Pitfalls in Computation

Richard P. Brent
CARMA, University of Newcastle

MATH1510 Guest Lecture
16 Oct 2019

This lecture is about some of the many reasons why a computer might give you the wrong answer, or no answer at all.

I'll concentrate on examples with a computer science flavour. However, there are many more mathematical examples that you can easily find if you are interested.

Getting the wrong answer

There are many reasons why computers sometimes give incorrect answers, including:

- Entered the wrong data.
- Program bug.
- Compiler bug.
- Firmware/hardware bug.
- Transient error.
- Asked the wrong question.
- Used an inaccurate or unstable numerical algorithm.
- Neural net training error.

We'll look at some examples.

Entered the wrong data

In September 1997 the US Navy missile cruiser USS Yorktown was off the coast of Virginia. A sailor accidentally entered a zero into a data field of the *Remote Database Manager Program*, part of the *Smart Ship* system, which was designed "to automate tasks that sailors have traditionally done themselves".

This caused a divide by zero in the program, since whoever wrote the program had not bothered to check for zero data. The divide by zero caused a buffer overflow, and the "failsafe" system shut down.

Unfortunately the program also controlled the ship's propulsion, so the ship was "dead in the water" for more than two hours before the Navy figured out how to reboot the system.

Lucky it wasn't in a war zone!

Program bug

If you are developing your own programs, then a program bug is the most likely reason for wrong answers.

Some suggestions:

- Start with as simple a program as possible and debug that before adding refinements such as optimisations, bells and whistles.
- As you add “improvements” check that the output is still correct.
- Does the program specification match what you actually want the program to do?
- If possible, use two different methods and compare results.
- If possible, try to replicate published results before trying to get new results (but be aware that published results are not always correct).

Ariane 5 rocket crash

In June 1996 the European Space Agency launched its new Ariane 5 rocket, an upgrade of the old Ariane 4. Unfortunately the software had not been (sufficiently) upgraded. The rocket veered off course and exploded 37 seconds after liftoff. $US500 million worth of communications satellites on board were lost (not to mention ESA's reputation).

It turned out that horizontal velocity was stored in a 64-bit floating-point number and then converted to a 16-bit signed integer. This was OK for Ariane 4, but for Ariane 5 the number exceeded $2^{15} - 1$ and integer overflow occurred, with disastrous results.

Was this a program bug? More a bug in the program specification, which was not upgraded at the same time as the rocket.

Patriot missile problem

The Patriot missile provides another example where the software met the original specification, but the specification proved inadequate when the system was used in a way that was not anticipated by the designers.

During the first Iraq war (Feb 1991) an Iraqi Scud missile was fired towards Dhahran, Saudi Arabia. A US Patriot missile should have intercepted it, but failed to do so. 28 soldiers died as a result, and 97 were injured.

What went wrong?

It turned out that the Patriot system had an internal clock that incremented every 0.1 seconds, and the time (in seconds) was determined by multiplying the counter value by a 24-bit approximation to $1/10$. Note that $1/n$ is a non-terminating fraction in binary for any n that is not a power of 2, in particular for $n = 10$. Effectively the Patriot was multiplying by a number close to 0.0999999 instead of 0.1000000

The Patriot was intended to be a mobile system that would run for only a few hours at one site, and in that case the rounding error would not be serious. However, in Dhahran it ran for 100 hours, and the rounding error was 0.34 seconds. The Patriot became confused (presumably because it had two different values for the time), could not track the Scud missile, and treated it as a false alarm.

It has been said that the Patriot “missed” the incoming Scud, but this is misleading because the Patriot never left the ground!

Mars Climate Orbiter

In Dec 1998 the Mars Climate Orbiter was launched from Earth. It arrived at Mars in Sept 1999. However, a software error caused it to burn up in the Martian atmosphere.

A review board found that some data was calculated on the ground in imperial units (pound-seconds) and reported that way to the navigation system, which was expecting the data in SI (metric) units (newton-seconds).

