SINTEF

CONSERVATION LAWS ON GPUS:
BEST PRACTICES

André R. Brodtkorb
Researcher, Department of Mathematics and Cybernetics, SINTEF Digital
Associate Professor, OsloMet – Oslo Metropolitan University

Challenges for scientific software development

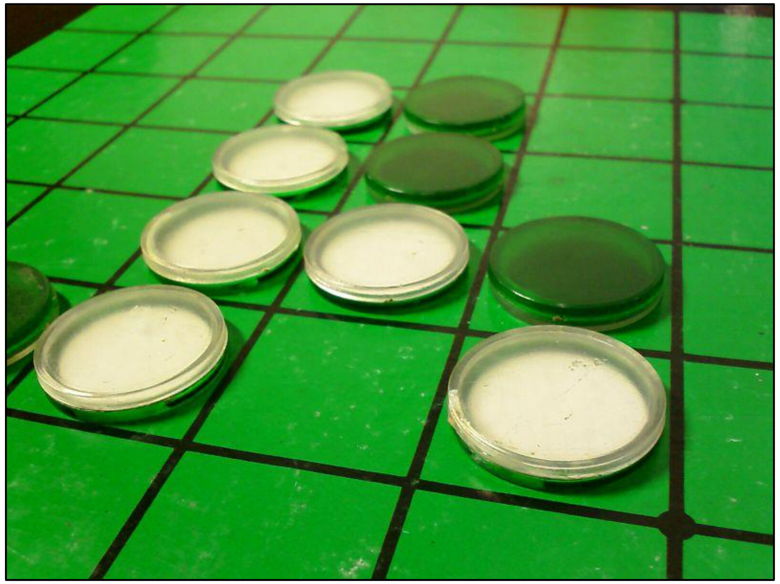
Challenges for scientific software development

- Developing scientific software is dead hard
  - Have to have deep knowledge of both the science and the programming
- Working with parallel computing is a major challenge by itself
  - "Everything" can go wrong
  - Debugging is near impossible
- We'll look into some typical challenges related to floating point

Floating point

Floating point

- Floating point is like chess:
  it takes minutes to learn, and
  a lifetime to master
  (or, at least it's quite complex
  for such a simple definition)



A game of Othello, Paul 012, CC-BY-SA 3.0

[1] IEEE Computer Society (August 29, 2008), *IEEE Standard for Floating-Point Arithmetic*

Intel Pentium with FDIV bug, Wikipedia, user Appaloosa, CC-BY-SA 3.0

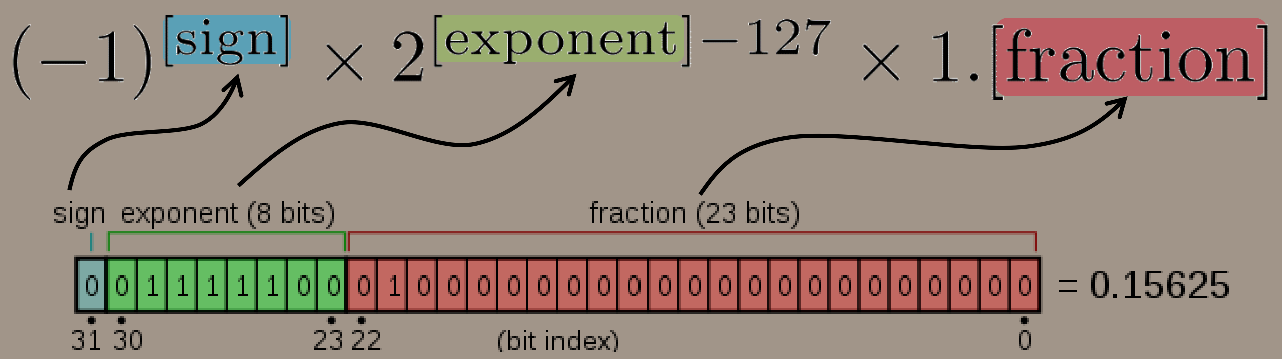
"update [...] to address the hang that occurs when parsing strings like “2.2250738585072012e-308” to a binary floating point number” [1]

