Approximating Sine

SIG Math Fall 2025

Because it's the inaugural meeting...

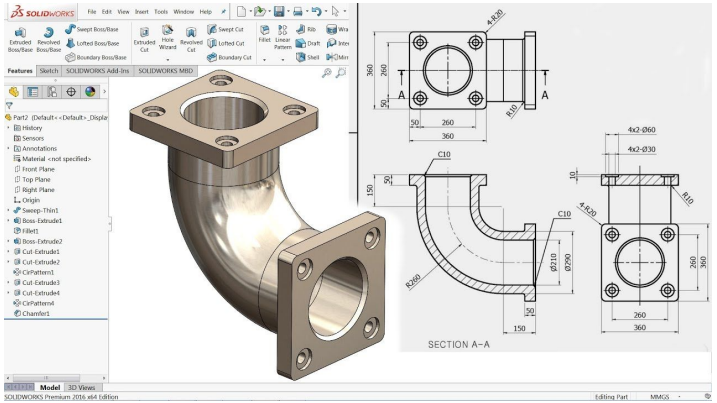
- I really like programming and math. I think there are a lot of ways that math knowledge can improve your programming skills!
- I always liked channels like Aleph 0, Numberphile, 3Blue1Brown, etc. that showed really cool applications of math that you wouldn't learn in school.
- Software always runs on a physical device, and there is definitely a lot of engineering in software engineering.
    - Algorithms, discrete math, etc. are all the most well-studied areas of math + CS, but I think they're a bit overstudied. Lots of times these topics are too theoretical to apply in the real world!

How SIG Math will operate

- Each week will be an approximately 45-60 minute long presentation on a topic in the intersection of programming and math
  - I am open to changing this. It might be the case that 30 minute presentations are better and then some discussion time, or something else entirely! We'll see how things pan out.
- Some topics will be more programming heavy, some more math heavy, some in the middle. That way there's a wide variety of topics and there's something for everyone.
- Please feel free to talk, ask questions, give feedback, etc!
  - I do a ton of research for these presentations and there's often a lot of stuff that I think is super cool but I have to cut out.
  - If you want to hear about a particular topic, just suggest it! I pick topics based on what I think is interesting to me, but I would prefer if there is a topic that is interesting to you all!
  - One theme for this SIG is making connections between different topics!

Now let’s get started!

Background



- Sine and cosine are used in many types of computer programs. Often these functions are called **thousands to millions of times a second**.
- There are many ways to implement sine and cosine, giving different tradeoffs such as accuracy vs speed.
- Because sine produces transcendental outputs for most input values, **any sine function we make will necessarily be an approximation**.

Problem statement

- Most programming languages include sine in their standard library (Math.sin() for Java, sin() for C, math.sin() for Python, etc.)
- If you're a language designer including a function in the standard library, it needs to work for as broad as a user base as possible. Therefore any inaccuracy is not desirable – you don't want users not being confident in the results!
- So a language designer's goal is to make the **fastest possible sine function that meets the precision threshold**.

If you took linear algebra, another way to think about this problem is to find an approximation A that minimizes the L^{∞} norm between A and sin.

So how precise do we have to be?

- The highest precision supported by most programming languages is a *double floating point*, using 64 bits (8 bytes) of data.
- Typically, a double value represents **15-17 decimal places of precision**.
- To recap: the goal is to make a sine function that is:
  - **Precise to 15-17 decimal places**
  - **For all inputs in the domain**
  - **As fast as possible**