Unfortunately most programming languages deal only with dimensionless quantities – the responsibility for conversion of units rests with the programmer!

Perhaps NASA learned something, because more recent missions to Mars (Mars Exploration Rovers) have been much more successful.

Concurrent Computing and Race Conditions

In concurrent computing, different resources may be shared between different tasks that run concurrently. This can cause problems unless the order in which shared resources are accessed is carefully controlled.

For example, suppose task A does something like (in C):
y = s; y = y+1; s = y (or s = s+1)
while task B is doing:
z = s; z = 2×z; s = z (or s = 2×s).

Suppose the shared variable s initially has the value 5. The final value of s could be 6, 10, 11, or 12, depending on the order in which the loads (. . . = s) and stores (s = . . .) are performed. (Check it out!) The outcome depends on a "race" between the tasks A and B.

Compiler bugs

If your program does not work as expected, it is tempting to blame the compiler. In nearly all cases the compiler is not to blame. It is usually a typo, logic bug, or a misunderstanding of the syntax/semantics of the programming language (e.g. beware implicit type conversions, the operators "=" and "==" in C, etc).

Turn on compiler warnings and don't ignore them unless you are sure it is safe to do so. Debug with optimisation turned off then see if turning it on changes anything. (If it does, check for uninitialised variables and array bound violations.)

Compiler bugs do exist, especially for "exotic" or rarely used features (e.g. extended precision). Avoid such features if possible.

The Pentium FDIV bug

Firmware/hardware bugs are the least likely, but also the most spectacular and expensive (for the computer manufacturer)!

Perhaps the best-known is the 1994 Intel Pentium “FDIV” bug.

After initially denying then downplaying the bug, Intel eventually offered to replace all faulty Pentium processors, at an estimated cost of $US475 million (1994 dollars).

Many users did not bother to get their processors replaced, so it cost Intel less than their estimate. However, no one at Intel got a Christmas bonus that year!

What was the FDIV bug?

When designing their Pentium processor to replace the 80486, Intel aimed to speed up floating-point scalar code by a factor of three compared to the 486DX chip,

The 486 used a traditional shift-and-subtract algorithm for division, generating one quotient bit per clock cycle. For the Pentium Intel decided to use the SRT algorithm that can generate two quotient bits per clock cycle. The SRT algorithm uses a lookup table to calculate intermediate quotients.

Intel's lookup table should have had 1066 table entries, but due to a faulty optimisation five of these were not downloaded into the programmable logic array (PLA). When any of these five entries is accessed by the floating point unit (FPU), the FPU fetches zero instead of the correct value. This results in an incorrect answer for FDIV (double-precision division) and related operations.

Example

The error is usually in the 9th or 10th decimal digit, but in rare cases it can be much worse.

FDIV is supposed to give a 53-bit result, i.e. about 16 decimal digits.