[1] http://www.oracle.com/technetwork/java/javase/fpupdater-tool-readme-305936.html

SINTEF

A floating point number on a binary computer

- Floating point numbers are represented using a binary format:



Floating point format [Wikipedia, en:User:Fresheneesz, traced by User:Stannered, CC BY-SA 3.0]

- Defined in the IEEE-754-1985, 2008 standards
  - 1985 standard mostly used up until the last couple of years

Rounding errors

- Floating point has limited precision
- All intermediate results are rounded
- Even worse, not all numbers are representable in floating point (limited precision)
- Demo: 0.1 in IPython

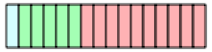
Python:

```
> print 0.1
0.1
> print "%.10f" % 0.1
0.1000000000
> print "%.20f" % 0.1
0.10000000000000000555
> print "%.30f" % 0.1
0.100000000000000005551115123126
```

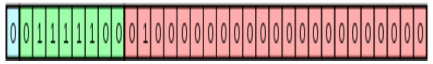
SINTEF

Floating point variations (IEEE-754 2008)

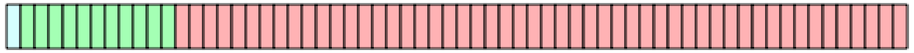
- Half: 16-bit float: Roughly 3-4 correct digits



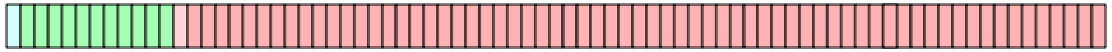
- Float / REAL*4: 32-bit float: Roughly 6-7 correct digits



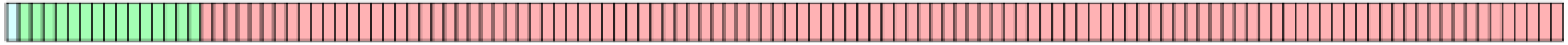
- Double / REAL*8: 64-bit float: Roughly 13-15 correct digits



- Long double / REAL*10: 80-bit float: Roughly 18-21 correct digits



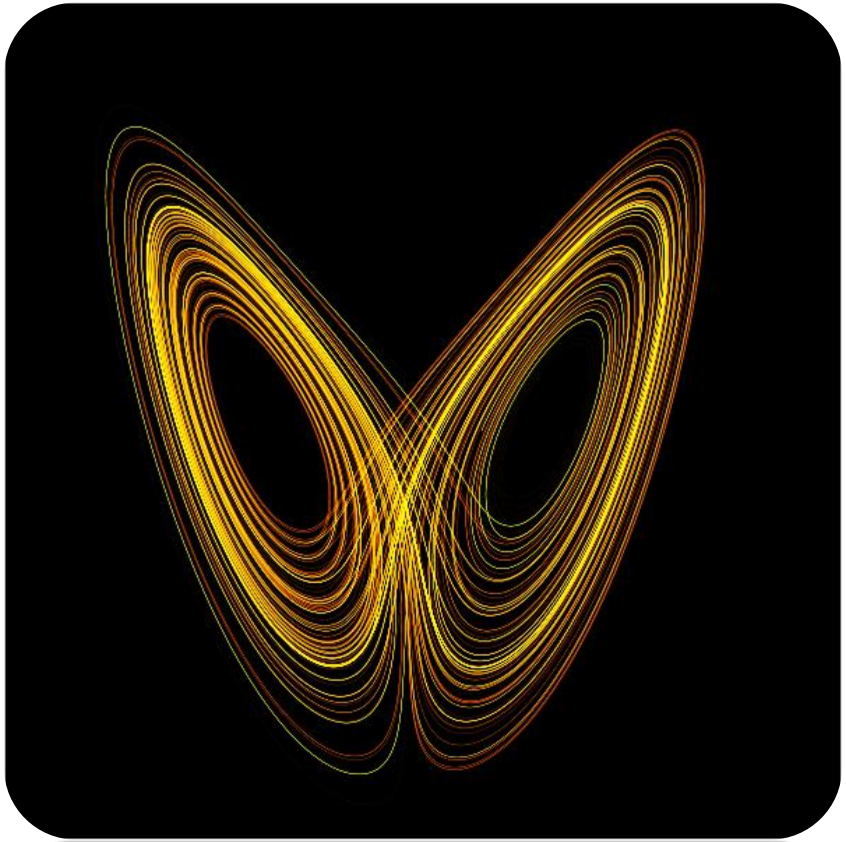
- Quad precision: 128-bit float: Roughly 33 - 36 correct digits