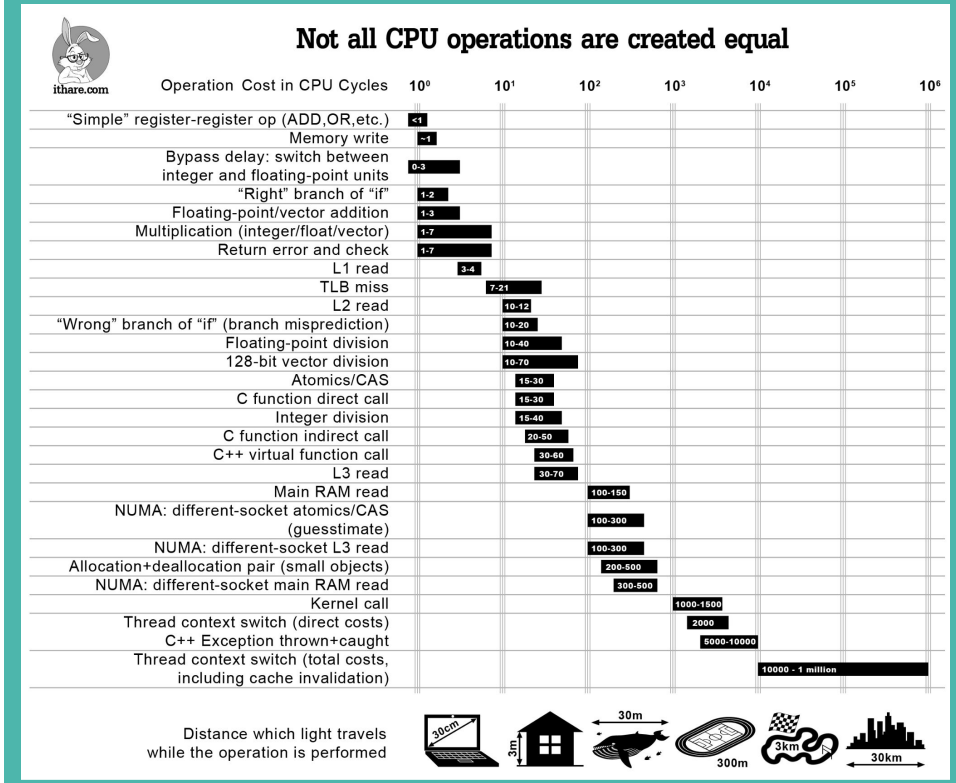
If you are interested in the intricacies of floating point, the manuscript *What Every Programmer Should Know About Floating Point* is very informative! (Or stick around for a future presentation)

One important note....

You might notice I'm only talking about sine, and not cosine. Cosine is just a shifted version of sine, and all the methods discussed in this presentation apply to cosine just as much as they do to sine.

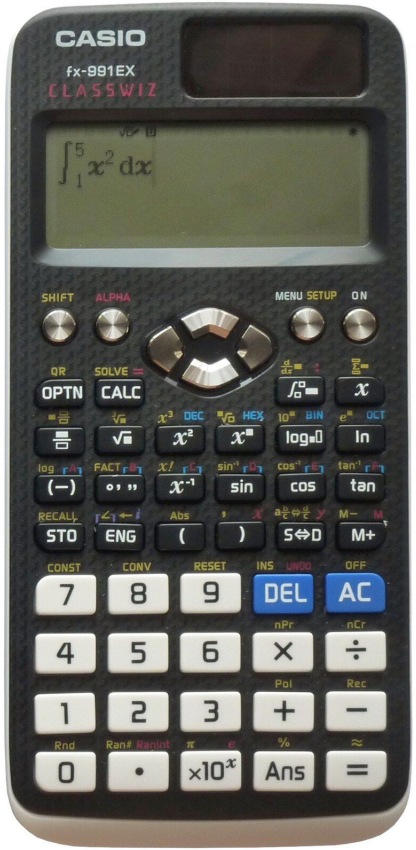
CPU operation speed chart

- This chart is *super* useful to be familiar with
- Additions and multiplications are very fast while division is slower



A brief note...

- If you read about the history of this problem, you will run into an algorithm called CORDIC
- CORDIC was created in an era when multiplications were significantly slower than additions. It uses only additions, bit shifts, and lookup tables.
- Modern computers aren't like this anymore, so this algorithm isn't really used except for places like pocket calculators
- This presentation will be focused on how sine is calculated on modern computers



If you remember from Calculus II...

- Taylor's theorem tells us that we can approximate sin with a polynomial!
  - Because polynomials are comprised of adds and multiplies, they are fast to evaluate
- The Taylor series for sin(x) is quite simple and easy to write a program for:

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

- Although the formula goes on to infinity, we will only ever use a finite amount of terms.
- So it seems like this is a great solution! Unfortunately, this is just the beginning of the process. There are a few issues with this formula that we need to address.

