
A BRIEF HISTORY
OF
RENDERING MATH
(and its future online)
GitHub
@gjtorikian

Hi there, how's it going. Thanks for joining today. I'm really grateful you've decided to attend this talk. I know that you have a lot of choices in attending talks and I really appreciate your being here.

I'm going to provide a brief history of rendering math, from its humble origins to its present usage. This talk is part history, part technical; towards the end of the talk I'll be introducing some exciting new advancements that GitHub is undertaking, so if math rendering is your thing, be sure to stick around for the new stuff.

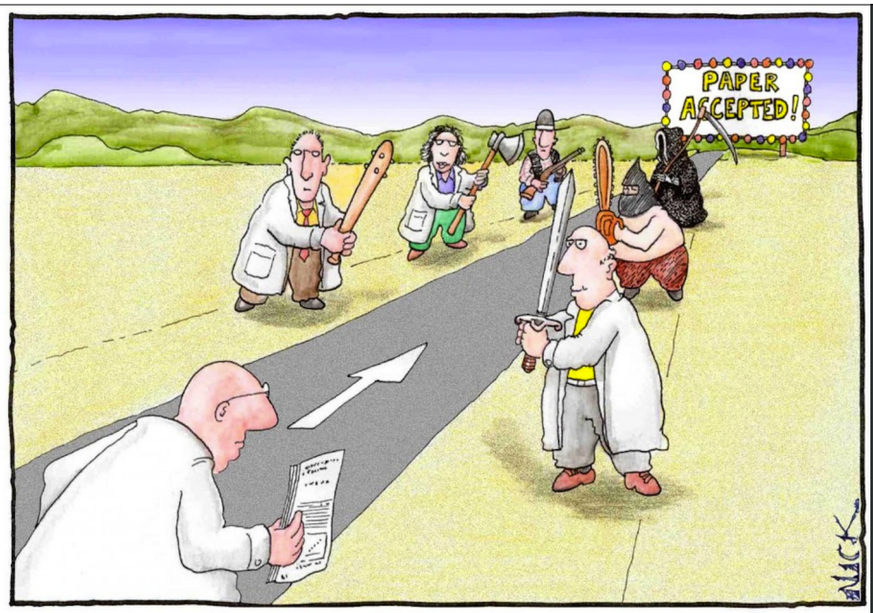
By way of a really quick introduction, my name is Garen Torikian and officially, I'm part of the Documentation team at GitHub.

That means working on stuff like Help guides, developer manuals, sometimes UI copy and blog material, as well as maintaining the tooling for all of that.

Documentation in general

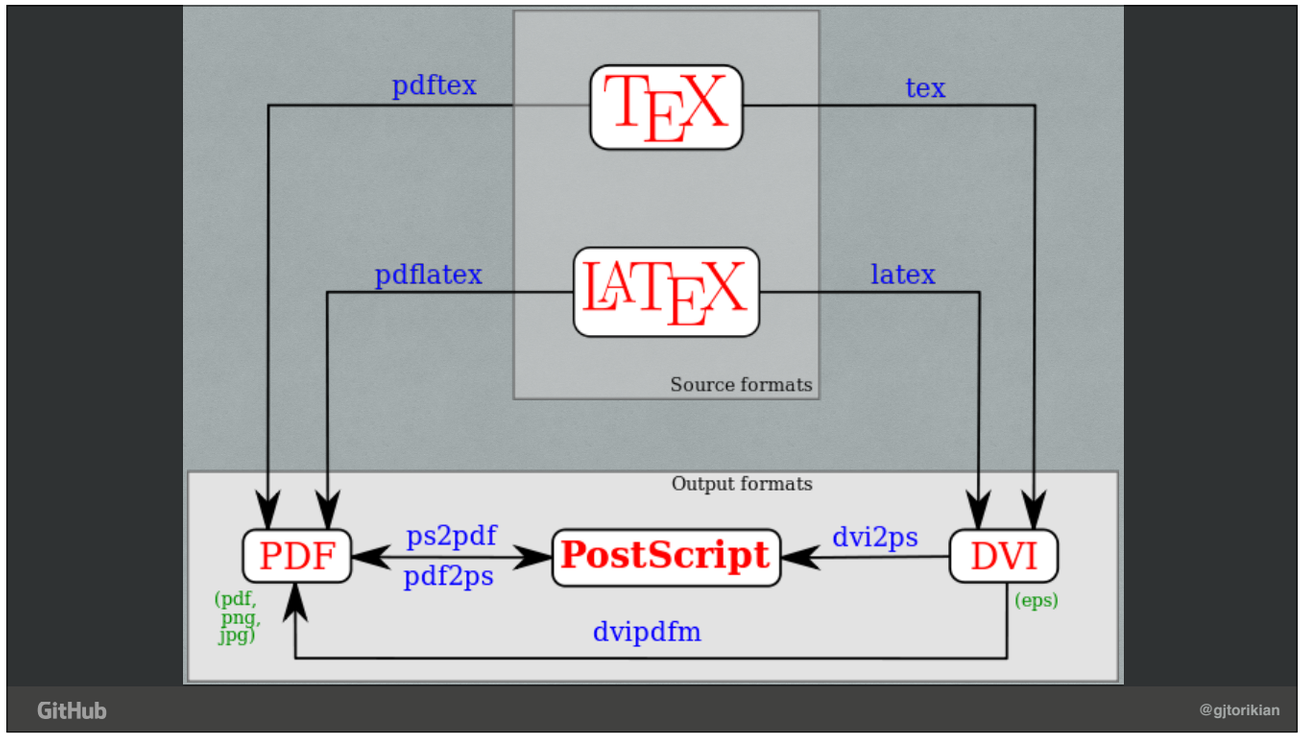
Unofficially, I've been interested in the way documentation is produced for a long time. I love documentation. I love writing it, and I love creating tools to make producing documentation even easier.

I've written something like three or four open source build tools in a variety of programming languages centered around making writing easier, for tech writers and non-techwriters alike.



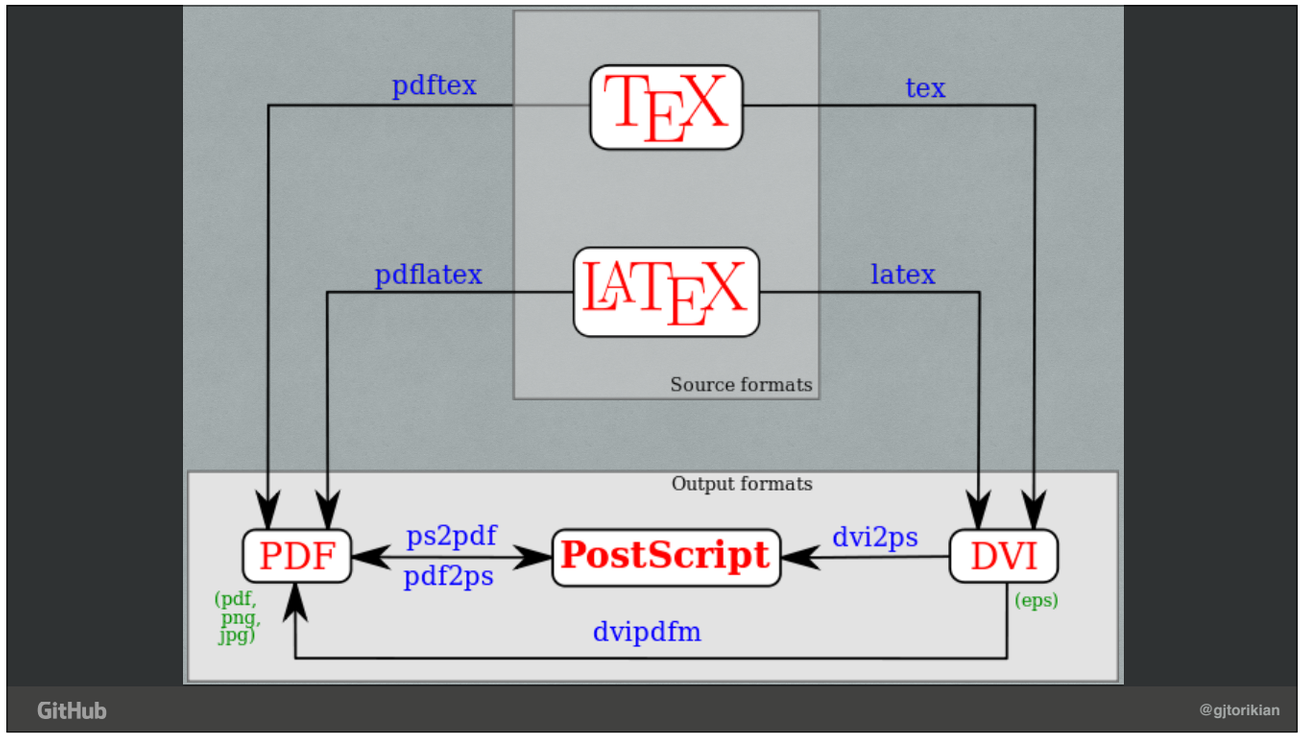
Most scientists regarded the new streamlined peer-review process as “quite an improvement.”

A little over a year ago, my colleague Arfon Smith approached me with a problem: that the state of producing scientific papers was problematic. For starters, many publishers expected Word or PDF submissions. These sorts of “locked” formats made collaboration between multiple authors difficult. In turn, this made representation of the content on GitHub extremely difficult.



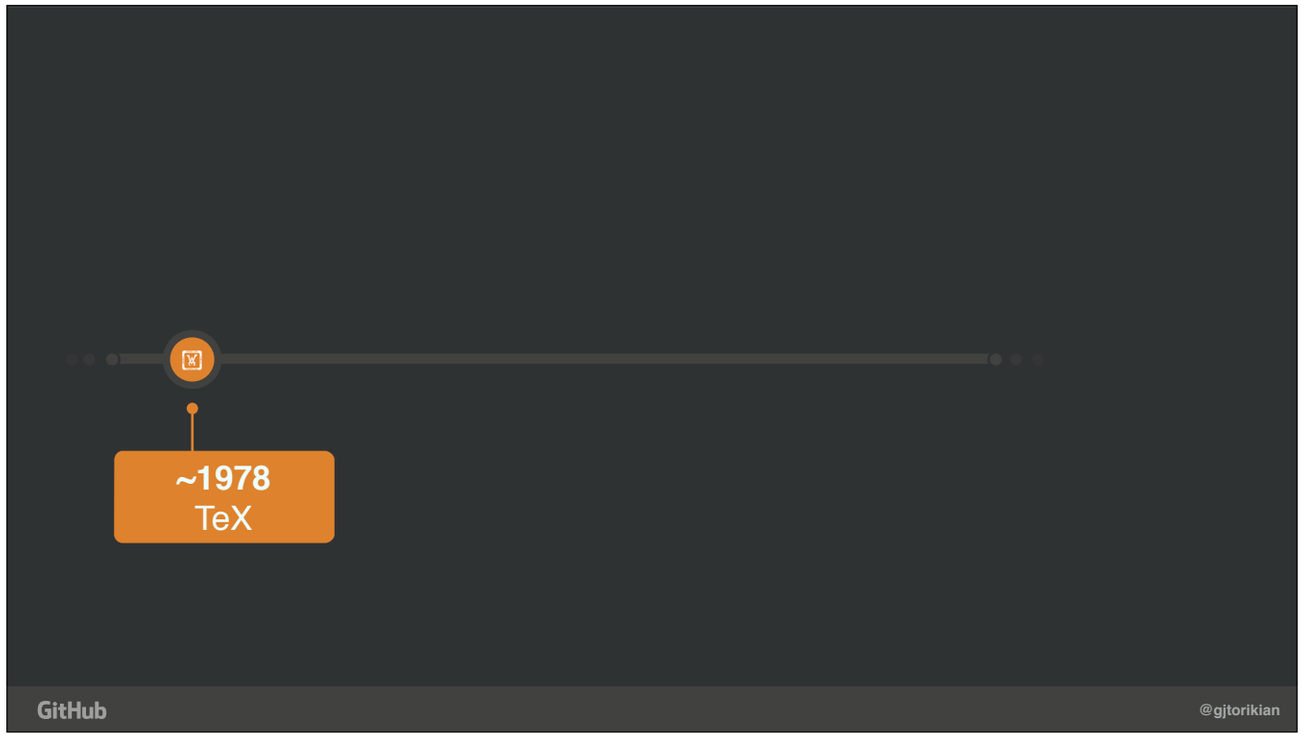
The other problem was that most universities encouraged papers to be written using tools devised several decades ago. While it's true that these tools are well-tested and therefore guaranteed to work, they require learning cumbersome techniques that no other technical profession required. One of the reasons these tools endured was because they could render mathematical equations with precision.

We decided that if we wanted to increase collaboration on GitHub, we would need to make the tooling more approachable for scientific writers in this day and age. Specifically, we needed to find a way to write and represent math in a much easier way.



I'm` a pretty stubborn person. I didn't want to believe that techniques that were devised in a pre-Internet era were still the best way to collaborate on writing papers. I wanted to help academics be able to write papers easier than they had been able to before. I started working on a hobby project to help solve the problem of writing math equations simply. Over the past eighteen months it's evolved from a crazy idea to one that works to one that's close to going into production.

Before introducing this project, I want to share with you the journey I went through in rediscovering the history of rendering math.

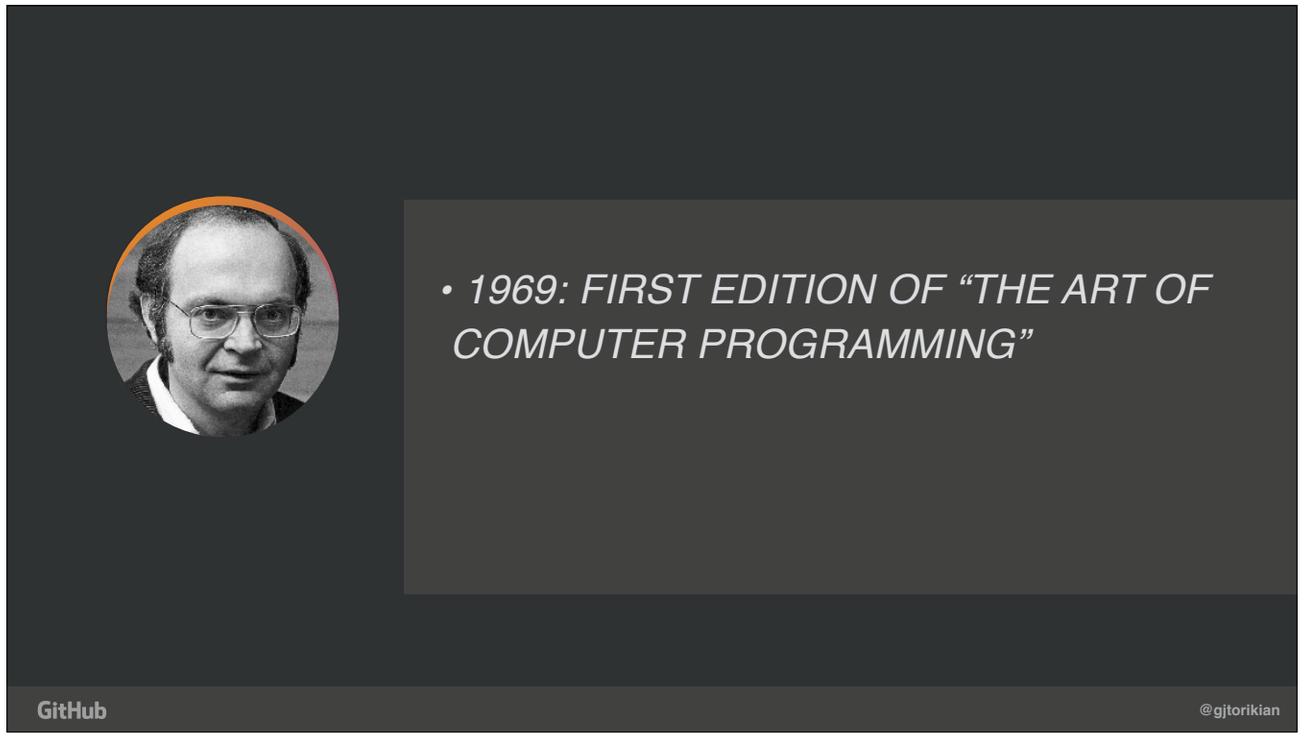


Let’s go back to the late 70s, to the time when digitally rendering mathematical equations was conceived.