SINTEF

Floating point and numerical errors

- Some systems are chaotic
  - Is single precision accurate enough for your model?
  - Is double precision --"--?
  - Is quad precision --"--?
  - Is ...

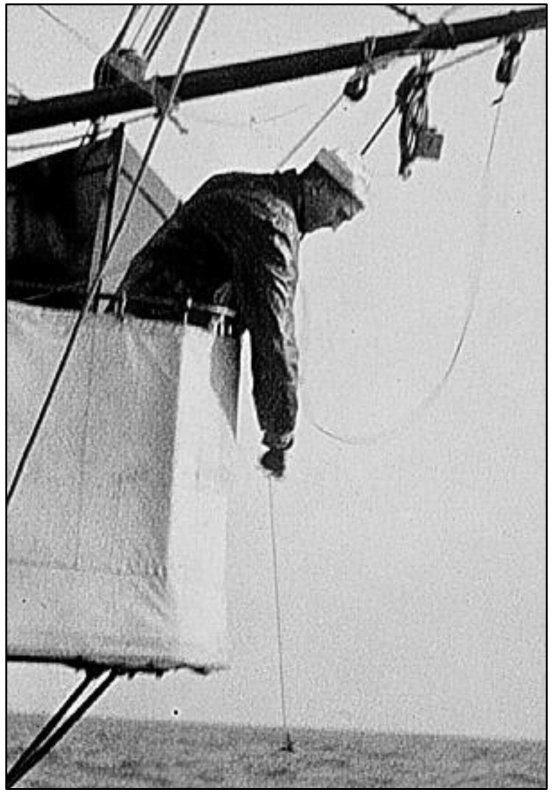


Lorenz strange attractor, Wikimol, wikipedia, CC-BY-SA 3.0

- Put another way:
  - What is the minimum precision required for your model?

There are often many sources for errors

- Garbage in, garbage out
- Many sources for errors
  - Humans!
  - Model and parameters
  - Measurement
  - Storage
  - Gridding
  - Resampling
  - Computer precision
  - …



Seaman paying out a sounding line during a hydrographic survey of the East coast of the U.S. in 1916. (NOAA, 2007).

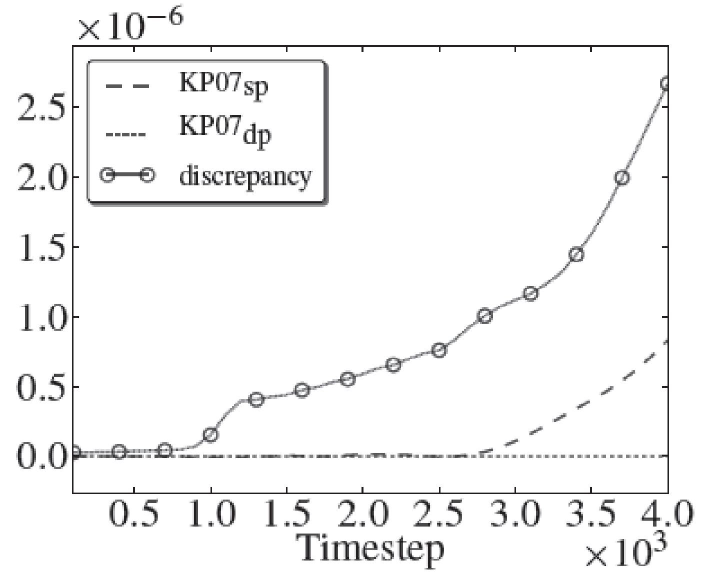
Recycle image from recyclereminders.com
Cray computer image from Wikipedia, user David.Monniaux