An example found by Tim Coe is

$$c = \frac{4195835}{3145727}.$$

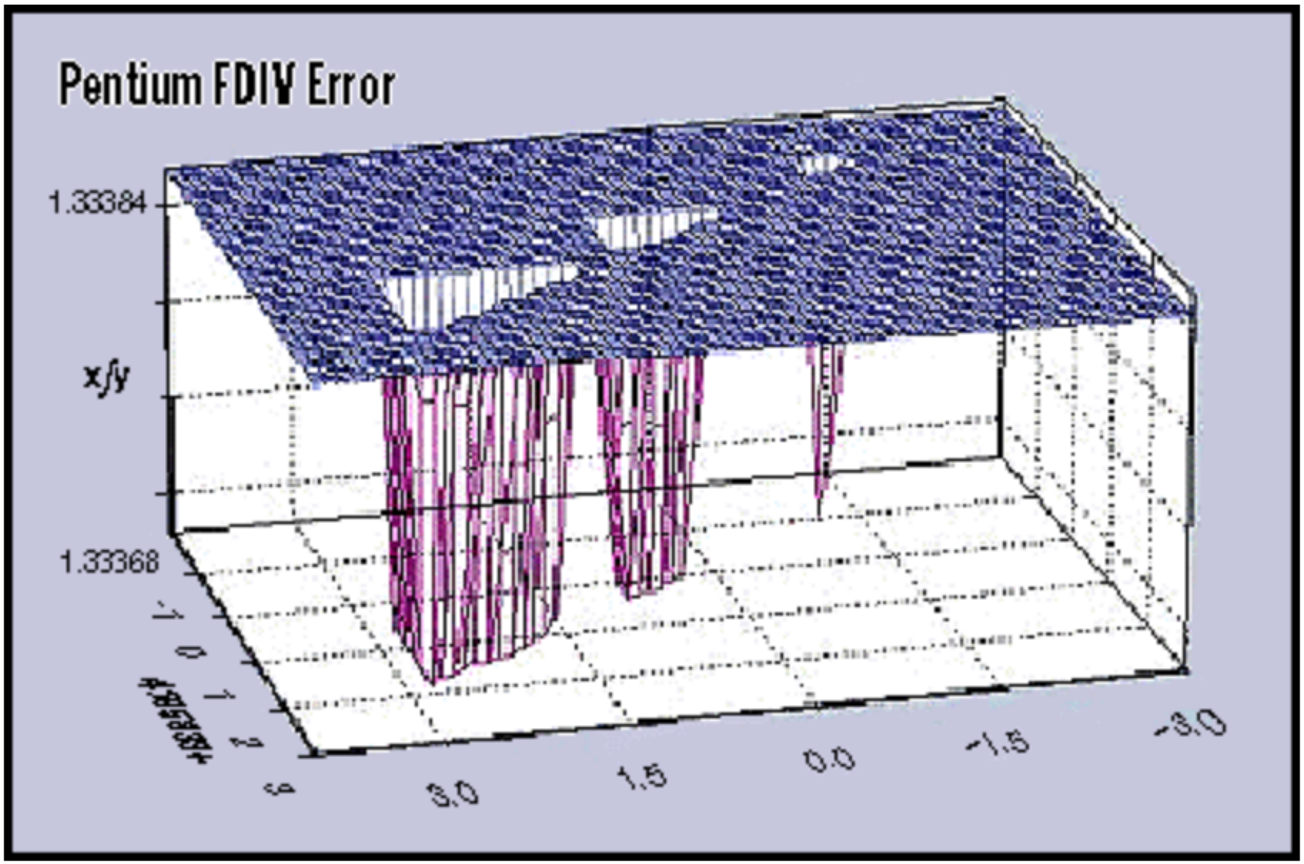
The correct value is

$$c = 1.333820449136241\cdots$$

but a faulty Pentium gives

$$c = 1.333\underline{7}3906\cdots$$

which is an error of about one part in 2^{14} (not one in 2^{53}).



A 3-D plot of the ratio x/y where $x \approx 4195835$, $y \approx 3145727$, calculated on a Pentium with the FDIV bug. The “spikes” indicate incorrect values.

Summary of Intel's reaction

1. There is no problem.
2. There is a small problem, but it is not serious.
3. The problem might be serious for some users; people who can prove that they are affected by the problem can have their Pentium processor replaced by Intel.
4. OK, we'll replace any flawed processor free of charge. [Note: the word "flaw" is used, not "bug".]

There are lots of Pentium bug jokes. For example:

Intel's new motto:

United We Stand,
Divided We Fall

Time-line

The history is interesting:

- July: Intel discovered the bug, but did not make this information public.
- Sept: Thomas Nicely, who was working on a number-theoretic computation related to “twin” primes, suspected the bug because he obtained different answers on Pentiums and 80486s.
- 30 Oct: Nicely, unable to convince Intel technical support, publicised the bug. Soon “all hell broke loose”.
- 7 Nov: Front page story in *Electronic Engineering Times*.