One aspect that’s interesting to me about the history of rendering math is that it’s inexorably linked to the history of computer science as well.

- *FATHER OF “ALGORITHMS ANALYSIS”*
- *CREATOR OF LITERATE PROGRAMMING*
- *TURING AWARD WINNER*
- *WRITER OF MANY TOMES*

And that’s because of this man, Donald Knuth. In case you aren’t aware, Dr. Knuth is one of the early pioneers of the computer science discipline. I’m not just talking about older programming languages like LISP or FORTRAN or whatever. I mean the actual, theoretical underpinnings for many things we take for granted today, particularly around algorithms.



Dr. Knuth was also a fervent writer, and, in 1969, he published the first copy of his book, “The Art of Computer Programming”

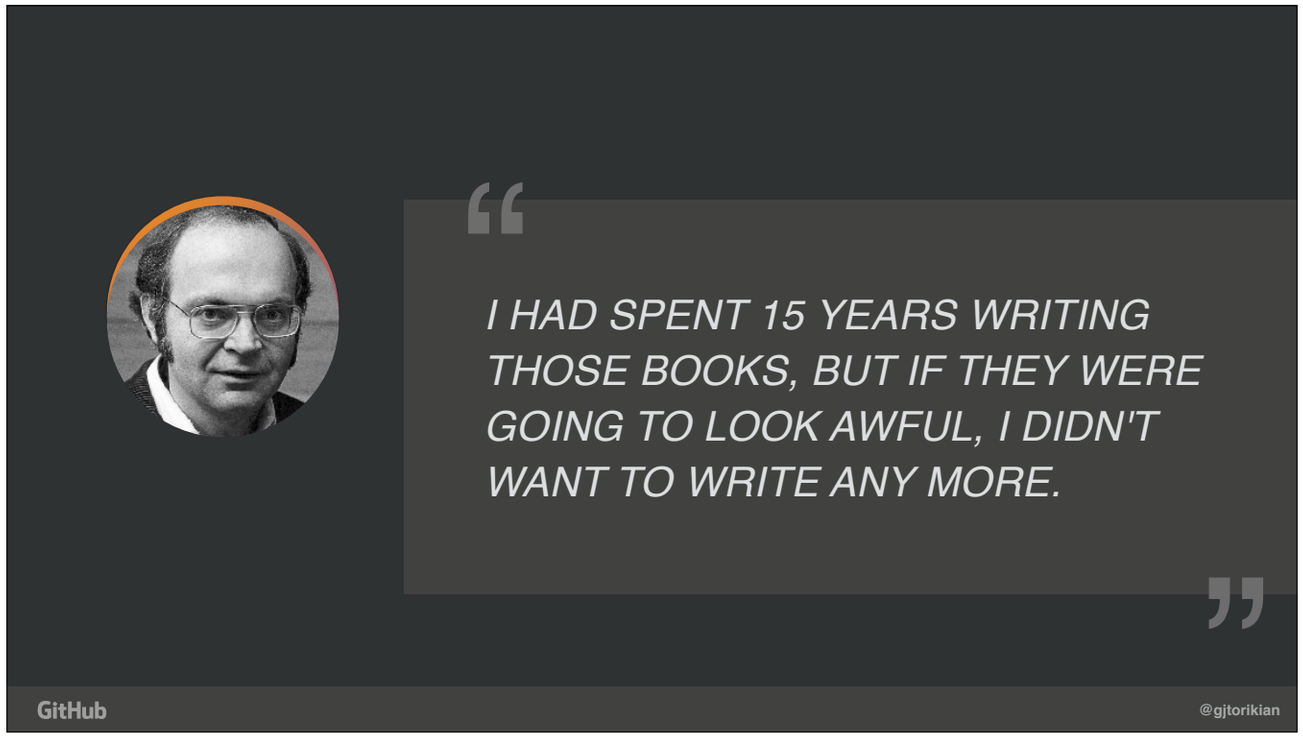
The first edition of the book was published using a technique called “hot metal typesetting.” Basically, you built individual glyphs snapped out of molten metal.

You can see in the photo here what that looks like, glyphs of all different shapes and sizes and fonts. You then douse these metal glyphs in ink, and press them firmly onto paper.

- *1969: FIRST EDITION OF “THE ART OF COMPUTER PROGRAMMING”*
- *1976: SECOND EDITION INTRODUCED*

Knuth adored the font used in the first edition of his book. Unfortunately, by the time the second edition was ready, the hot metal typesetting technique had been superseded, and the original font ceased to exist. In 1976, the second version of his book was slated to be published.

But after seven short years, newer photographic techniques were the vogue way to publishing books.



So, galley proofs are kind of like the “final draft” of a book before it gets published and distributed to readers. When Knuth retrieved the galley proofs of his second edition, he was extremely dissatisfied. He wrote: “I had spent 15 years….”

Aside from the fact that his font of choice had disappeared, many of his mathematical formulae appeared on the page as extremely sloppy and blurry.

- *1969: FIRST EDITION OF “THE ART OF COMPUTER PROGRAMMING”*
- *1976: SECOND EDITION INTRODUCED*
- *~1977: TEX IS CONCEIVED*

At that point, Knuth became fully interested in solving the problem of digital typesetting and developed a program called TeX. If anyone tells you it’s pronounced “Tex” as in Texas, they’re wrong. It’s pronounced TeX as in “technology.”

1. WRITE ONCE, PUBLISH THE SAME

TeX, at its core, is a typesetting system. There are three underlying goals to the program. One is that no matter how you write out TeX, it should render the exact same way across every format. TeX outputs an independent format that can then be converted to other formats such as PDF.

The second goal is that the paper's author should be able to control everything about the paper. Not just what font to use and what size to print at, but also the margins around a paper, the spacing between words, the line height of footnotes, etc.

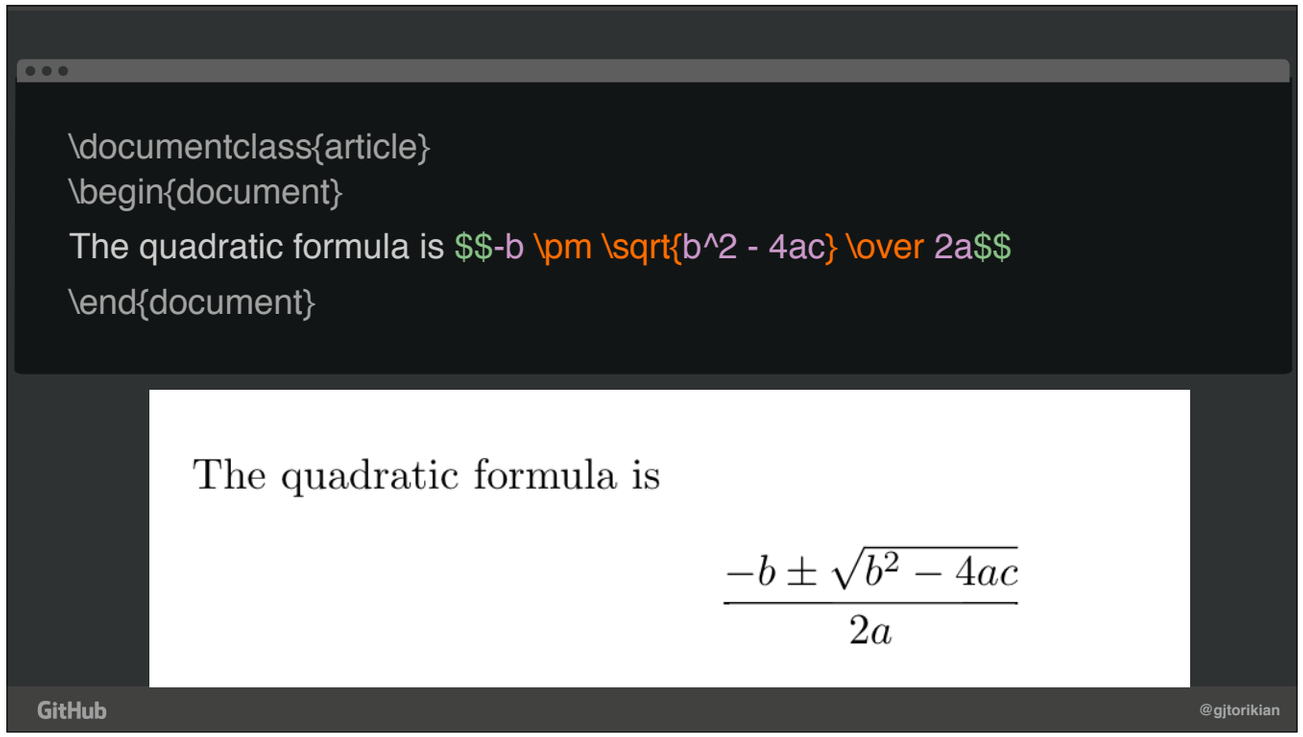
The output of TeX, at this point, is more like the output of a program, rather than a word processing document.

1. *WRITE ONCE, PUBLISH THE SAME*
2. *THE AUTHOR CONTROLS EVERYTHING*
3. *DESIGNED FOR COMPLEX MATH*

GitHub

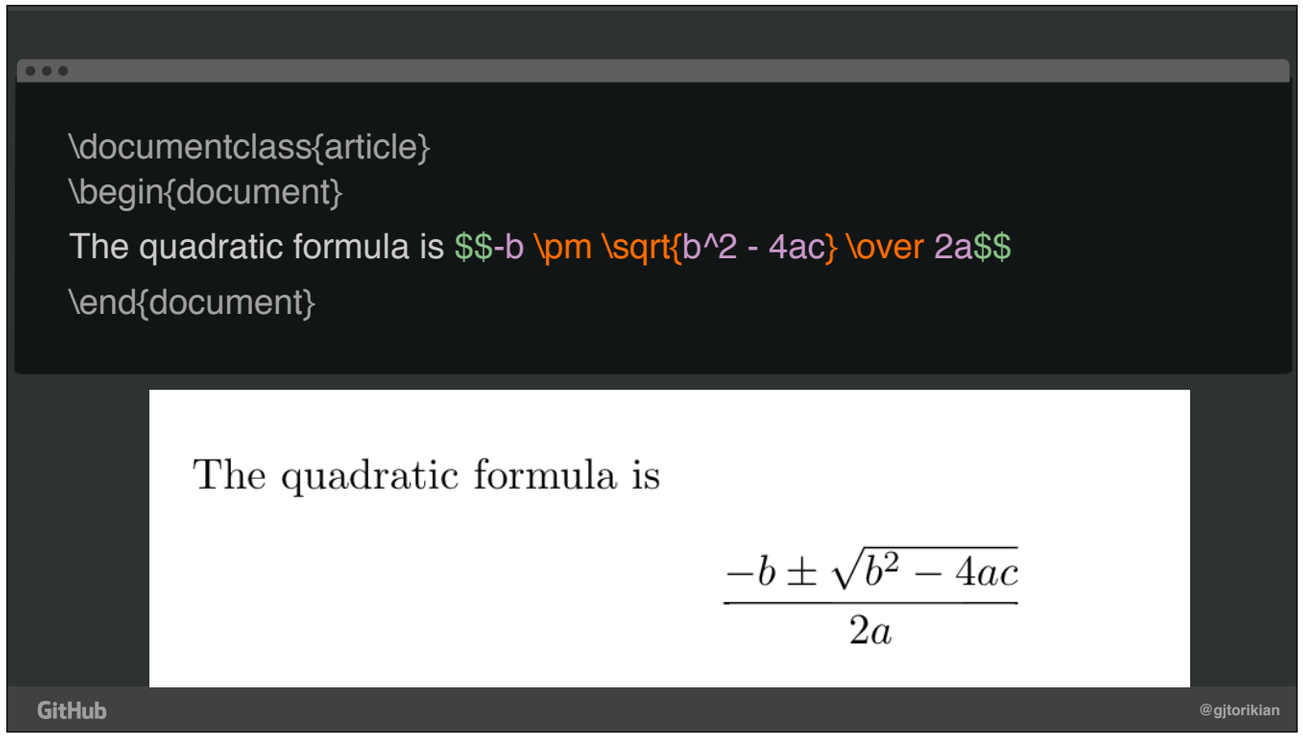
@gjtorikian

The final goal was the ability to typeset complex mathematical equations. This was the first time anyone had tried to digitally produce math.