Example: Single versus double precision in shallow water

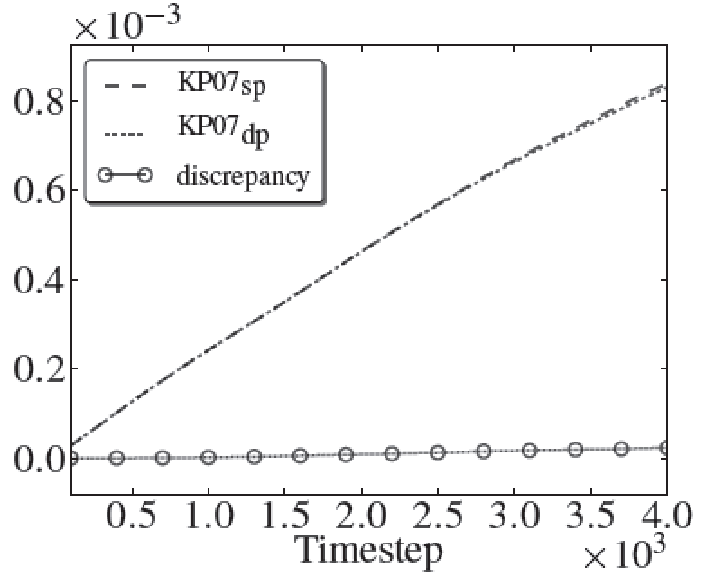
- Shallow water equations: Well studied equations for physical phenomenon
- Difficult to capture wet-dry interfaces accurately
- Let's see the effect of single versus double precision measured as error in conservation of mass

Single versus double precision [1]

- Simple case (analytic-like solution)
  - No wet-dry interfaces
  - Single precision gives growing errors that are "devastating"!



- Realistic case (real-world bathymetry)
  - Single precision errors are drowned by model errors



[1] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie and J. R. Natvig, **Simulation and Visualization of the Saint-Venant System using GPUs**, *Computing and Visualization in Science, 2011*

SINTEF

Catastrophic and benign cancellations [1]

- A classical way to introduce a large numerical error is to have a catastrophic cancellation:

$$x^2 - y^2 \Rightarrow (x - y)(x + y)$$

- The first variant above is subject to catastrophic cancellation if x and y are relatively close. The second does not suffer as badly from this catastrophic cancellation!

- Same for the quadratic formula: If c very small compared to b, we get catastrophic cancellation:

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

vs

$$r1 = \frac{-b - sign(b)\sqrt{b^2 - 4ac}}{2a}$$

$$r2 = \frac{c}{a * r1}$$

[1] *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, David Goldberg, Computing Surveys, 1991

SINTEF

So what should I use?

- Single precision
  - Single precision uses **half** the memory of double precision
  - Single precision executes **twice** as fast for certain situations (SSE & AVX instructions)
  - Single precision gives you **half** the number of correct digits

- Double precision is not enough in certain cases
  - Quad precision? Arbitrary precision?
  - Extremely expensive operations (100x+++ time usage)