Time-line continued

- 22 Nov: Intel press release "$\cdots$ can make errors in the 9th digit. $\cdots$ [Only] theoretical mathematicians should be concerned."
- 5 Dec: Intel claimed the flaw would occur "once in $27,000$ years" for a typical spreadsheet user.
- 12 Dec: IBM Research said that the error could occur as often as "once every 24 days". IBM stopped shipping PCs based on the Pentium.
- 21 Dec: Intel said "We at Intel sincerely apologize for our recent handling of the recently publicized Pentium processor flaw" and offered to replace faulty processors.

Other discrepancies

Nicely continued his computations (related to twin primes and Brun's constant), and occasionally other discrepancies appeared between duplicated runs.

A *twin prime* is a prime number p such that $p + 2$ is also prime. For example $p = 101$ is a twin prime, because $p + 2 = 103$ is also prime.

Brun's constant is the sum of reciprocals of twin primes. It is finite, unlike the sum of reciprocals of all primes.

Nicely traced two discrepancies to defective memory (SIMM) chips; parity checking had failed to report the errors. Once a disk subsystem failure generated many incorrect (but plausible) results. Other discrepancies were probably caused by "soft" memory errors.

Soft memory errors

A *soft memory error* occurs when a cosmic ray or alpha particle flips one or more bits in memory. A single bit error should be detected by a parity check and can be corrected if the memory has “error checking and correction” (ECC) hardware. Usually ECC can detect (but not always correct) a double bit error – this is called SEC/DED. Some systems write an entry in an error log whenever an error is detected.

Soft memory errors are a known but not well-advertised feature of modern memory chips. According to Sun, a system with 10 GB of memory might get an “ECC event” every 100 to 1000 hours, though this depends on the solar sunspot cycle, the latitude, altitude, amount of shielding, degree of miniaturisation, whether the memory is interleaved, etc.

Soft memory errors are probably the most common sort of *transient* errors (errors that can not be replicated).

Another example of transient errors

GIMPS is the "Great Internet Mersenne Prime Search". This is a project to search for primes of the form $2^n - 1$. On 21 Dec 2018 GIMPS announced the Mersenne prime

$$2^{82589933} - 1 .$$

A number $N = 2^n - 1 > 3$ can be proved prime by performing a *Lucas-Lehmer* test: if $s_0 = 4$ and

$$s_{k+1} = s_k^2 - 2 \bmod N$$

then N is prime if and only if $s_{n-2} = 0$.

For example, if $n = 7$, we get the sequence $(4, 14, 67, 42, 111, 0)$, so 127 is prime.

GIMPS is cautious; all results are checked before being announced. This has avoided at least one embarrassment.

GIMPS error statistics

Occasionally two computers testing the same N find different values of s_{n-2}. Thus at least one is incorrect! Millions of Lucas-Lehmer tests have been checked, and the observed error rate is about 1.1%.

A "P-90 CPU-year" is the work done by a 90 Mhz Pentium in one year. According to George Woltman:

> *The average LL test now takes about 6.5 P-90 CPU-years. So my rough calculations are 0.011 errors per 6.5 CPU-years or 1 error every 600 P-90 CPU-years.*

Nowadays most machines are much faster than 90 MHz. If you have a cluster of 256 $\times$ 2 GHz machines, you can expect an error about once a month (unless reliability has improved)!

Case study – Primality testing

In 2002 Agrawal, Kayal and Saxena (AKS) surprised number theorists by finding a deterministic polynomial time primality test. This was certainly a great theoretical result. However, from a *practical* point of view, nothing changed.

Before AKS, the best practical algorithm was the Rabin-Miller *probabilistic* (or *Monte Carlo*) algorithm. If you run this algorithm once, it has a probability $< 1/4$ of giving the wrong answer.

Thus, if you run Rabin-Miller 100 times (using independent random numbers) and it gives the same answer each time, the probability of error is

$$< 4^{-100} < 10^{-60},$$

which is close enough to certainty for practical purposes.