Problems with Taylor series

- We can think about the efficiency of the algorithm in two ways:
  - How much work is being done per iteration
  - How many iterations it takes to converge
- For each iteration, we have to do 3 multiplications to compute the numerator and denominator, then a division to compute the term value, then an addition to add the term to the overall sum.

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

Ops/iter:
3 mul
1 div
1 add

Optimizing Taylor Series calculation: **Reducing the work per iteration**

Horner's method

- There's actually a simple way to make this more efficient. Most of you have probably spent years practicing this very technique in math class.
- We can use factoring!
- Let's demonstrate on an approximation of sin up to the x^7 term. We can first factor out the x term, like so:

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

$$sin(x) = x(1 + \frac{x^2}{3!} + \frac{x^4}{5!} + \frac{x^6}{7!})$$

- Note the final 3 terms all are multiples of x^2, so we can factor that too:

$$sin(x) = x(1 + x^2(\frac{1}{3!} + \frac{x^2}{5!} + \frac{x^4}{7!}))$$

- You can probably see the pattern by now. Let's finish the rewrite:

$$sin(x) = x(1 + x^2(\frac{1}{3!} + x^2(\frac{1}{5!} + x^2(\frac{1}{7!}))))$$

Ops/iter:
2 mul
1 div
1 add

What about the division?

- We've removed a multiplication operation, but division is still the most time-consuming operation in the process right now.

$$sin(x) = x(1 + x^2(\frac{1}{3!} + x^2(\frac{1}{5!} + x^2(\frac{1}{7!}))))$$

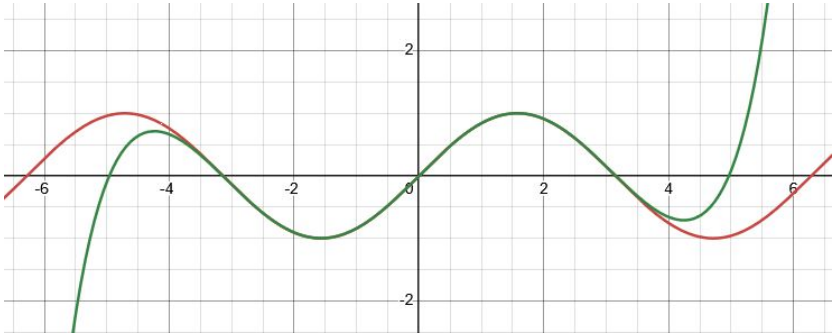
- However, we don't actually need to compute the constant values every time. So we can just compute them once and store them in an array!
- This replaces a multiplication and division operation (~28 cycles) with a L1 read (~3 cycles)

Ops/iter:
1 mul
1 read
1 add

Optimizing Taylor Series calculation: **Reducing the number of iterations**

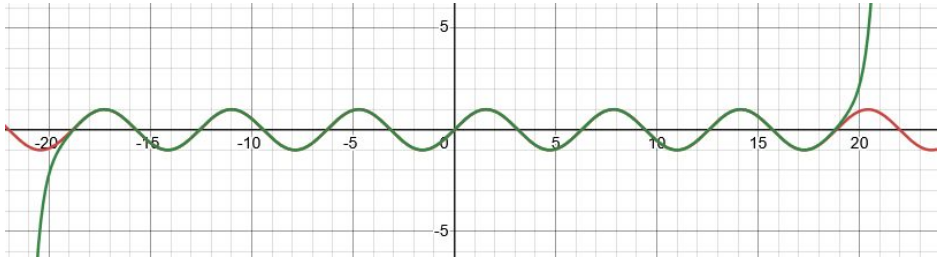
Problems with Taylor series (reprise)

- How many terms do you think it would take to compute sin(40)?
- Here's a graph of our Taylor series for 5 terms. The function is only good for values in the range of -3 and 3. Beyond that, it goes to infinity:



Problems with Taylor series (reprise)