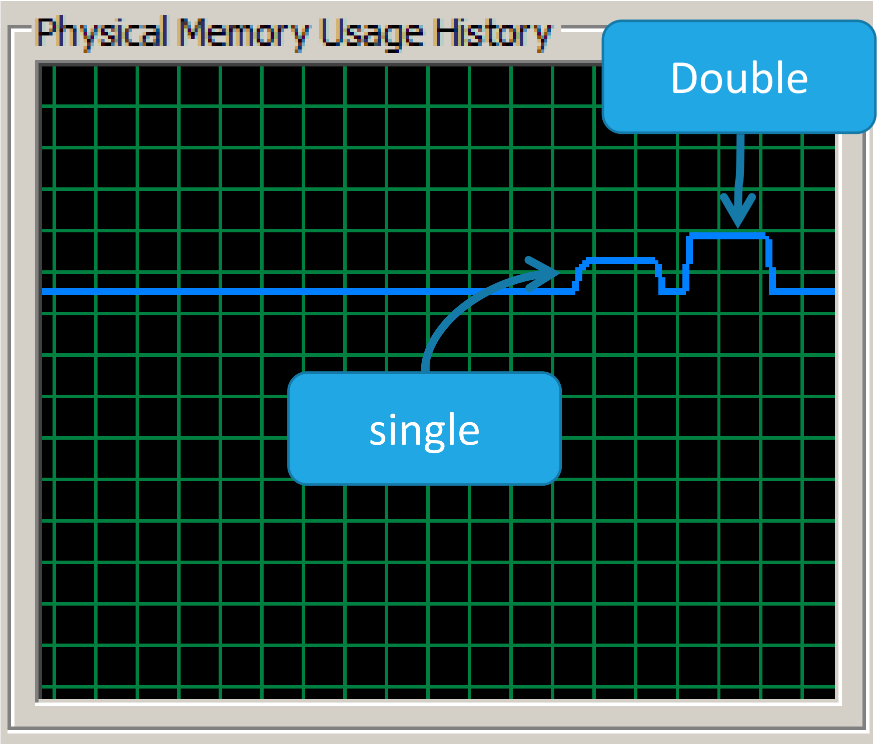
SINTEF

Floating point allocation demo

- Memory allocation example
  - How much memory does the computer need if I'm allocating 100.000.000 floating point values in a) single precision, and b) double precision?

Allocating float:
Address of first element: 00DC0040
Address of last element: 18B38440
Bytes allocated: 400000000

Allocating double:
Address of first element: 00DC0040
Address of last element: 308B0840
Bytes allocated: 800000000



Floating point summation demo

Floating point example

- What is the result of the following computation?

```
val = 0.1;
for (i=0  to 10.000.000) {
    result = result + val
}
```

Float:
Floating point bits=32
1087937.00000000000000000000000000000000000000000000000000
Completed in 0.01859299999999999841726605609437683597207069396973 s.

Double:
Floating point bits=64
999999.99983897537458688020706176757812500000000000000000
Completed in 0.02386800000000000326849658449646085500717163085940 s.

Long double (__float80):
Floating point bits=128
1000000.00000008712743237992981448769569396972656250000000
Completed in 0.02043599999999999930477834197972697438672184944153 s.

Quad (__float128):
Floating point bits=128
1000000.00000000000000000000000000000000000000000000000000
Completed in 1.39770400000000000574686964682769030332565307617188 s.

The patriot missile...

- Designed by Raytheon (US) as an air defense system.
- Designed for time-limited use (up-to 8 hours) in mobile locations.
- Heavily used as static defenses using the Gulf war.
- Failed to intercept an incoming Iraqi Scud missile in 1991.
- 28 killed, 98 injured.

The patriot missile…

- It appears, that 0.1 seconds is not really 0.1 seconds…
  - Especially if you add a large amount of them

Hours	Inaccuracy (sec)	Approx. shift in Range Gate (meters)
0	0	0
1	.0034	7
8	.0025	55
20	.0687	137
48	.1648	330
72	.2472	494
100	.3433	687

http://sydney.edu.au/engineering/it/~alum/patriot_bug.html

Floating point and parallelism

Floating point and parallelism

- Fact 1: Floating point is non-associative:
  - a*(b*c) != (a*b)*c
  - a+(b+c) != (a+b)+c
  - …
- Fact 2: Parallel execution is non-deterministic
  - Reduction operations (sum of elements, maximum value, minimum value, average value, etc.)
- Combine fact 1 and fact 2 for great joys!