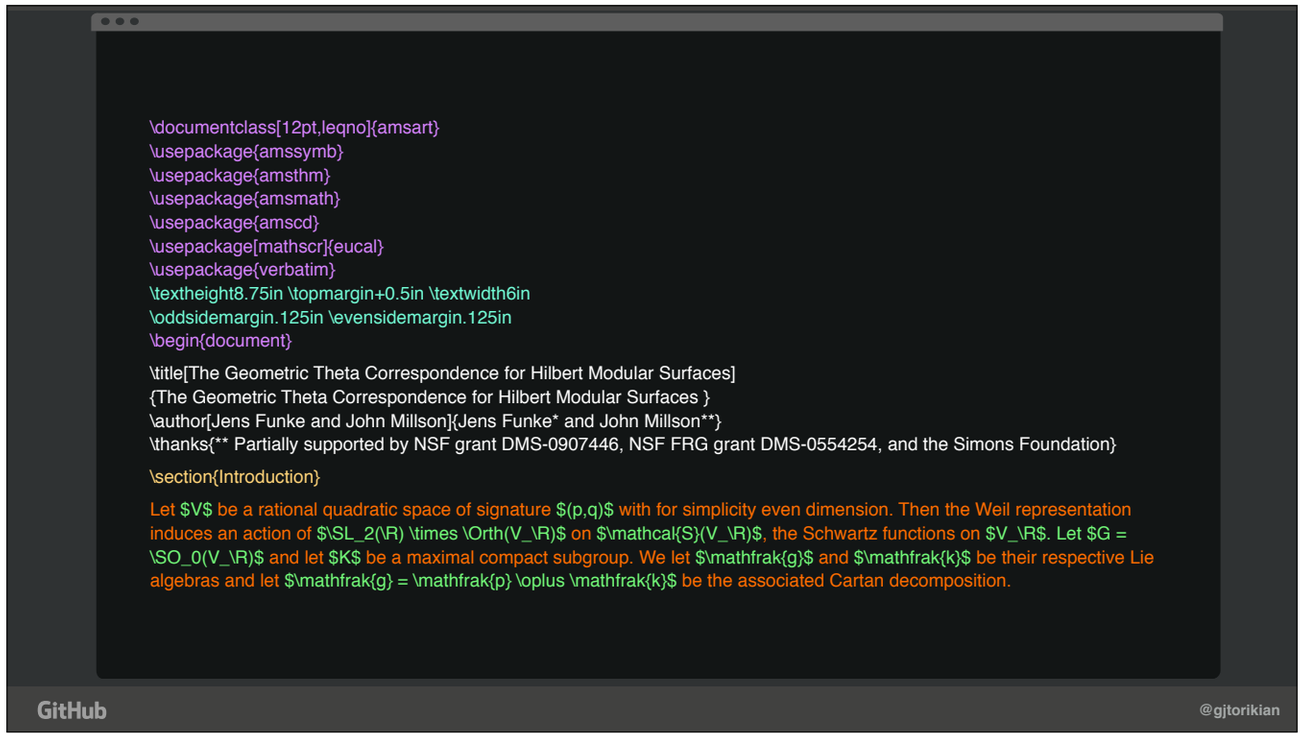
For the purposes of this talk, here's an extraordinarily simple representation of TeX's ability to render math. We're going to ignore all the tags at the beginning and end of this document for now and concentrate on the highlighted portions. The double dollar signs on the bookends demarcate the math. There are several operator keywords here, like \pm for "plus minus," \sqrt for "square root," and \over for fractions.

Knuth introduced keyword operators for every type of math function you can think of: integrals, exponents, Greek letters, glyph spacing, unions, and so on. Being an accomplished mathematician himself, his ASCII representation of the entirety of math is really an amazing thing to behold.



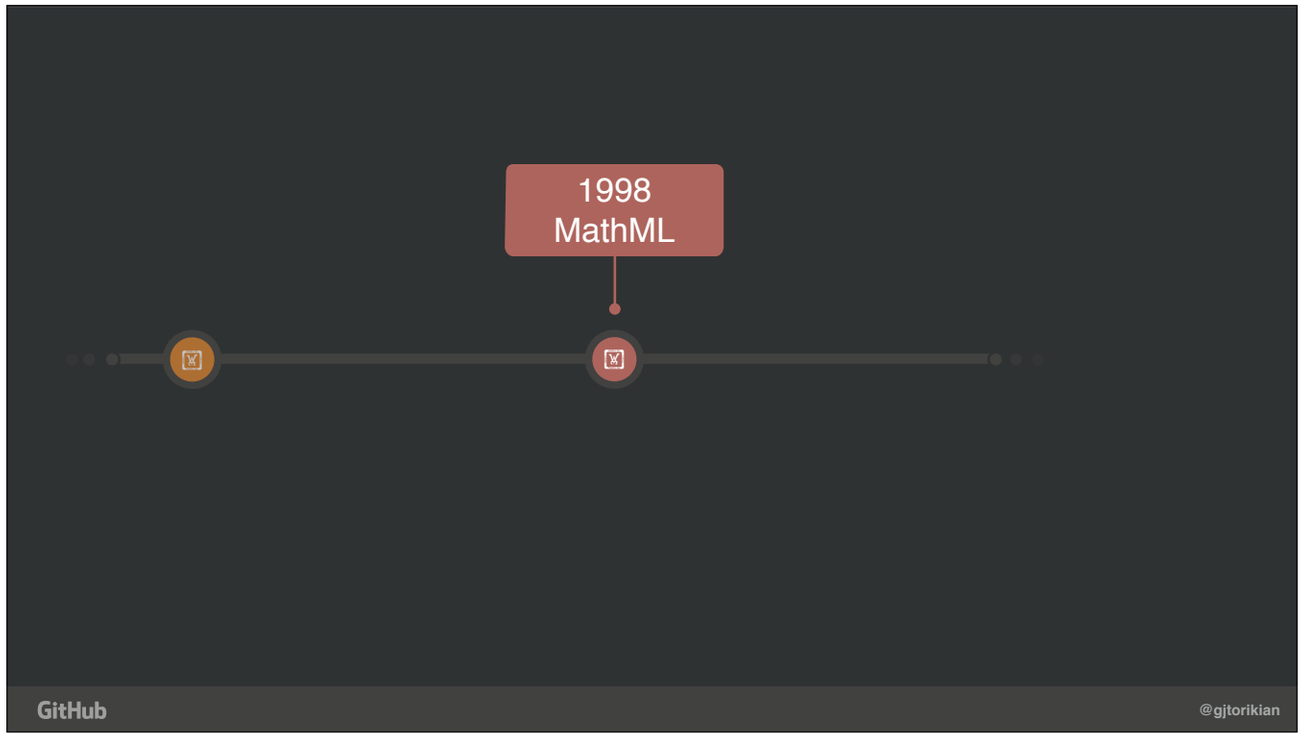
In just a few short years, TeX, and its later evolution, LaTeX, became *the* way to write academic and scientific papers. To this day, forty years later, you can still read, write, and generate TeX documents, as the program has been ported to newer operating systems over the years. This can't be said of most programs.

The greatest downside to LaTeX is that the learning curve is quite steep. There's an enormous amount of typesetting power that comes with the system, allowing you to manipulate every single speck and pixel that comes out. For some, that's a necessity. For most, it's overkill.

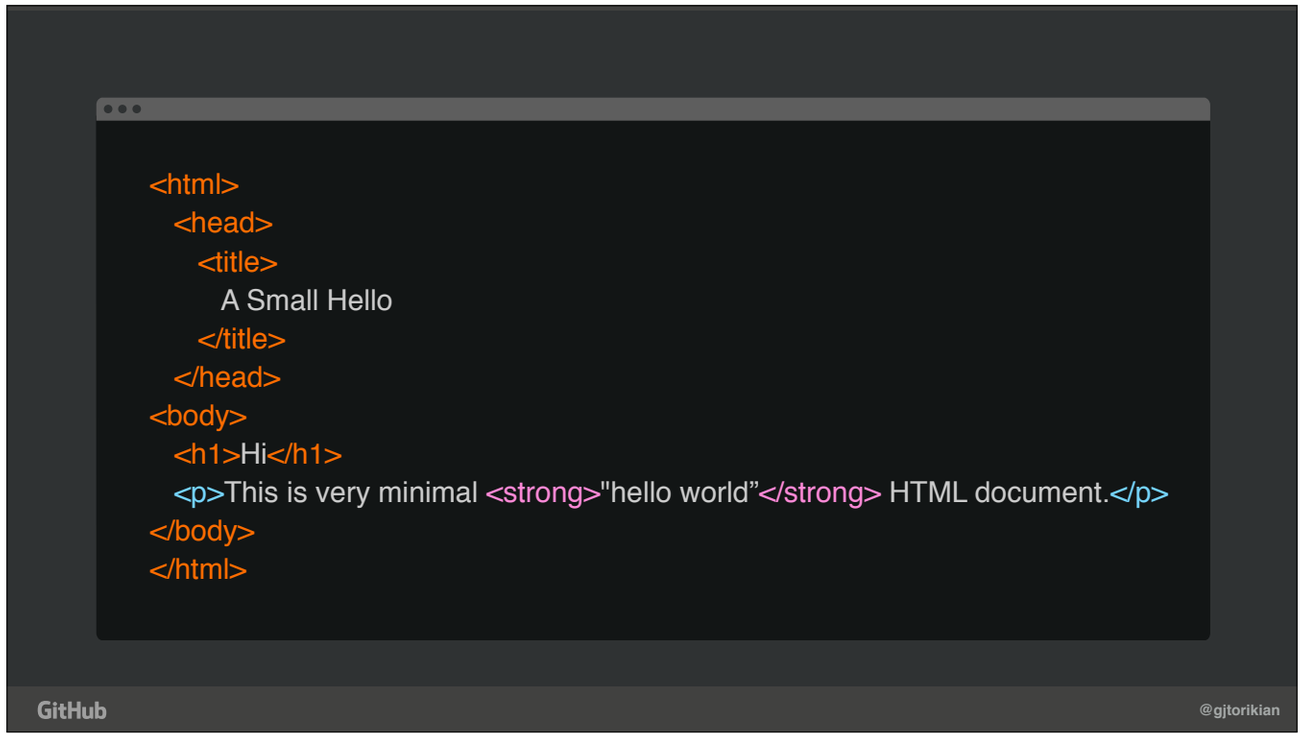


Here's what the start of a typical TeX paper might look like. It's some 200 lines of tags and keywords. We declare all the packages we want to use--that's the purple bit at the top. Then we define the margins, which is the blue sections. Then we define the metadata, which is in white. And finally, we start the actual content of the paper. This is a trivial example that only scratches the surface of all the keywords available to TeX.

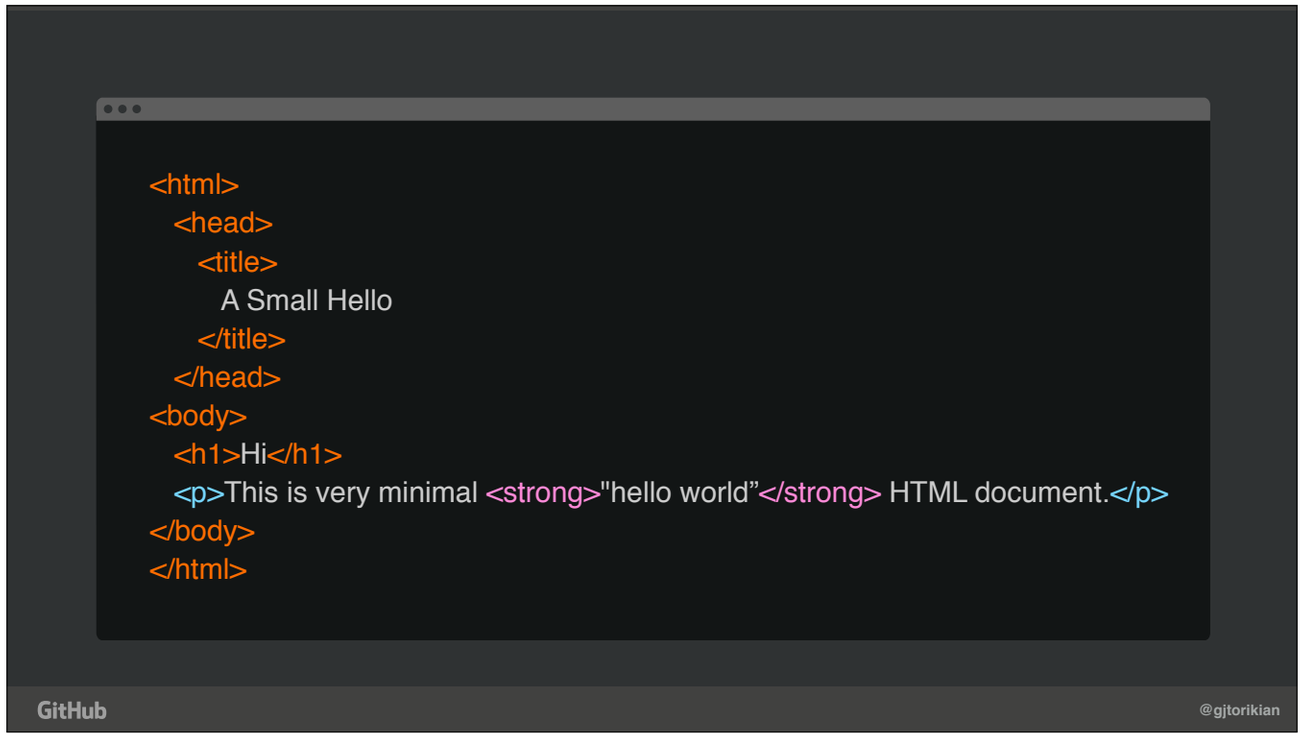
You have a great amount of power generating documents from TeX, if you're able to spare the months it would take to understand how to properly wield the language. TeX is essentially a programming language, and you're programming what you want your paper to be.



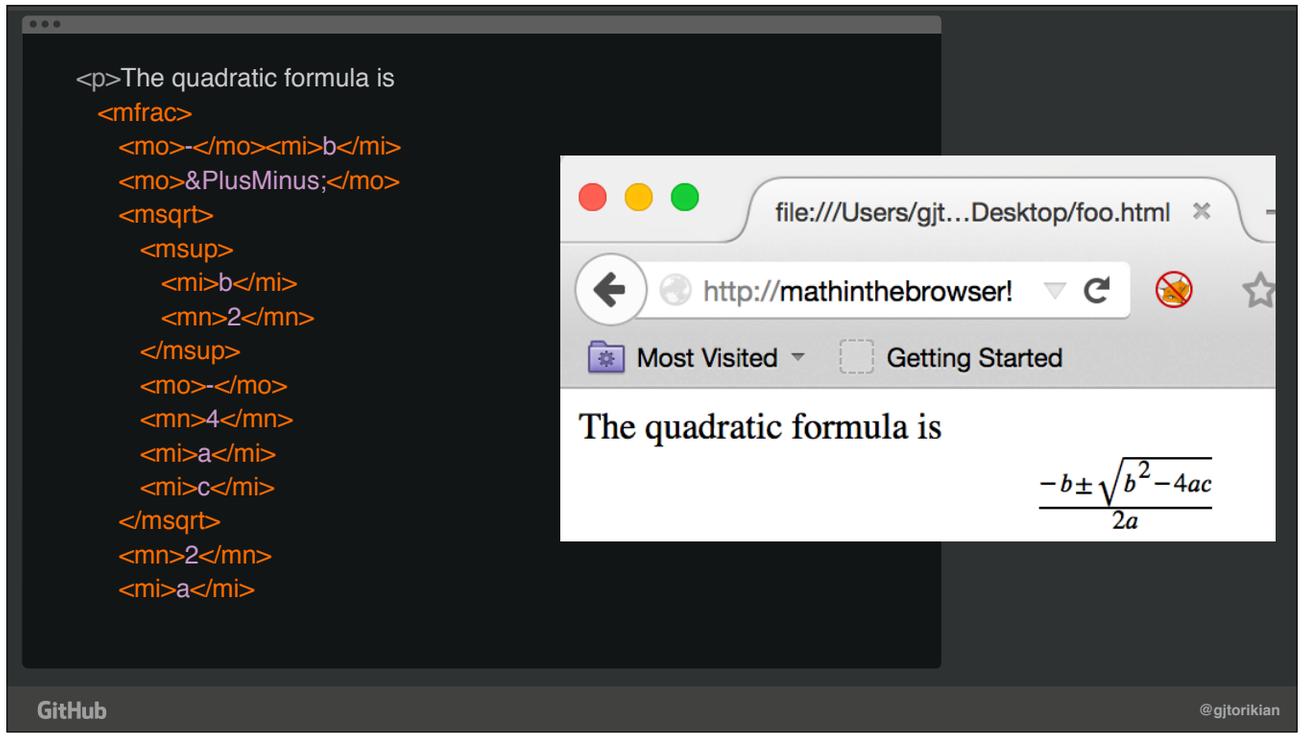
So, TeX for math rendering was the only game in town, and it was chugging around nicely, until around the mid-1990s with the maturation of the Internet. The Internet gave birth to a whole new environment, and entirely new advancements for humans to communicate with each other.



