillators", IEEE Signal Processing, 2003

Why?

Because the matrix can be expressed as the product of three complex matrices S-D-INVERSE-S

Iterating raises this expression to an integer power

Which iterates the D matrix and leaves the S matrices untouched

See the link for a better explanation than I can give

Catalog of Oscillators

- Coupled Quadrature

$$k = \sin(\theta)$$
$$M = \begin{bmatrix} \sqrt{1-k^2} & k \\ -k & \sqrt{1-k^2} \end{bmatrix}$$

- BiQuad Oscillator

$$k = 2\cos(\theta)$$
$$M = \begin{bmatrix} k & -1 \\ 1 & 0 \end{bmatrix}$$

- Type A

$$k = 4\sin^2(\theta/2)$$
$$M = \begin{bmatrix} 1-k & -k \\ 1 & 1 \end{bmatrix}$$

- Magic Circle

$$k = 2\sin(\theta/2)$$
$$\mathrm{M} = \begin{bmatrix} 1-k^2 & k \\ -k & 1 \end{bmatrix}$$

- Digital Waveguide

$$k = \cos(\theta)$$
$$M = \begin{bmatrix} k & k-1 \\ k+1 & k \end{bmatrix}$$

- Type B

$$k = 2\cos(\theta)$$
$$M = \begin{bmatrix} 0 & 1 \\ -1 & k \end{bmatrix}$$

- Quadrature Staggered

$$k = \cos(\theta)$$
$$\mathrm{M} = \begin{bmatrix} k & 1-k^2 \\ -1 & k \end{bmatrix}$$

- Reinsch

$$k = 2\cos(\theta) - 1$$
$$M = \begin{bmatrix} k & 1 \\ k-1 & 1 \end{bmatrix}$$

- Type C

$$k = 4\cos^2(\theta/2)$$
$$M = \begin{bmatrix} k-1 & k \\ -1 & -1 \end{bmatrix}$$

C.Turner, "Oscillator Resonator Parameter Table for Goertzel Applications", Apr 2010

From this analysis several new forms of Discrete Oscillator were found

Some were well known to the DSP community, one was even patented (no more).

Changing Amplitude with Auto Gain Control

- We can calculate the power P (sum of amplitudes squared)

$$P = \frac{x_n^2 - \frac{b}{c}y_n^2 - 2\sqrt{\frac{-b}{c}}x_n y_n \cos(\phi)}{\sin^2(\phi)}$$

- If we restrict ourselves to quadrature oscillators, it simplifies to

$$P = x_n^2 - \frac{b}{c}y_n^2$$

- Apply gain to the state variables x_n, y_n using convergence factor q

$$G = \frac{P_0^q}{P^q} \qquad \approx 1 + q - q\frac{P}{P_0}$$

Evaluating the oscillator over time, errors can accumulate and appear as rising or falling amplitude

So for long-term use (e.g. software synths) we need to apply AUTO GAIN CONTROL

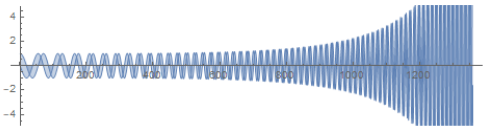
The full expression for amplitude is hairy

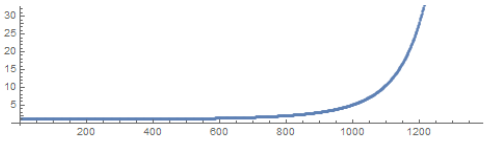
Assuming quadrature oscillation simplifies it a lot

And of course there's a Taylor Series approximation that's cheaper than raising to a power

Changing Frequency

- To change frequency, you update k and recalculate M for each change in θ_i
- Doing this leads to the problem of amplitude change





- Assuming quadrature oscillation, we combat this using AGC:

1. Update the state variables x_n, y_n
2. Update the matrix M for the new frequency θ_i
3. Calculate power P using the updated variables b, c
4. Use the Automatic Gain Control to re-normalize the state variables x_n, y_n

Changing frequency is a matter of updating the Matrix term and re-applying it next iteration

Doing this can lead to amplitude change

So couple it with AUTO GAIN CONTROL to keep your oscillator stable

Hyperstable Quadrature Oscillator

- Changing frequency only works for frequencies up to 1/4 sampling rate
- Martin Vicanek published a quadrature oscillator that takes two half-steps:

$$
\begin{aligned}
k_1 &= \tan(\theta/2) \\
k_2 &= \sin(\theta) = 2k_1/(1+k_1^2) \\
w_n &= x_{n-1} - k_1 y_{n-1} \\
y_n &= y_{n-1} + k_2 w_n \\
x_n &= w_n - k_1 y_n
\end{aligned}
$$

- This remains stable from VLF to KHz rates independent of sampling rate

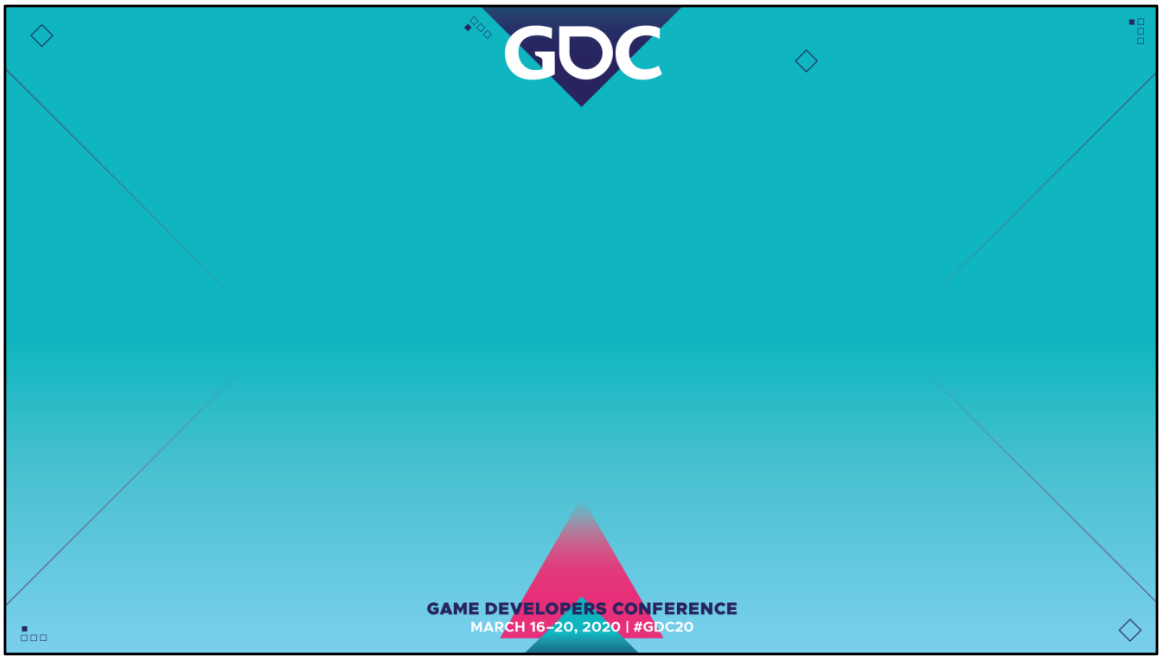
https://vicanek.de/articles/QuadOsc.pdf

Most oscillators can only handle frequencies up to 1/4 of SAMPLING RATE before becoming unstable

A new form of double-step oscillator was recently proposed

Stable from VLF to KHz frequencies

Almost independent of sampling rate


GDC
GAME DEVELOPERS CONFERENCE
MARCH 16–20, 2020 | #GDC20