Demo time ver 3

- OpenMP summation of 10.000.000 numbers using 10 threads

```
val = 0.1;
#omp parallel for
for (i=0  to 10.000.000) {
    result = result + val
}
```

OpenMP float test using 10 threads
Float:
Floating point bits=32
Run 0: 976668.75000000000000000000000000000000000000000000000000
Run 1: 976759.37500000000000000000000000000000000000000000000000
Run 2: 976424.87500000000000000000000000000000000000000000000000
Run 3: 977388.37500000000000000000000000000000000000000000000000
Run 4: 981089.06250000000000000000000000000000000000000000000000
Run 5: 976620.25000000000000000000000000000000000000000000000000

Double:
Floating point bits=64
Run 0: 1000000.00003875180000000000000000000000000000000000000000
Run 1: 1000000.00003898310000000000000000000000000000000000000000
Run 2: 1000000.00003432810000000000000000000000000000000000000000
Run 3: 1000000.00003912390000000000000000000000000000000000000000
Run 4: 1000000.00003827200000000000000000000000000000000000000000
Run 5: 1000000.00003756480000000000000000000000000000000000000000

SINTEF

Floating point and parallelism

- Why is parallel summation "more accurate" than serial summation in this case?

Kahan summation [1]

- It appears that naïve summation works really poorly for floating point, especially with parallelism
- We can try to use algorithms that take floating point into account

```
 function KahanSum(input)
    var sum = 0.0
    var c = 0.0          //A running compensation for lost low-order bits.
    for i = 1 to input.length {
        y = input[i] - c    //So far, so good: c is zero.
        t = sum + y         //Alas, sum is big, y small,
                            //so low-order digits of y are lost.
        c = (t - sum) - y   //(t - sum) recovers the high-order part of y;
                            //subtracting y recovers -(low part of y)
                            //Algebraically, c should always be zero.
                            //Beware eagerly optimising compilers!
        sum = t
    }
return sum
```

[1] Inspired by Bob Robey, EPSum, ICERM 2012 talk, http://faculty.washington.edu/rjl/icerm2012/Lightning/Robey.pdf

SINTEF

Demo time ver 4

- Kahan summation in parallel!

Float:
Floating point bits=32
Traditional sum, Kahan sum
Run 0: 499677.062500, 4996754.500
Run 1: 499679.250000, 4996754.500
Run 2: 499677.468750, 4996754.500
Run 3: 499676.312500, 4996754.500
Run 4: 499676.687500, 4996754.500
Run 5: 499679.937500, 4996754.500

Double:
Floating point bits=64
Traditional sum, Kahan sum
Run 0: 500136.4879299310900, 5001364.87929929420
Run 1: 500136.4879299307400, 5001364.87929929420
Run 2: 500136.4879299291600, 5001364.87929929420
Run 3: 500136.4879299313800, 5001364.87929929420
Run 4: 500136.4879299254400, 5001364.87929929420
Run 5: 500136.4879299341700, 5001364.87929929420

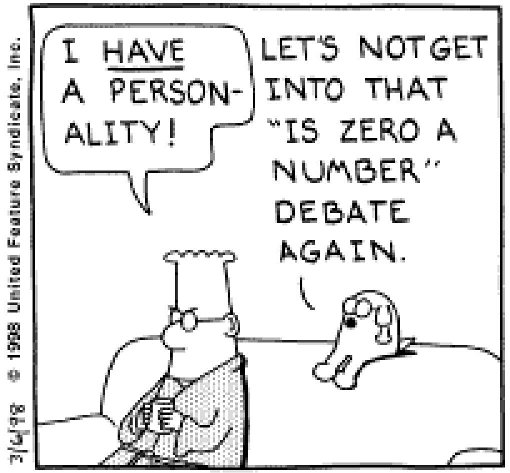
Advanced floating point

Rounding modes

- Round towards +infinity (ceil)
- Round towards –infinity (floor)
- Round to nearest (and up for 0.5)
- Round to nearest (and towards zero for 0.5)
- Round towards zero
- **Can be used for interval arithmetics!**

Special floating point numbers

- Signed zeros -0 != +0



- Signed not-a-numbers:
  quiet NaN, and signaling NaN (gives exception)
  examples: 0/0, sqrt(-1), …
    (x == x) is false if x is a NaN