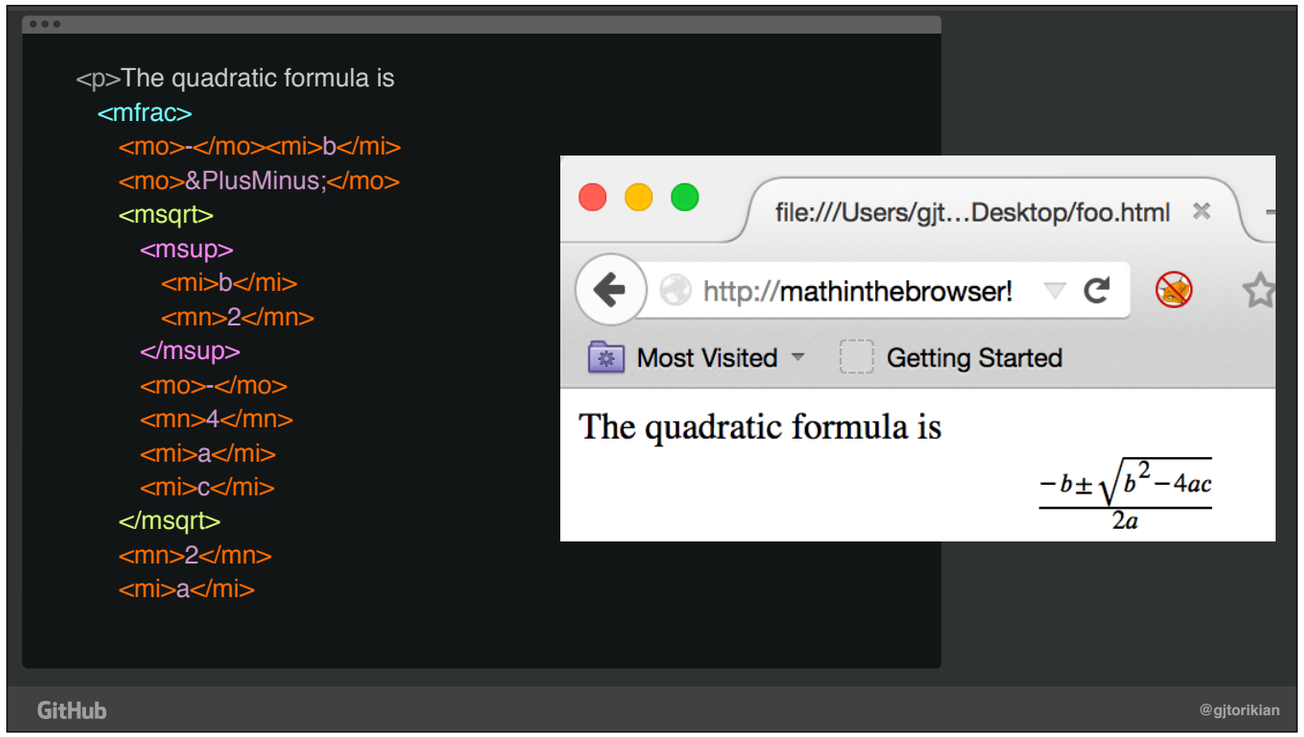
The main format of transmitting information was and is HTML. With a few semantic tags, you could express yourself to anyone in the world. For example, <p> meant paragraph, and meant to bold text. You could build and format entire structures of information and typography with these tags.



Naturally, the math nerds were extremely interested in this evolution. Already by the mid-90s, although still extremely popular, the opaqueness of TeX was taking its toll on writers. This sort of HTML markup introduced a new way of writing and sending text. So, it was decided that mathematics also needed a way of representing itself on the Internet.



They devised a system called the “math markup language,” or MathML. HTML was officially adopted by a standards body around 1992. MathML followed shortly thereafter, with version 1.0 released in 1999, and the most common implementation, version 2.0, adopted in 2001.

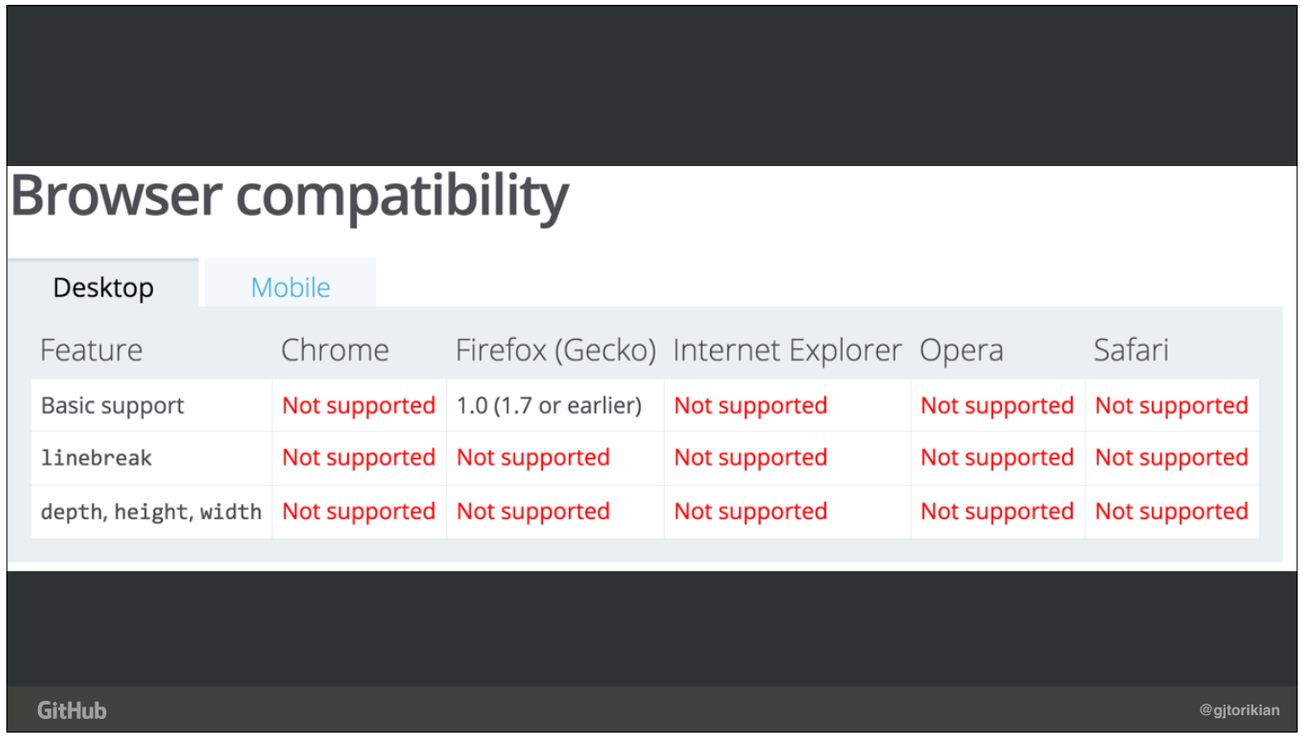


MathML took a lot of inspiration from HTML. It, too, followed the semantic tag format that was popular at the time. You can see some of the tags here, and how they're used in this: mfrac for "fraction," msqrt for "square root," msup for "superscript" Like TeX, MathML was devised with a solid understanding of all the renderable mathematics in the world. With HTML and MathML, the web gave academics a new way to write and distribute their papers.

If MathML were widely accepted as *the* dominant way to render math for the web, the story would end right here. But there's a twist.

In the 15 years since it was standardized, only one of the major browsers best supports rendering MathML.

And that’s Firefox.



The real tragedy is that even Firefox doesn't support *all* of the MathML specification. There are still a large number of missing tags and attributes. Here's a screenshot from the official FIrefox documentation, where you can see a huge gap in supporting specific attributes.

The world has a standardized way of rendering math online, yet no browser will fully support it.

So what exactly happened to MathML? Why didn't it gain more adoption?

*** YOU SHOULD BE AT FIFTEEN MINUTES!!! ***

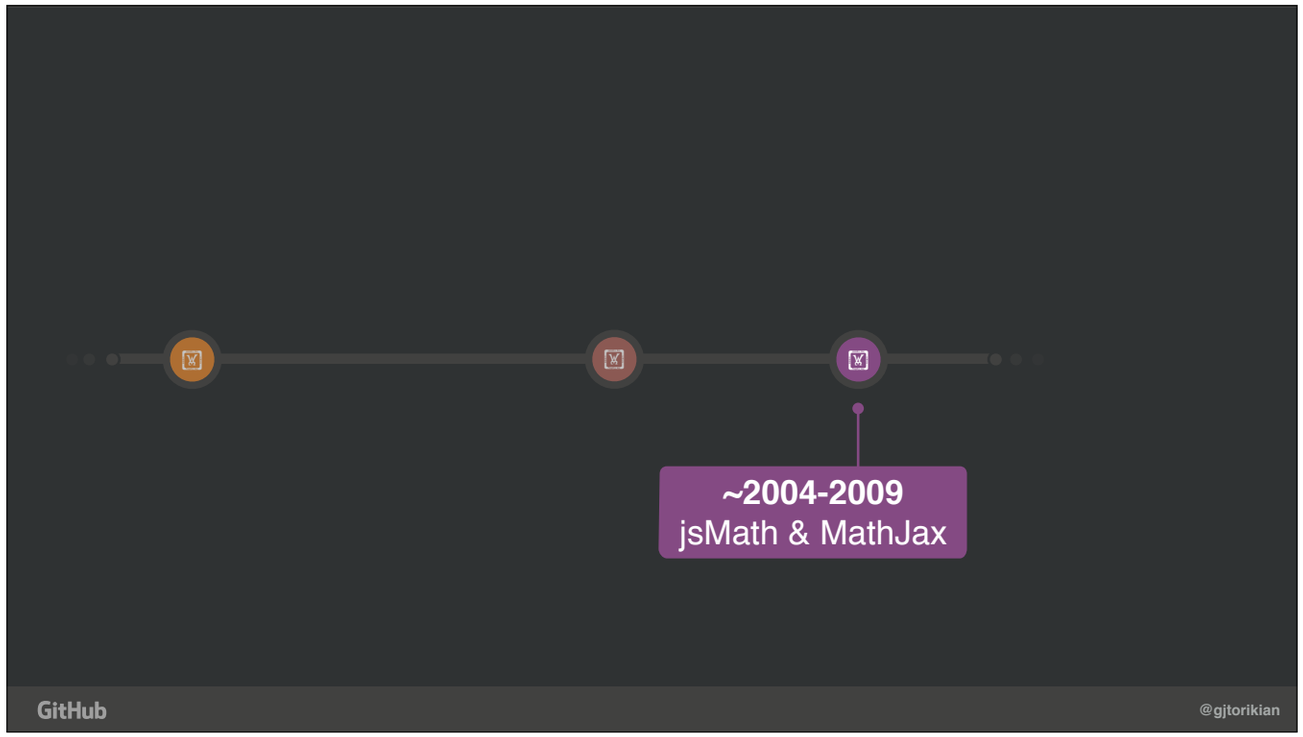
I think the main reason for this is probably well known: humans hate writing with tags. HTML and XML brought to the masses a human-readable, machine-parseable format for communication, and that system of writing web pages worked…for about a decade.

Soon, the verbosity of having to open and close tags, and the variance in support amongst browsers for those tags, exhausted people writing online. People looked for ways to make writing for the internet simpler.

The truth is, since the beginning of the aughts, writing raw HTML has been declining in favor of simpler markup languages. Imagine this: MathML 2.0 was ratified in 2001. In 2004, Markdown was introduced. And even by that time, Wikipedia's wiki format had already introduced thousand of writers to a way of abstracting HTML in favor of plaintext writing. When you edit wikipedia, you edit plaintext, not HTML. And since then, a bevy of new formats and templates in a variety of languages have exploded.

By the time MathML had been standardized, people were becoming exhausted writing HTML. Markup languages were designed to be written by humans and converted into HTML.

For writers, the advantages of a simpler markup system are huge. You didn't need to learn an entire set of cumbersome tags—you could simply apply a few symbols around regular words and have the browser pick it up as HTML.



In some ways, MathML didn’t really have a chance. People didn’t want to write web pages OR math using HTML. For math rendering, newer client-side tools began to form.

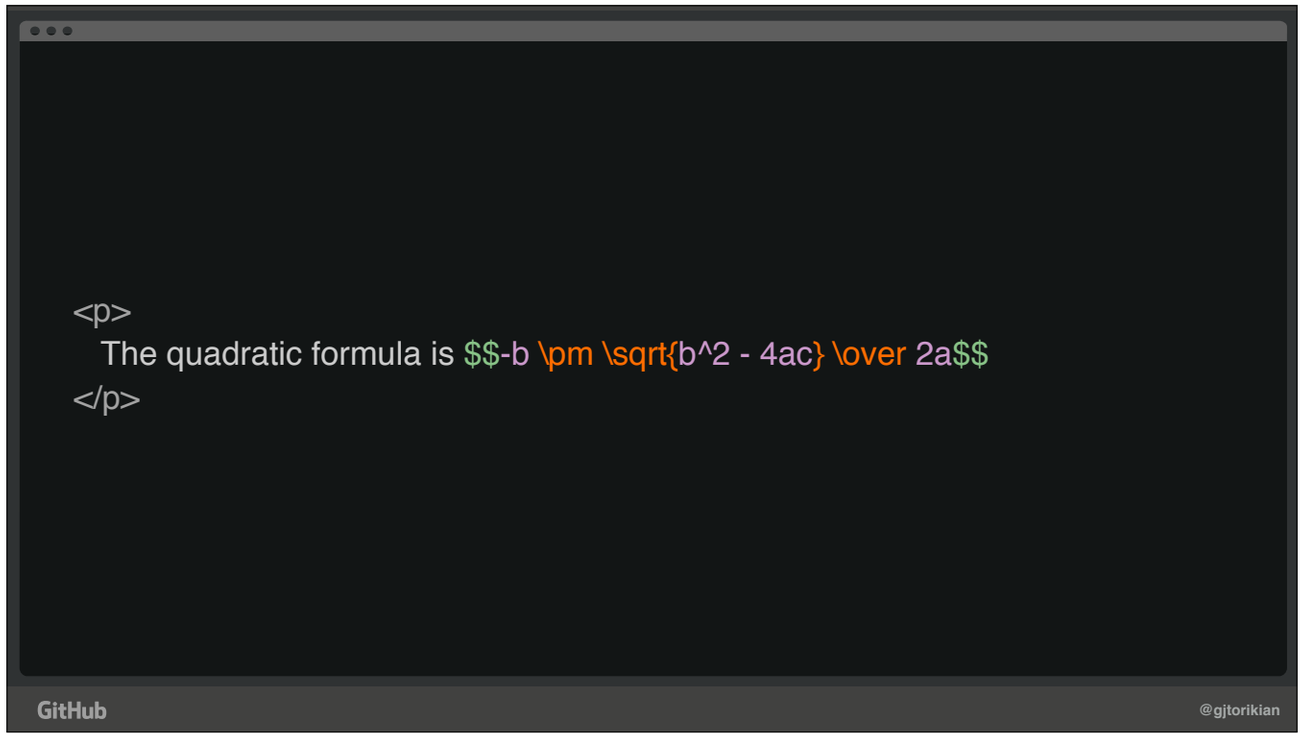
jsMath was a project that parsed a website in JavaScript and rendered mathematical equations into images for better visualization. The continuation of these efforts resulted in MathJax, which is now the most widely used, de facto way to write math online.