Comparison of algorithms

Let's consider testing a 100-decimal digit number on a 1 GHz machine. (Cryptographic algorithms need to do this sort of thing.)

- 100 independent trials of Rabin-Miller takes 0.3 seconds (using a Magma implementation), with error probability $< 10^{-60}$, i.e. one error in 10^{60} trials. There are about 10^{57} atoms in the whole solar system!
- AKS takes 37 weeks and, using the GIMPS statistics, the result has a probability of error exceeding 0.01 (when testing a number that is actually prime) even if the program is correct!

Which would you choose ?

Asked the wrong question or used a bad algorithm

If someone asks your advice on a numerical or statistical problem, they usually say

"I want to do XXX."

You can answer

"This is how to do XXX $\cdots$"

but it is often better to answer

"Do you *really* want to do XXX?

Wouldn't YYY be better?"

Example – Runge's phenomenon

For example, you might have some data y_j given at n equally spaced points x_j, and want to fit an approximation $P_n(x)$ such that $P(x_j) = y_j$. It's natural to consider a polynomial $P_n(x)$ of degree $n-1$. However, this is likely to be disastrous as $P_n(x)$ may oscillate violently between the data points (and the oscillations only get worse as n increases).

Carl Runge gave the example

$$f(x) = \frac{1}{1+25x^2}$$

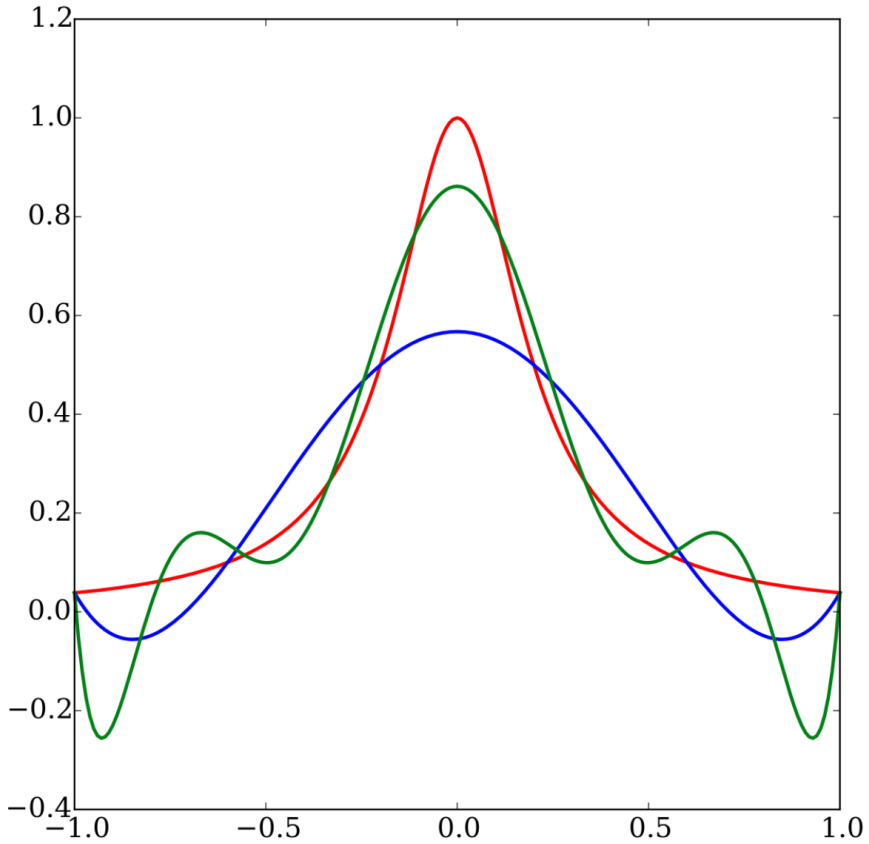
on $[-1,1]$ with equally spaced points (spacing $h = 2/(n-1)$).