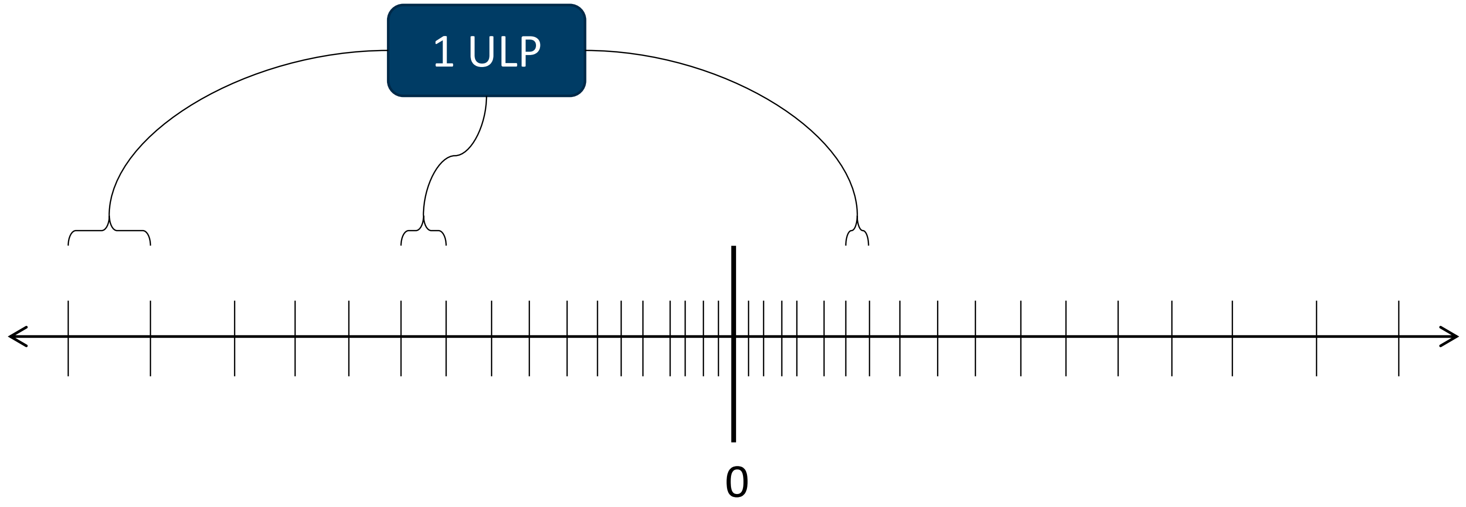
Special floating point numbers

- Signed infinity
  - Numbers that are too large to represent
    5/0 = +infty, -8/0 = -infty

- Subnormal or denormal numbers
  - Numbers that are too small to represent

Units in the last place [1]

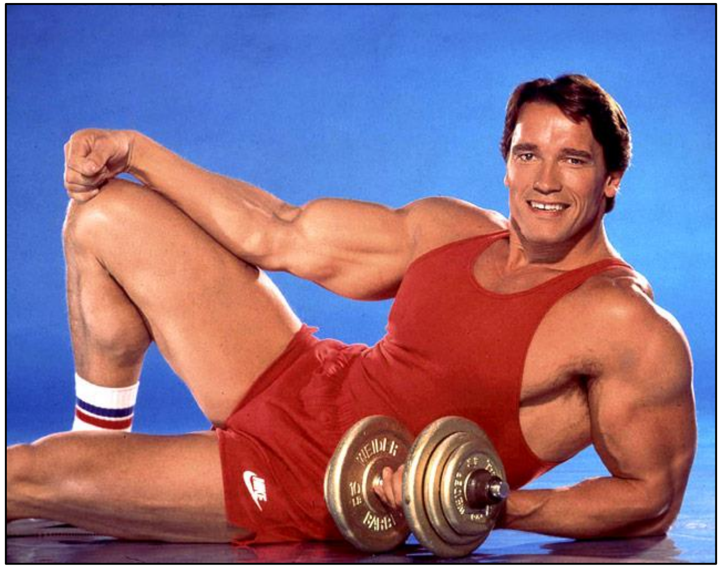
- Unit in the last place or unit of least precision (ULP) is the spacing between floating point numbers