What does using client-side MathJax solution look like?

```
<p>The quadratic formula is
  <mfrac>
    <mo>-</mo><mi>b</mi>
    <mo>&PlusMinus;</mo>
    <msqrt>
      <msup>
        <mi>b</mi>
        <mn>2</mn>
      </msup>
      <mo>-</mo>
      <mn>4</mn>
      <mi>a</mi>
      <mi>c</mi>
    </msqrt>
    <mn>2</mn>
    <mi>a</mi>
```

GitHub @gjtorikian

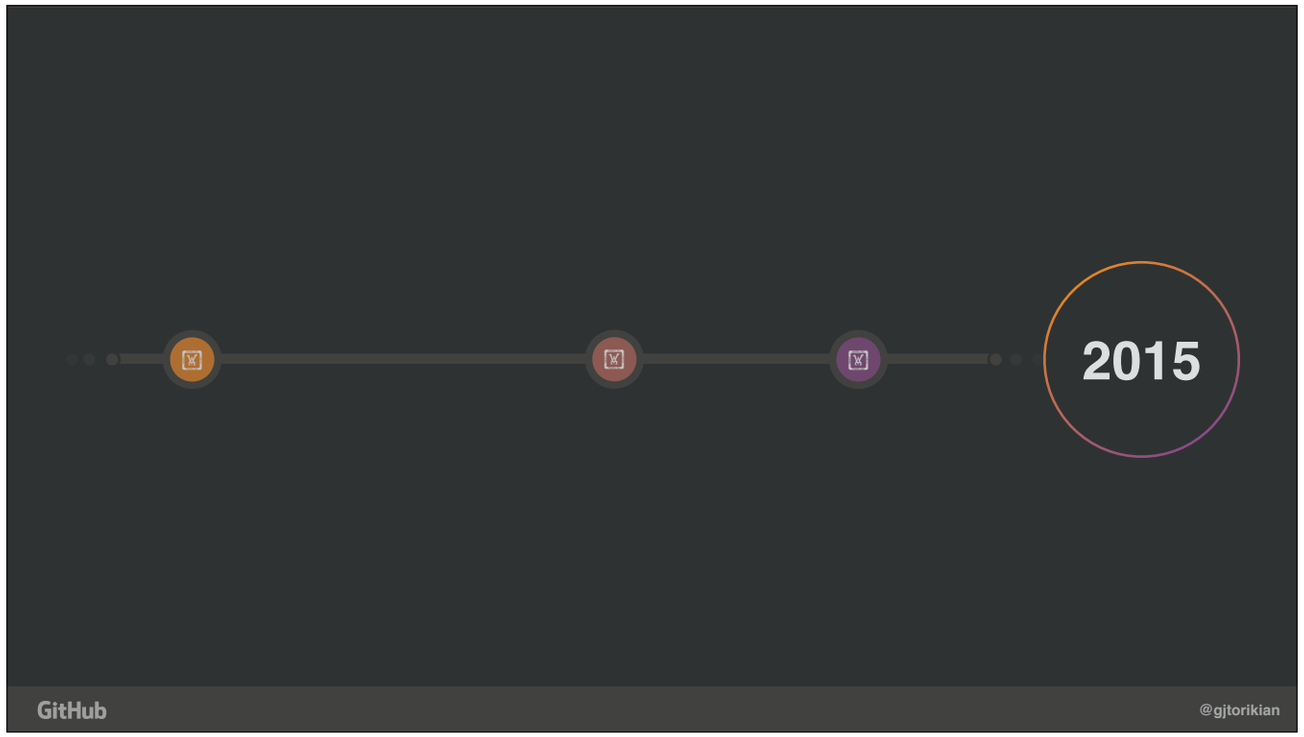
Instead of writing verbose HTML pages containing math like this…



People are writing them like this. That's right, we've come full circle. What's old is new again. TeX math equations are interspersed into HTML documents. MathJax parses web pages, detects TeX like markup, and replaces TeX with images, all within the browser. In the 40 years since TeX was introduced, for online communication, the academic community at large *still* prefers writing with the TeX ASCII equations.

The great advancement is that these days, we can take all the brilliant simplification of writing math equations, and drop all of the other TeX document formatting stuff. We can use standard web tools like HTML and CSS to control margins and font sizes. This simplifies the way we write scientific papers by latching on to other readily available Internet technologies.

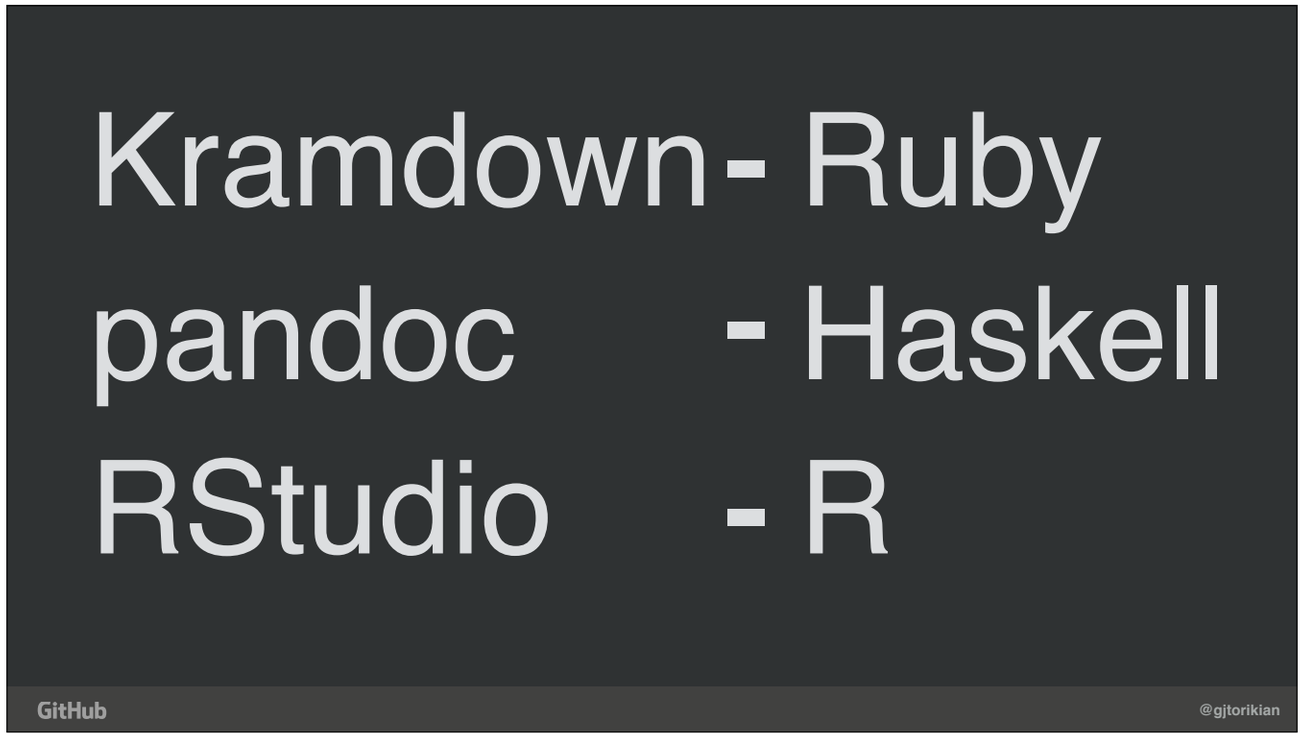
With dwindling browser support, MathML may, unfortunately, never experience the popularity it deserves.



So now it’s 2015. What’s one of the newer changes from the last decade to now?

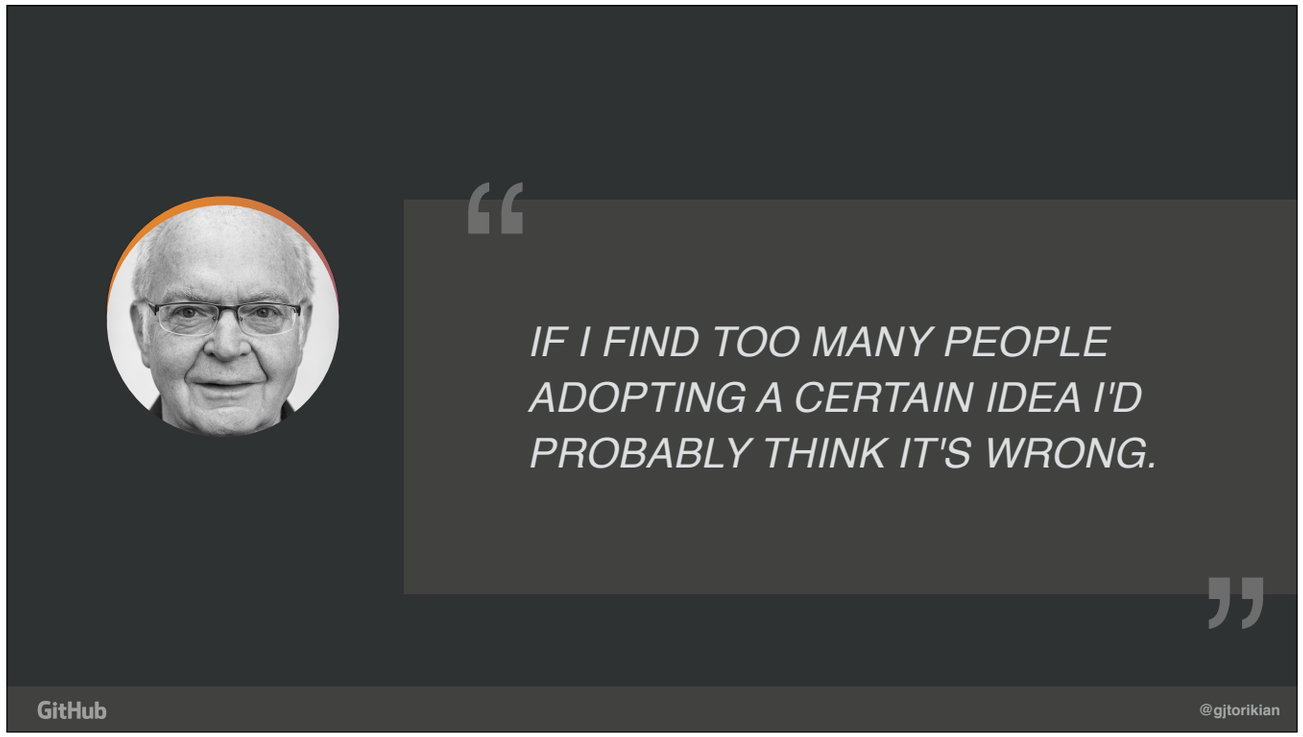
JavaScripts is frikkin' everywhere. It's in the browser. It's on the server. You can read and write binary data with it. You can built entire applications out of it, even on the desktop. And of course, people are using tools like MathJax to render math with it.

But there's a problem.



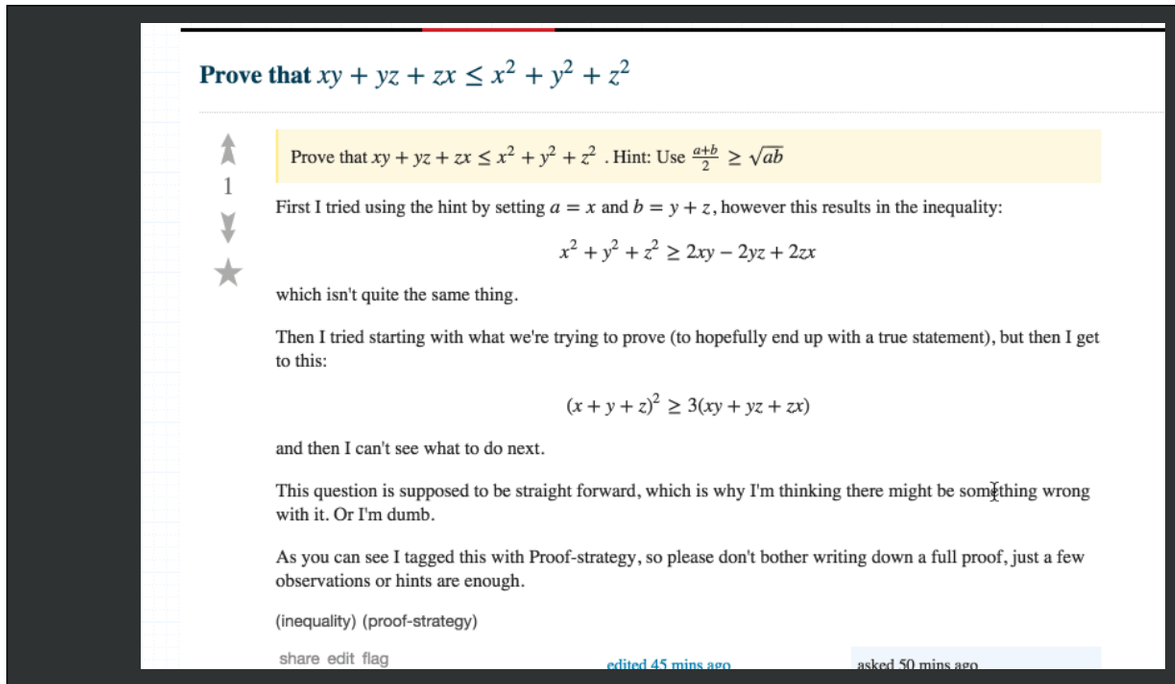
MathJax is, for all intents and purposes, perfect, as a browser-based, client-side rendering solution. The problem is that many offline build tools are using it in conjunction with other languages, in lieu of anything better. For example, Kramdown, a Markdown converter written in Ruby, calls out to MathJax, in JavaScript, to render math equations. In fact, nearly every single program I found that's used to render documents calls out to the MathJax JavaScript library to perform math rendering.