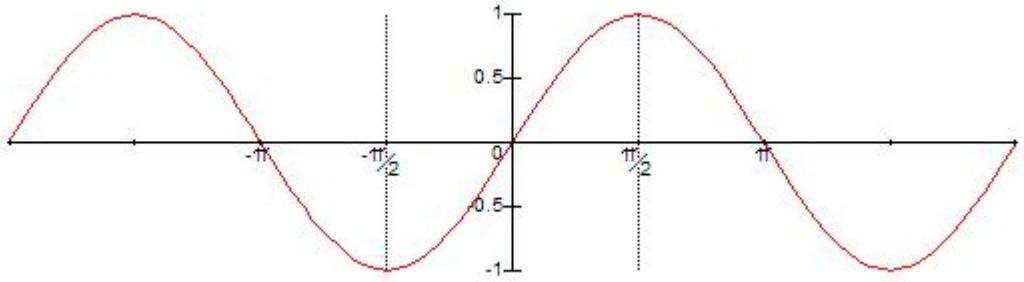
- So maybe we can just fix this by adding more terms? Here's the graph of our Taylor series with 25 terms:



- Not much better. The acceptable input range is certainly larger, but 25 terms takes a long time to compute and it still won't work if our values are even remotely large. Besides, at 25 terms the values are so small that floating-point precision errors come into play. How do we fix this?

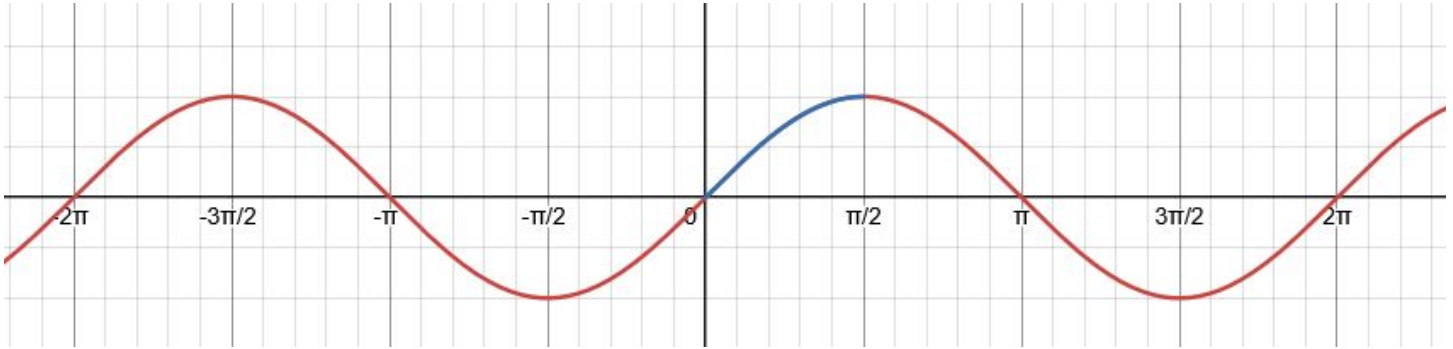
Range reduction

- Instead of trying to make our function approximate the whole input range, we can make a **sine function that works on a small input range**, and then **map any inputs to that smaller domain**.
- Remember that sine repeats every 2pi – in other words, $sin(x) = sin(2n\pi + x)$
- So instead of calculating the Taylor approximation for x directly, we can calculate the approximation for x modulo 2pi and get the same result!



Range reduction

- Furthermore, we can note that sine is odd around the origin: $sin(-x) = -sin(x)$
- Sine is also symmetric around pi/2, since: $sin(x) = sin(\pi - x)$
- Ultimately we've reduced our domain to [0, pi/2].
  - Theoretically we could reduce it further to [0, pi/sqrt(2)], but this reduction isn't always performed for sine. It is very useful for arcsine though!