Unfortunately

$$\lim_{n\to\infty} ||f - P_n||_\infty = +\infty.$$

Here, $||\cdots||_\infty$ means the maximum over the interval $[-1,1]$.

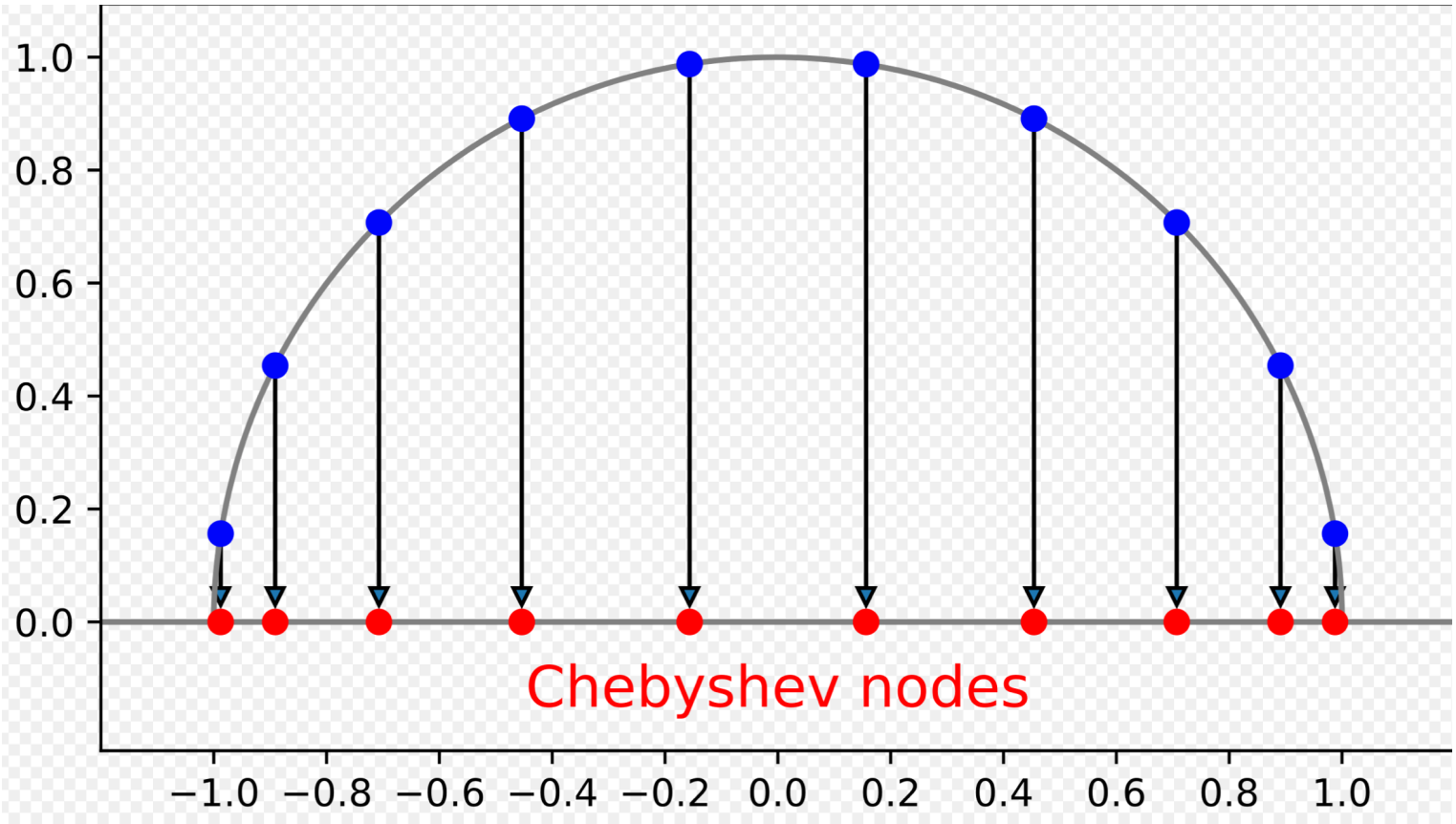
Illustration



The red curve is the Runge function.
The blue curve interpolates at 6 equally-spaced points.
The green curve interpolates at 10 equally-spaced points.
The oscillations near ± 1 get worse as you use more points!