We've stopped using JavaScript solely on the client-side, and have begun interspersing it with other languages. This causes horrid performance. Expecting MathJax to render documents on your local machine takes far longer than it should. I'll show a demo of this later on.



Everyone uses MathJax. Stackoverflow uses MathJax. Hundreds of personal blogs use it. Dozens of offline rendering tools use it.

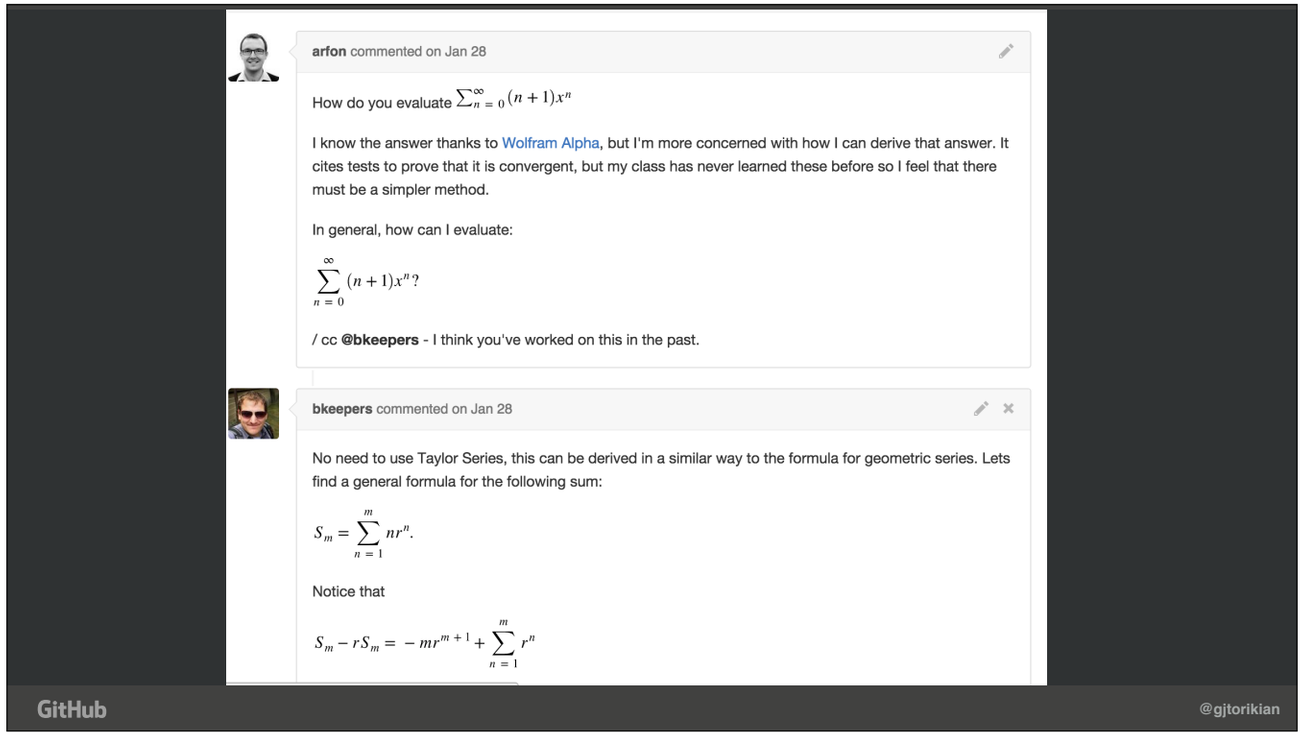
I consider this a monopoly. And in turn, I’m reminded of another Knuth quote that can describe this situation (read quote)



This is what clientside JavaScript rendering looks like. There are no tricks in this GIF, this is my first visit to a page on my home's wireless internet.

After the page fully loads, you can see the stutter, as the math is interpreted and redrawn. Redrawing a page causes text to reflow, as the browser tries to put everything back in its proper place. This is a small example on StackOverflow, but imagine a large academic document containing hundreds of equations.

At GitHub, we tried to implement the JavaScript based approach to rendering math. We didn't enjoy the experience of reshuffling text. We also found that integrating JavaScript serverside into our Ruby processes was a bit of a pain.



For example, our comments on the site are processed through Ruby. If we wanted to also process math using MathJax, we'd need to jump through several hoops to get it to work, and even then, performance would be bad.

We sought out a way to process the Markdown and the math serverside, so that we could render a page all at once.

To break up this Javascript monopoly, I'd like to introduce the library I've been working on called mtex2MML. That stands for "math tex to Math ML."

- Flex / Bison parser

It’s a parser written in the usual Flex / Bison style. It’s written in pure C, with no other dependencies. It also understands and accepts the entirety of the TeX keyword list.

- Flex / Bison parser
- ~93% like MathJax

mtex2MML is 93% compatible with MathJax. What that means is that it understands almost everything that MathJax does. In fact, I took the entire MathJax test suite and used it as a basis for testing mtex2MML.

- Flex / Bison parser
- ~93% like MathJax
- GPL/MPL/LGPL

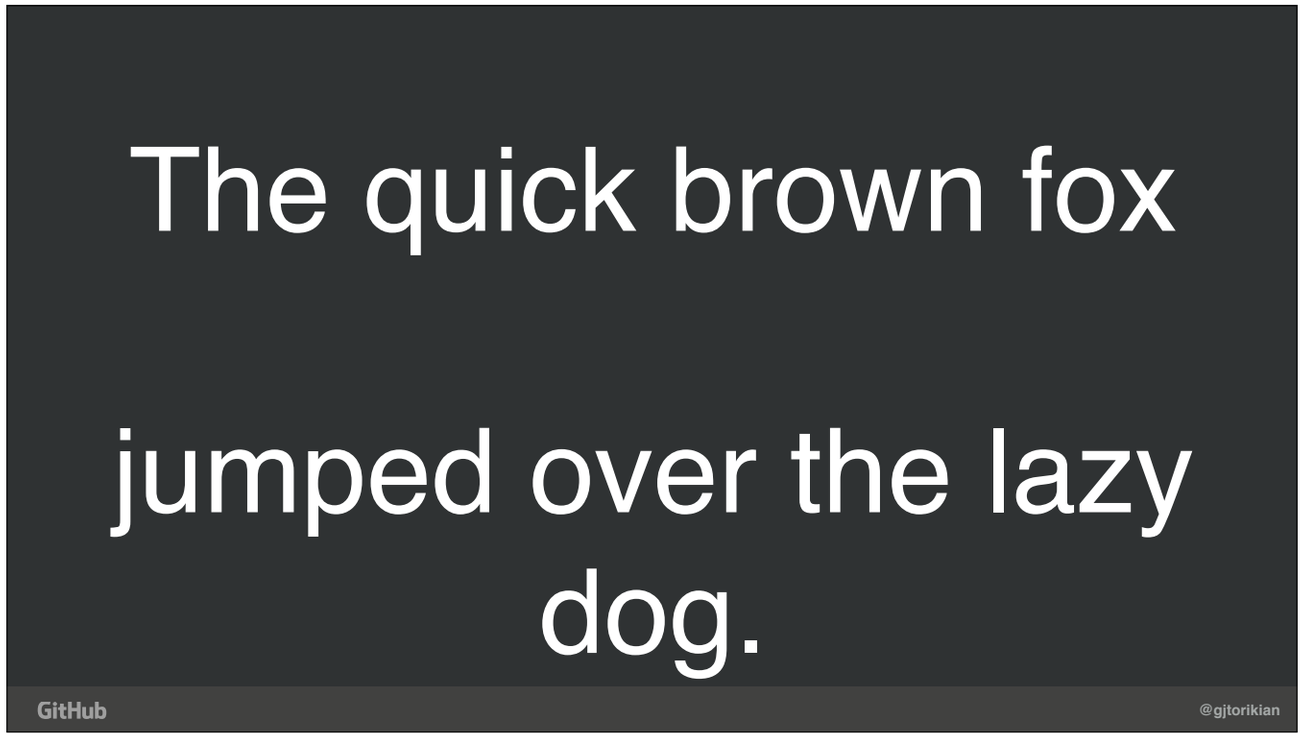
The project is triple-licensed under the GPL, MPL, and LGPL licenses.

- Flex / Bison parser
- ~93% like MathJax
- GPL/MPL/LGPL
- Cross-platform

Probably best of all, the library is cross-platform. It compiles and runs successfully under Mac OS X, Linux, and Windows.

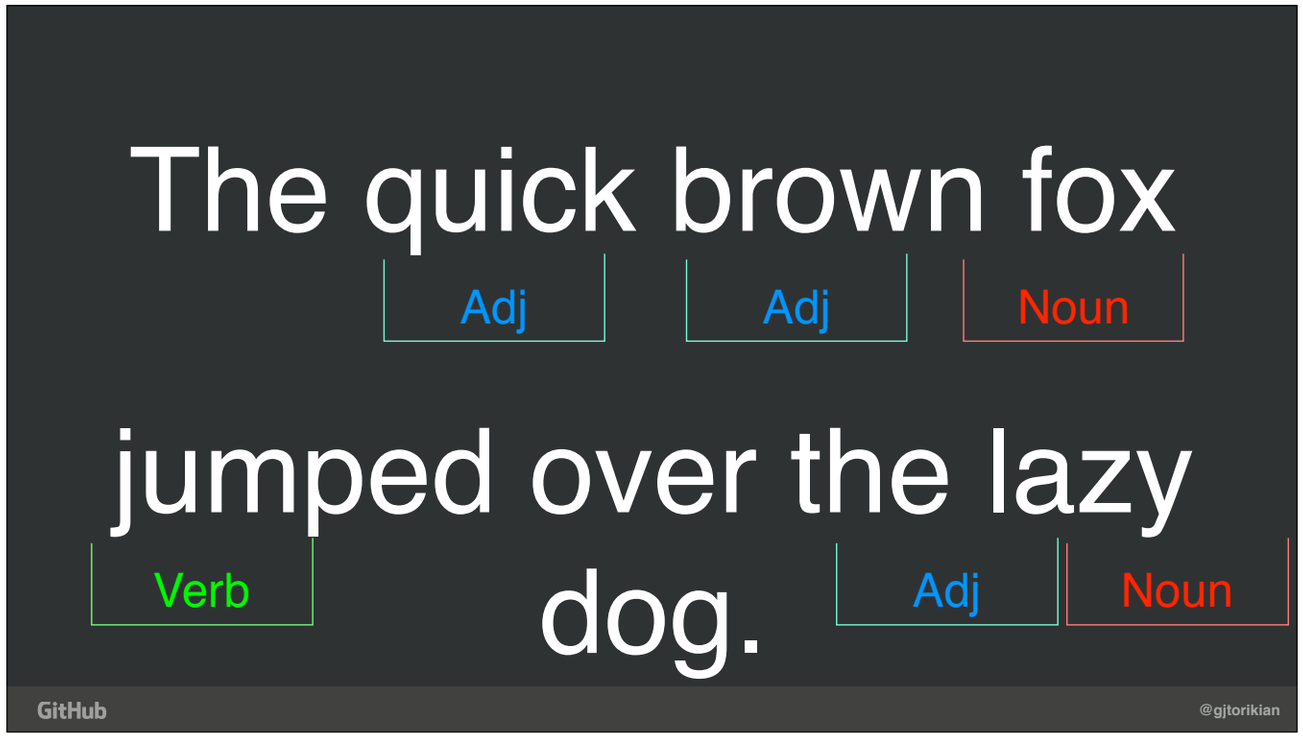
- Flex / Bison parser
- ~93% like MathJax
- GPL/MPL/LGPL
- Cross-platform

I want to talk really quickly about the decision to use a Bison grammar. What it means, and why it’s great.



In order to understand how grammars work, let’s take a look at the following typical English sentence. Most grammars for programming languages read the same way as you would read english: left to right.

“The quick brown fox jumped over the lazy dog”



~ 23 minutes?

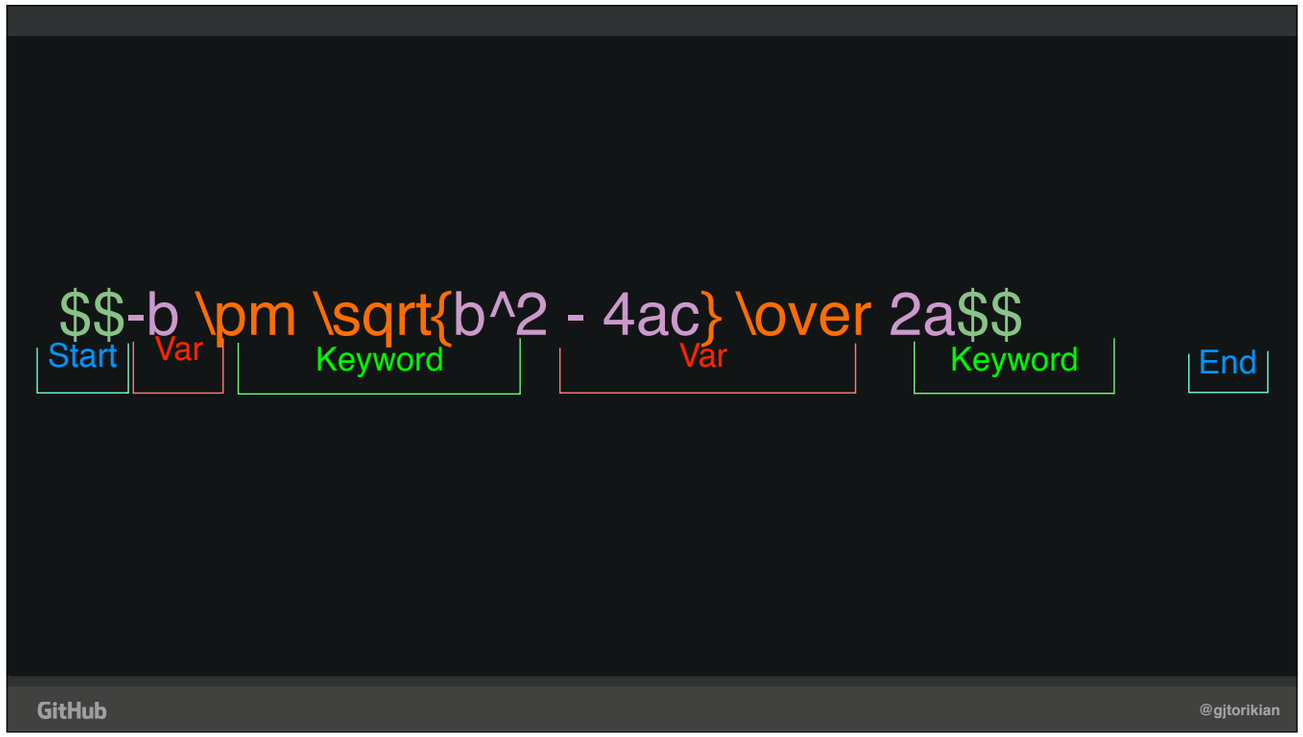
What a parser does is two things. First, it tokenizes a sentence into bits. If we tokenized this sentence into Adjectives, Nouns, and Verbs, we'd get a result that looks something like this.