Putting it all together

- Let's put all our tools together – Horner's method, precomputed coefficients, and range reduction. How many iterations do we need to compute sine to double precision (15-17 decimal places)?
- The answer? **11 terms!**
- The time it takes to compute this approximation is roughly **50-150 CPU cycles**, depending on a ton of factors.
    - About 20-30% of the time is spent doing range reduction (though for large inputs, this can easily take up most of the time!)
    - The rest of it is spent doing the polynomial approximation itself

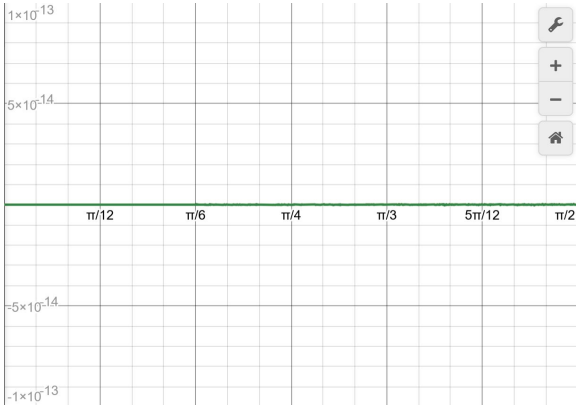
We have an approximation. Can we do better?

Can we do better?

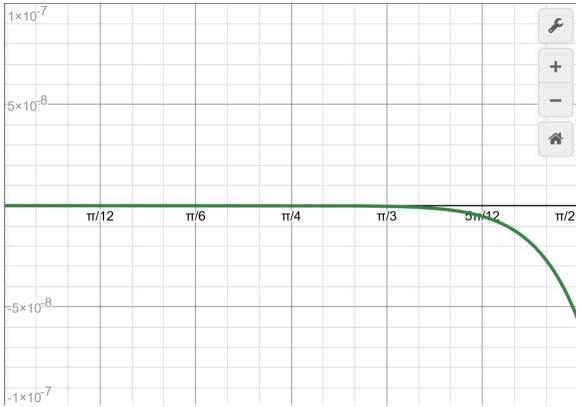
- The problem is that **Taylor series are just not that good** for numerical approximations like this.
  - In fact, Taylor series work exceptionally *well* for sine – for other functions they are much worse!
- I don't mean to discount Taylor series – they are very important in math and are the foundation of calculus. Yes, math says that the Taylor series for sine converges everywhere. But math isn't reality! It's important to remember the distinction between math theory and software engineering.
- 11 terms is just way too much! We can use less terms and get the same precision, and the secret is to **pick better coefficients.**

Error graphs

- Let’s plot the error as a function of the domain (from 0 to pi/2, since we implemented range reduction)



11-term Taylor approximation



6-term Taylor approximation

- The error is really good near zero, and really bad far from zero.
- We’re basically doing a lot of extra work for ~80% of the inputs to bring down the error for the bad ~20%.

Picking better coefficients

- Picking the best coefficients is actually a surprisingly hard task and requires a good deal of knowledge in numerical methods. I will try to summarize the key points in an easily digestible way, but there is a ton of really interesting content if you are interested in this stuff!
- The main algorithm at play is the Remez algorithm, also known as minimax. The idea is that instead of computing the coefficients for x, x^2, x^3, etc, we compute the coefficients of **Chebyshev polynomials**.

What are the Chebyshev polynomials exactly?

- The Chebyshev polynomials are:

$$T_0(x) = 1$$
$$T_1(x) = x$$
$$T_2(x) = 2x^2 - 1$$
$$T_3(x) = 4x^3 - 3x$$
$$T_4(x) = 8x^4 - 8x^2 + 1$$
$$T_5(x) = 16x^5 - 20x^3 + 5x$$
$$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1$$
$$T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x$$
$$T_8(x) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1$$
$$T_9(x) = 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x$$