A better question

Instead of asking to interpolate at equally-spaced points, we could ask to interpolate at Chebyshev points.



The Chebyshev points (or nodes) are the projections on the x-axis of equally spaced points on a unit semicircle. They cluster towards the end-points of the interval $[-1, 1]$.

Polynomial interpolation

Polynomial interpolation is not always bad – interpolation at Chebyshev points

$$x_j = \cos\left(\frac{2j-1}{2n}\pi\right), \quad j = 1, \ldots, n$$

works for Runge's example and many others.

Weierstrass's theorem says that *any* continuous function can be approximated arbitrarily closely by a polynomial. Such a polynomial can be computed using the Remez algorithm.

In practice it is often simpler and better to use piecewise polynomials, i.e. splines.

Splines can be generalised to two or three dimensions, e.g. for approximating the shape of an aeroplane wing.

Machine learning and neural nets

Recently, artificial intelligence (AI) has become important because of its many potential applications, e.g. to face recognition, machine translation, medical diagnosis, control of autonomous vehicles, etc.

Many such systems depend on training "neural nets", which are hardware/software analogues of a simplified model of the human brain. This can work well in limited domains (e.g. chess/go, where the rules are simple and well-defined).

A problem is that no one (not even the programmers) understands exactly how such neural nets work. They are just too complicated and unintuitive. Thus, no one knows all their limitations.

Example

There have been several crashes (and some fatalities) involving autonomous vehicles.

For example, on 7 May 2016, a Tesla Model S was driving in "Autopilot" mode. An oncoming semitrailer was making a turn at an uncontrolled (i.e. no traffic lights) intersection, but the Tesla software did not recognize the semi and continued straight on without applying the brakes. The Tesla actually passed underneath the trailer, and the "driver" (of the Tesla) was killed. I wrote "driver" because the real driver was the software, not the human!

There are different theories about the cause of the accident, but a plausible one is that the software was not able to react to the trailer coming from the side, as it was trained to avoid rear-end collisions by detecting the rear of vehicles ahead.

Further reading

A recent book on the topic of this lecture is:

Thomas Huckle and Tobias Neckel, Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science, SIAM, 2019.

You could also look at my old talk:

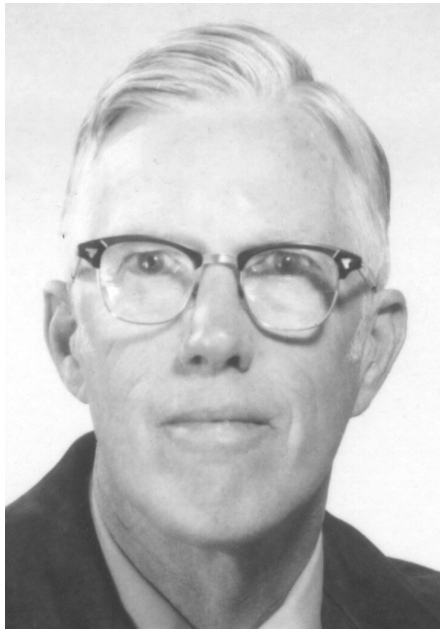
Pitfalls in Computation: Random and not so Random Numbers, available from `https://maths-people.anu.edu.au/~brent/pd/NAMS06t4.pdf`.

Inspiration

The topic of this lecture was inspired by the paper:

George E. Forsythe, "Pitfalls in Computation, or why a Math Book isn't Enough", *Amer. Math. Monthly* 77 (1970), 931–956. https://www.jstor.org/stable/2318109.

Forsythe was the founder and first head of the Computer Science Department at Stanford University.



George Forsythe
1917–1972