Tokenization is little more than defining a bunch of keywords and indicating that you want to perform an action once a keyword is found.

After tokenization comes the actual parsing step. The parser can be used to represent the tokens into actions.

Basically, after a certain sequence of words is matched, you can go ahead and perform an action. The arrangement and placement of the words is extremely important.

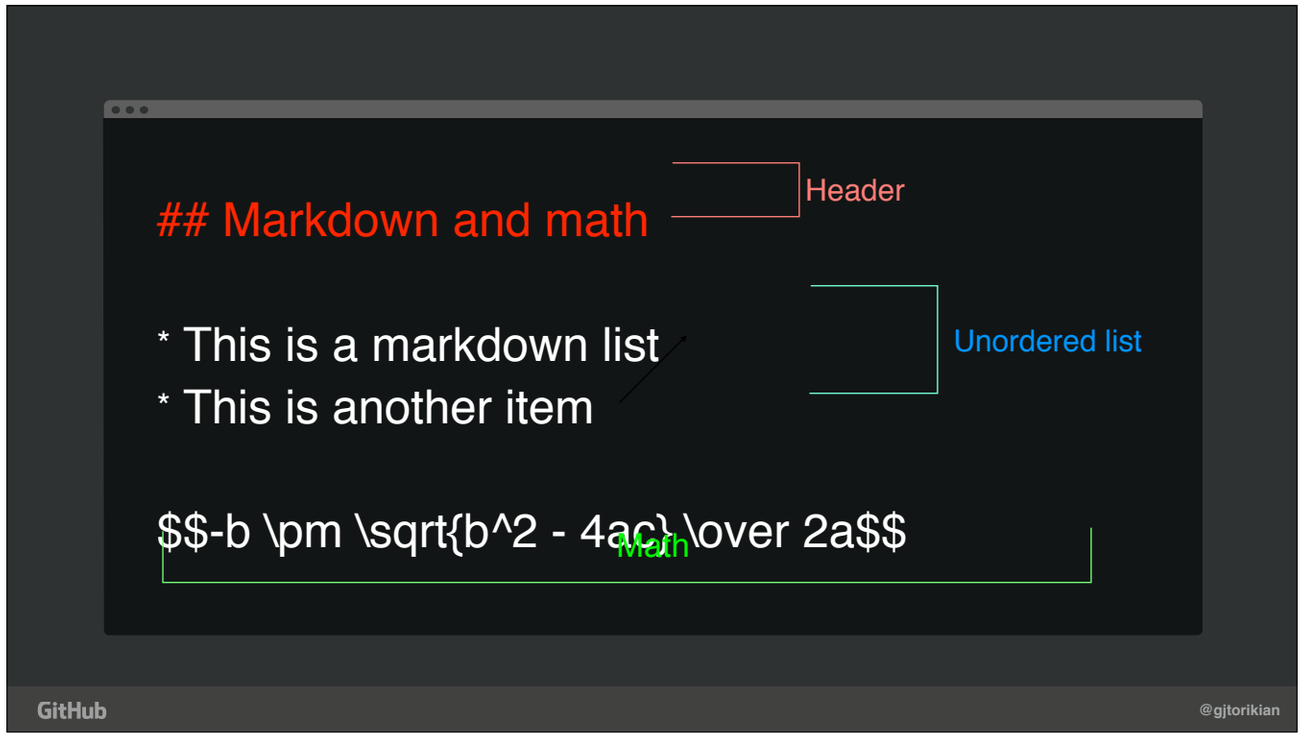
If a token sequence is found that the parser doesn’t know how to handle, it’s simply ignored. This makes it extremely safe when processing unknown user input, because you don’t need to trust that the text you’re processing is perfect.



To put it into our specific context, a grammar would look at an equation and break it down into something like this: it would know when to start parsing math, when to stop parsing math, and it would know which bits of the path are TeX keywords, and which bits are variables to draw.

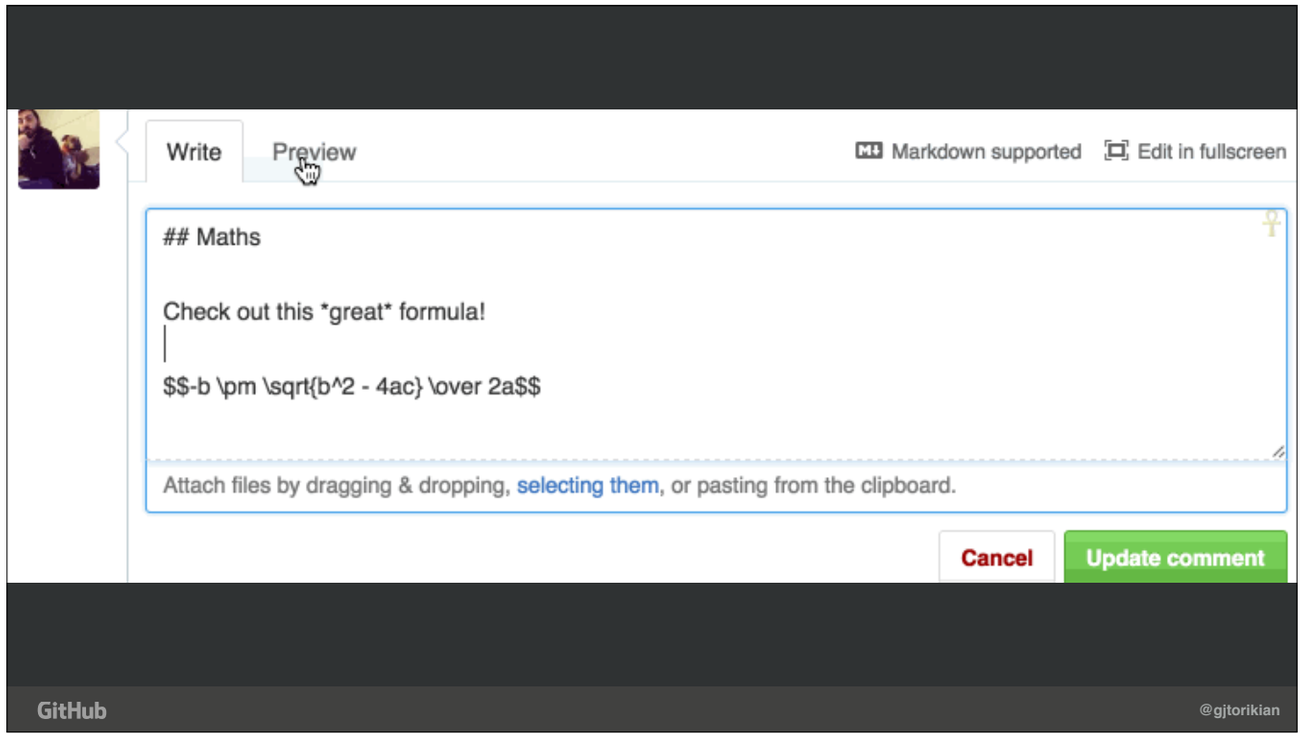
The reason a Bison grammar works well here is because TeX has, for the most part, a finite amount of keywords. mtex2MML knows all of those keywords, and knows how to act upon those keywords. It knows that when you write \sqrt, curly brace, variable, close curly brace}--that the action to take is to draw a square root.

All of this is a super simplification, but it's a general idea of how mtex2MML works. The fact that it's all in C also makes it an extraordinarily fast process.



The goal of mtex2MML was to allow it to be integrated into a markup language processor. Grammars and parsers for Markdown have existed for years.

Something that processes Markdown, for example, would know that the hash signs means “a header,” it would know that asterisks means “a list,” and now we can know that the dollar signs meant “this is math.”



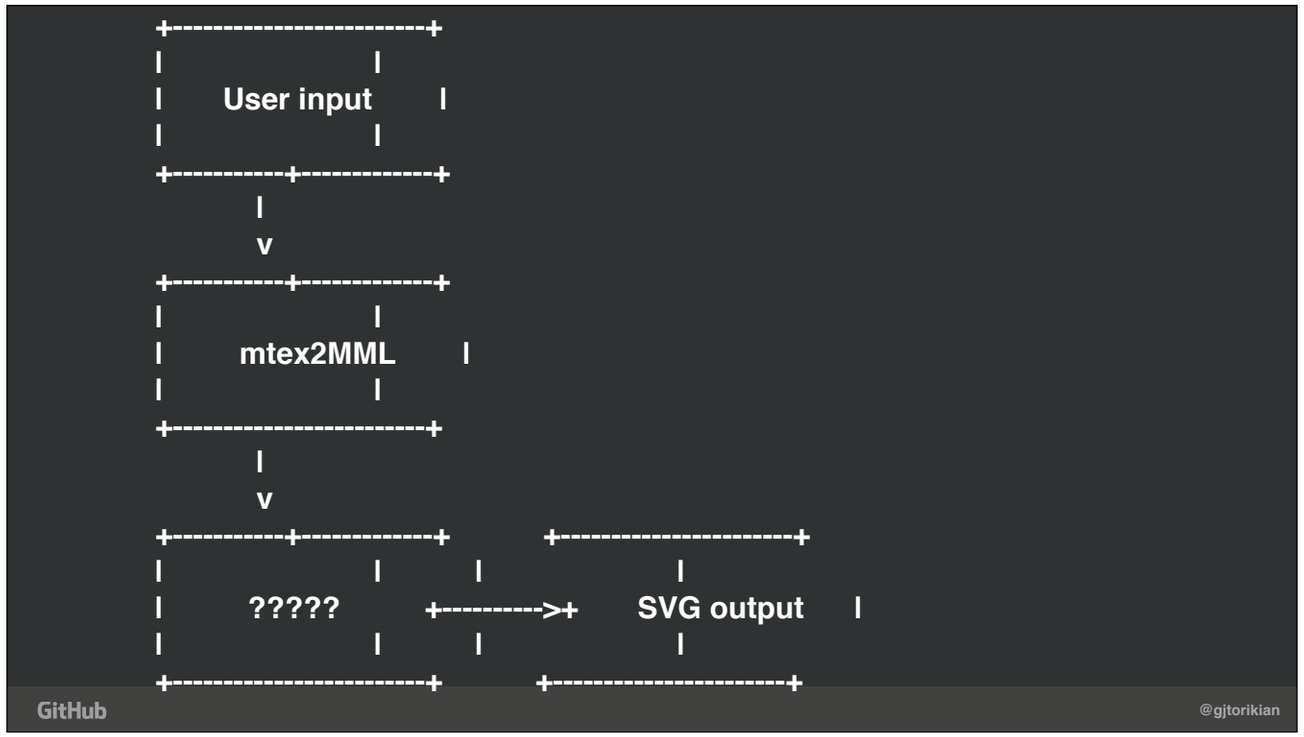
By moving the math processing to the server, you also get a far better experience in terms of performance for the user.

Here's a real life example of math rendering running on real life GitHub. Once the markup and math rendering process converts a comment, the conversion stays. There's no need to reflow or rerender any piece of the final output. That is the main advantage of serverside rendering versus clientside rendering.

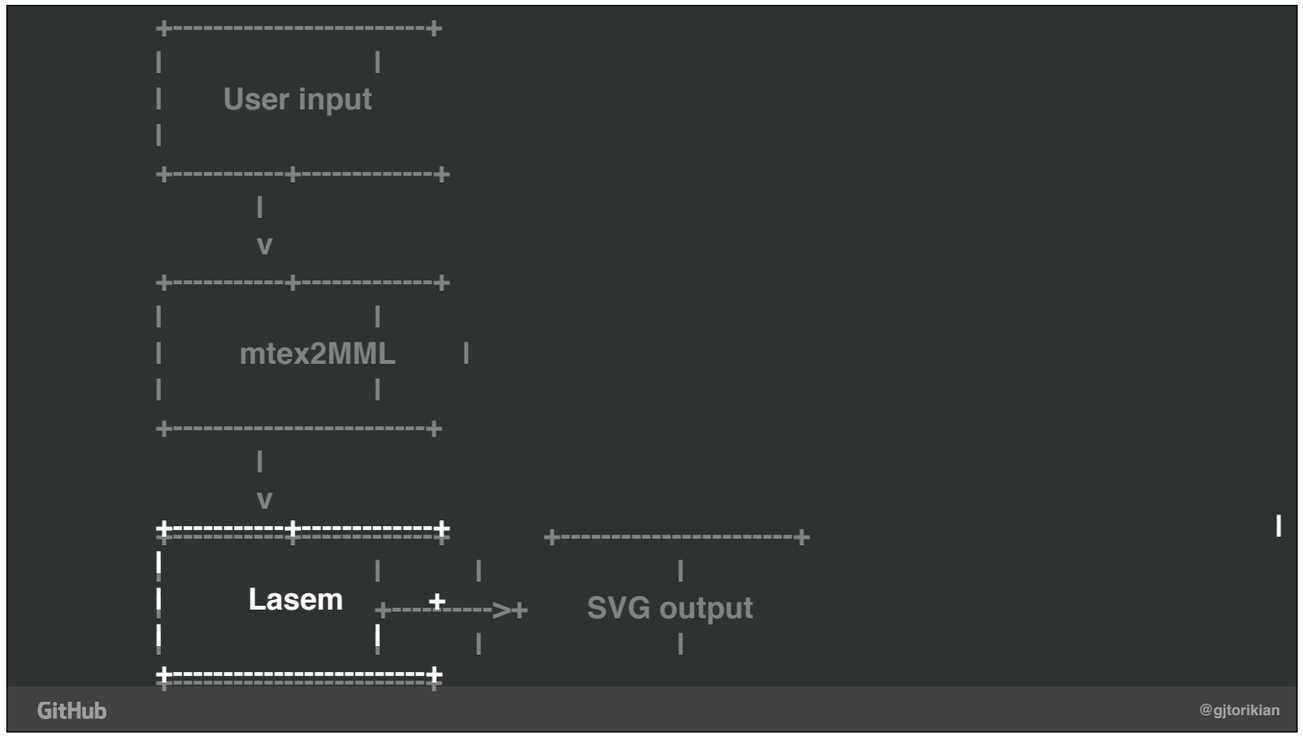
I also want to call out, explicitly, that I think MathJax is a fantastic solution for many use cases, especially ones where you don't have control over the server. Things like personal blogs or GitHub Pages require MathJax to exist. But in my humble opinion, we shouldn't just jump to the JavaScript solution because it's the only one available. We should be able to push for more options.

mtex2MML only outputs to MathML

If there’s one catch in all of this, it’s this: mtex2MML is only capable of outputting to MathML.



The problem here is that, as I talked about earlier, browser support in MathML is fairly terrible. What's needed is some way to transform that MathML into something that's able to be visualized in the browser.