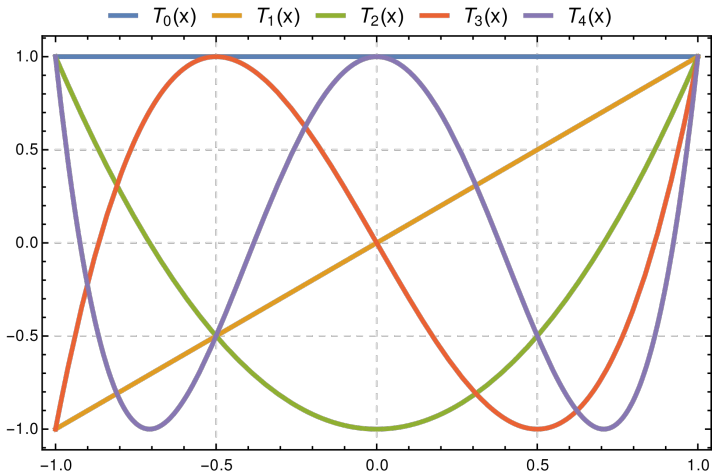
If you took linear algebra, another way to think about this is that 1, x, x^2, x^3... forms a basis for the polynomials, and the Chebyshev polynomials are just another basis for polynomials.

- This seems kind of random, but it's actually just the polynomials where

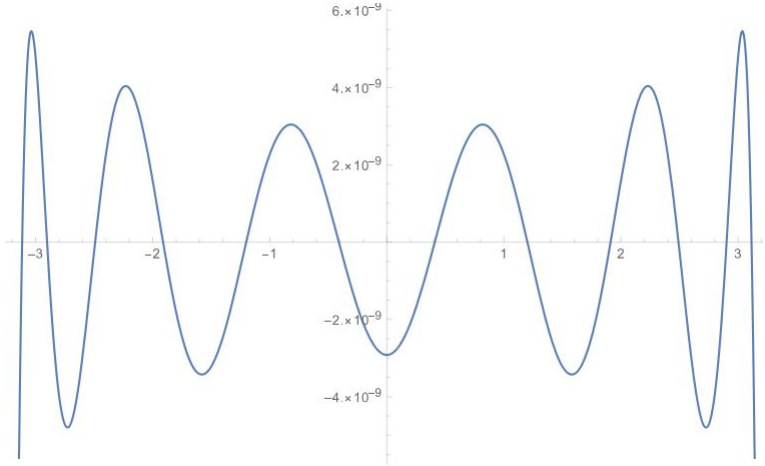
$$T_n(x) = cos(n * arccos(\theta))$$

Why Chebyshev polynomials are awesome

- Chebyshev polynomials oscillate between -1 and 1, and have their values equally distributed across the interval
  - What this means is that the error will also be equally distributed across the interval!



Graph of the Chebyshev polynomials



6-term Chebyshev approximation error graph

Putting it all together

- We finally reached the end of our journey! There's always more to learn, but this is a pretty good idea of what's going on when computers calculate sine.
- To recap:
  - We first perform **range reduction**...
  - then do a **polynomial approximation**...
  - Using **Horner's method** to evaluate it quickly...
  - With **precomputed coefficients** based on **Chebyshev polynomials**
- Not all platforms and programming languages compute sine the same way. Some platforms use lookup tables or other approximations. It's very inconsistent, though the results are usually quite similar!
- As always, all of these methods apply just as well to cosine.

Why does this matter?

- We analyzed how sine is calculated in detail, but *who cares*? It's not like you're reimplementing sine every day... right?

Why might you write your own sine function?

- **Performance** – if you don't need 15 decimal digits of precision, why do the work to calculate it? It wastes precious time, especially in an environment like a video game, VR, robotics, embedded system, etc.
- **Determinism** – the algorithm for sine isn't guaranteed to be the same across platforms. For scientific simulations or multiplayer games, having a single sine function that you know is the same everywhere is important!
- **Precision** – if you are working on a scientific project where precision is paramount, having a sine implementation that goes to *more* than 17 decimal places is important (e.g. NASA, cryptography).

Case study: Minecraft Optifine

- Being a 3D game, Minecraft uses sine and cosine *everywhere*. It probably uses somewhere on the order to 10k-100k sine/cosine calls per second.
- Knowing this, Mojang decided to write their own implementation of sine and cosine that didn't have maximum precision – it was only accurate to around 5 decimal places.
- In fact, the famous mod Optifine has a setting (Fast Math) that increases FPS simply by making the sine implementation slightly less precise.

Thanks for listening!

Any questions?