Throughout the process of building mtex2MML, I stumbled across a GNOME project called Lasem that fits the bill perfectly.

Lasem

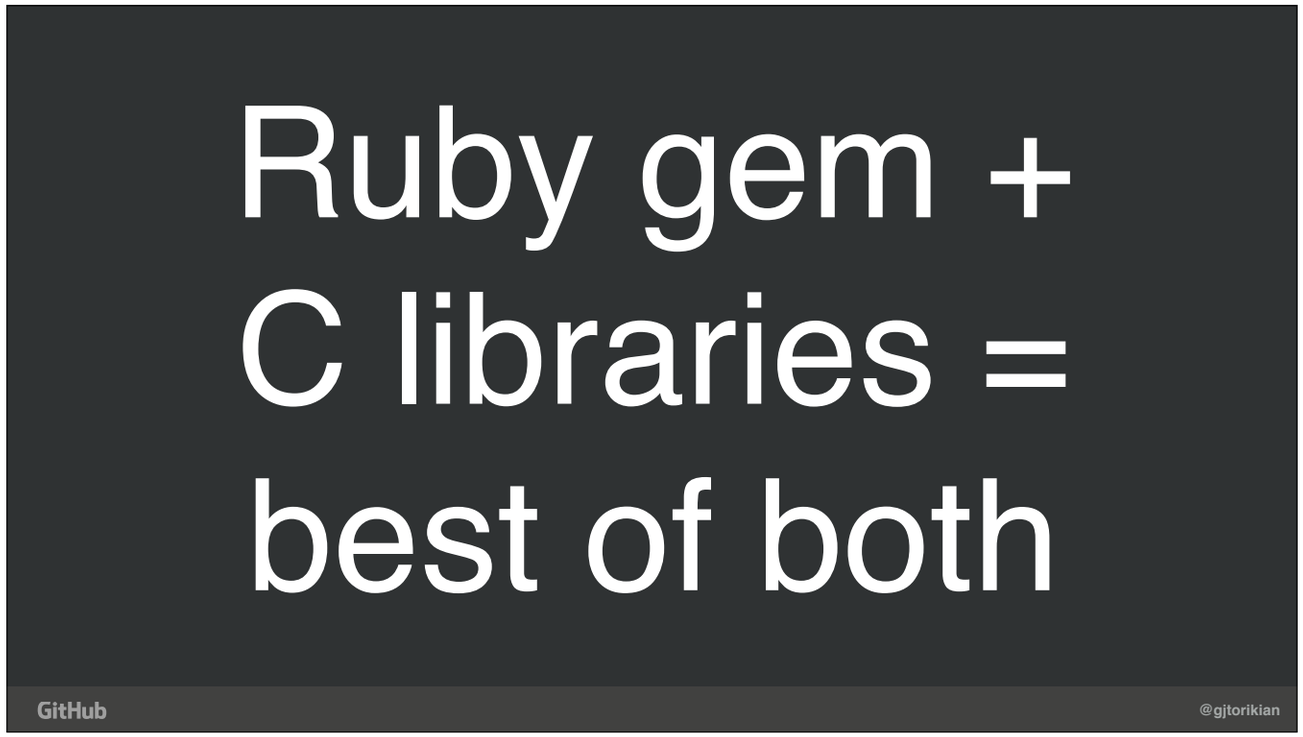
https://git.gnome.org/browse/lasem/

Lasem is, thankfully, also written in C. It takes MathML as input, and outputs SVG or PNG files. This is the final piece in our proposed pipeline to convert user input to high-fidelity output on the server.

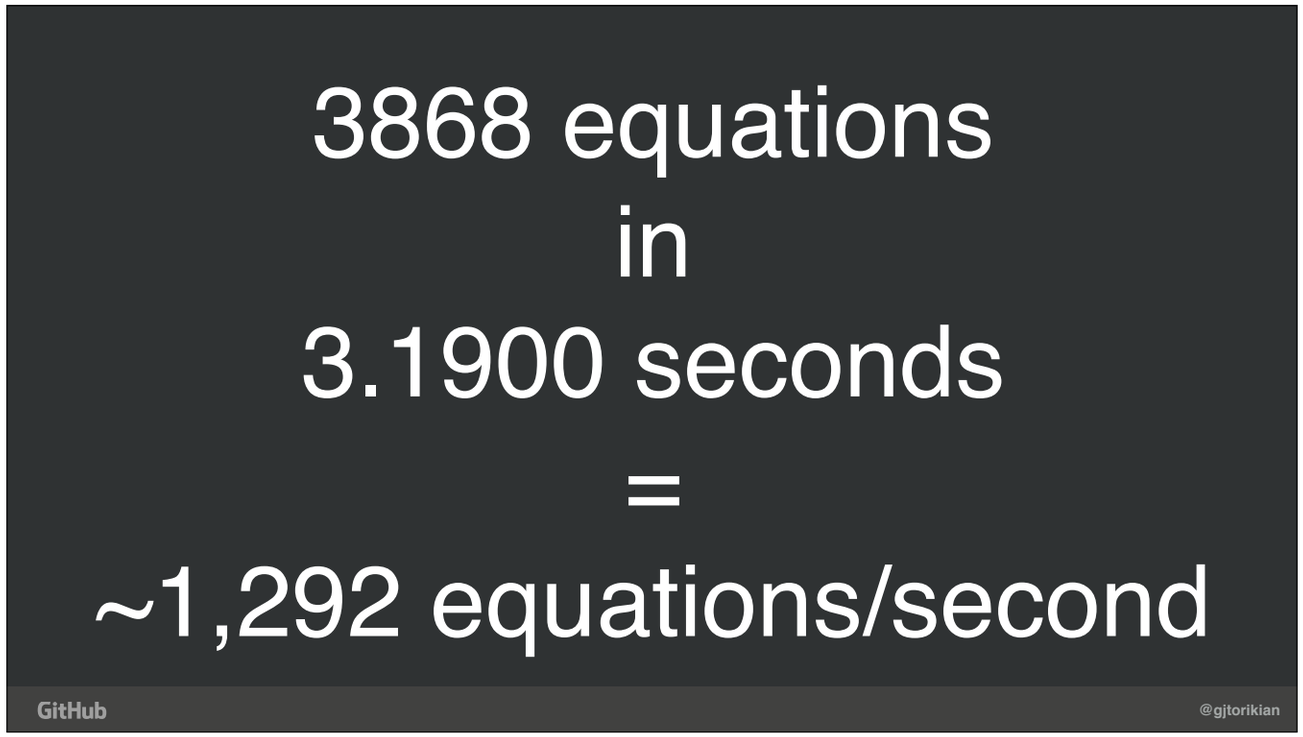
Mathematical

https://github.com/gjtorikian/mathematical

To wrap this talk up, I’d like to discuss, quickly, the Mathematical project. Mathematical is a Ruby gem that puts all of these ideas into practice.



Mathematical wraps mtex2MML and Lasem. Ruby has a fantastic integration with native code, as do many other high-level languages like Node.js or Python.
By wrapping our native C libraries in a higher-level language, we get all the performance benefits of mtex2MML and Lasem, with the ease of use of integration into other systems.



**** YOU SHOULD BE AT TWENTY EIGHT MINUTES ****

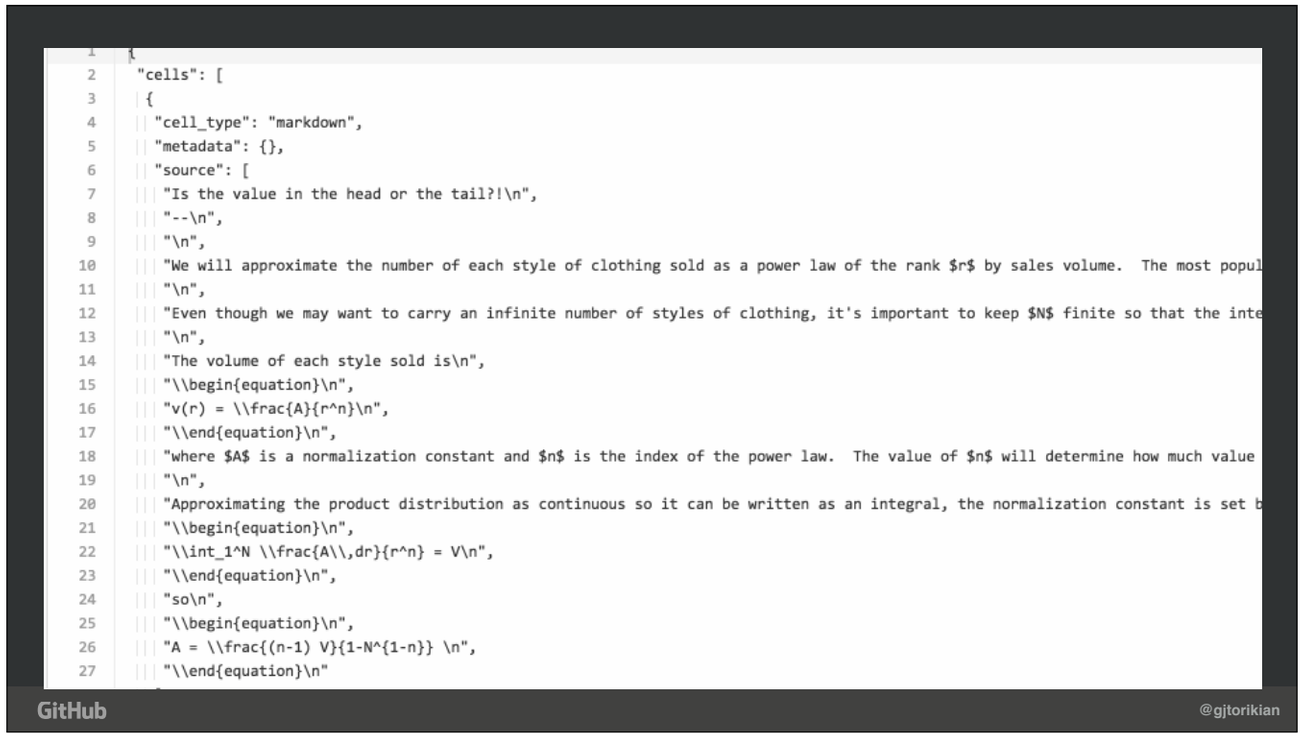
I ran a benchmark before I came in here of the latest Mathematical build. I was able to translate 3,868 equations in 3.19 seconds. This comes out to about 1,292 equations per second.

I admittedly haven’t run similar benchmarks in JavaScript, but I am willing to go out on a limb and say that this is very fast.

As an example of how Mathematical can help shape the future of rendering math, it’s time for the part of the talk where I get to show you everything I’ve said. And hopefully it won’t be a horrible disaster.

Usage

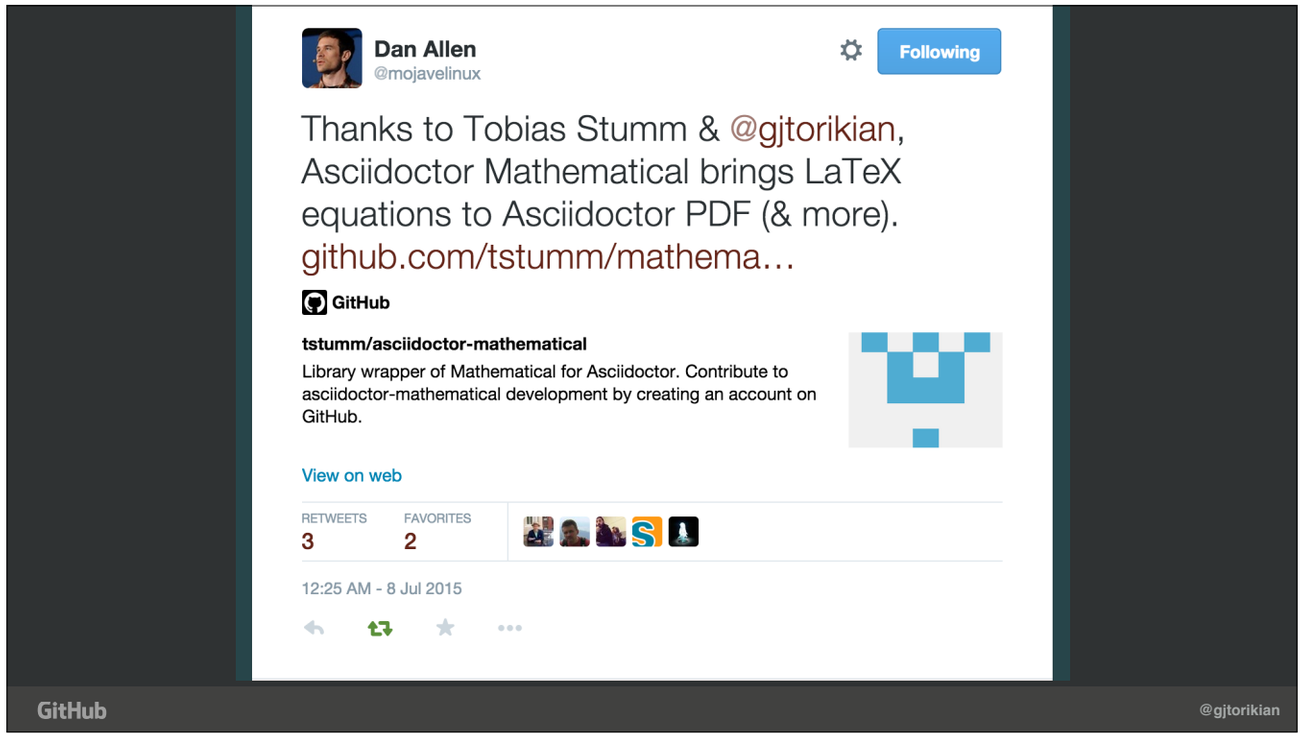
Internally, there’s been some interest in using Mathematical for rendering math equations at GitHub.



Recently, we've begun supporting IPython notebook rendering on GitHub. If you're not familiar with it, an IPython notebook basically a blob of JSON that is rendered into a rich document. It's huge in the data scientist community.

IPython already implements everything I just said. Content is written in Markdown, math is treated as a TeX equation. And MathJax is used to convert the math equations. So, once again, we have the disadvantage of a Python library that needs to call out to Javascript.

We've been slowly rolling out Mathematical to replace the math rendering, and we're seeing a noticeable improvement in both speed and quality amongst different browsers.



As well, some folks in the Asciidoctor community have picked up on Mathematical, and have begun using it to generate PDF documents with rich math support. These are documented written in Asciidoc, using math equations, rendering into PDF. So you don't really need to use LaTeX to create PDF documents either.

This is exactly the sort of use case I imagined when starting the mtex2MML project, so I'm thrilled that the open source community has picked up on it.

That’s all the time I’ve got! I hoped you enjoyed this talk and learned something about the history of rendering math. I look forward to a brighter future with projects incorporating mtex2MML, or perhaps spinning off the C code into something bigger and brighter.

Thanks again for your time. I’ll take any questions you’ve got now.