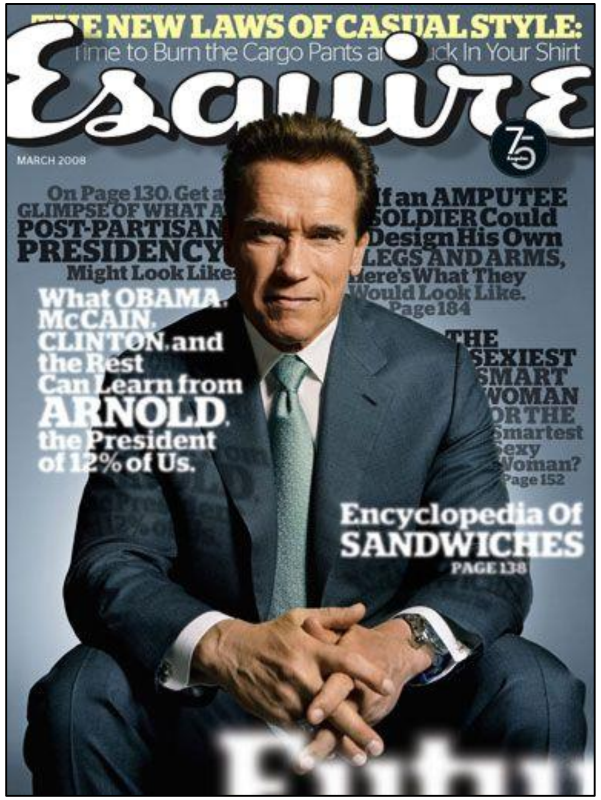


- "The most natural way to measure floating point errors"
- Number of contaminated digits: $\log_2 n$ when the error is n ulps
- Numbers close to zero have the smallest ULPs!

[1] What every computer scientist should know about floating-point arithmetic, David Goldberg, Computing Surveys , 1991

Some differences between 1985 and 2008





- Floating point multiply-add as a fused operation
  - a = b*c+d with only **<u>one</u>** round-off error
  - GPUs implement this already
- This is basically the same deal as the extended precision.
  - It's a good idea to use this instruction, but it gives "unpredictable" results
  - Users need to be aware that computers are not exact, and that two computers will not always give the same answer

Best Practices

See also
Best Practices for Scientific Computing,
Greg Wilson et al., 2012, arXiv:1210.0530

Keep it simple!

KISS: Keep it simple, stupid

- Design your code and work flow so "anyone" can repair it using standard tools
- If it's extremely complicated, does it really have to be?
- Simplicity in design is a virtue

- A common pitfall for computer scientists is to design "the one software to rule them all" instead of small easy-to-use components with a single use

Write elegant, clean code efficiently

Use a high-level language

- Your productivity increases dramatically the less details you have to consider
- Use an interpreted languages to also avoid compilation times:
  - Python
  - Matlab
  - Etc.

Write programs for people, not computers

- If a code is easy to read, it is easier to check if it is doing what it should
- Does the code you just wrote make sense to "most people"?
- Human memory is extremely limited: "a program should not require its readers to hold more than a handful of facts in memory at once"

Store changes and development history

Use version control

- Learn how to see the difference (diff) between two versions of the software, and how to revert changes
- Put "everything that has been created manually" in version control
- Version control is also a simple backup system

Use the computer to record history

- Data and source code provenance should automatically be stored
  "history" in Matlab or the Linux command-line,
  "doskey /history" on windows command line,
  Ipython, etc.
- Automatically record versions of software and data,
  and parameters used to produce results

Optimization and testing

Optimize software only after it works correctly

- When it works, use a profiler to find out what the bottleneck is
- Software developers write the same amount of code independently of the language: "write code in the highest-level language possible"

Write tests

- Regression testing => has something changed
- Verification testing => does the code produce known correct/analytical solutions?
- Run the tests regularly

Software testing

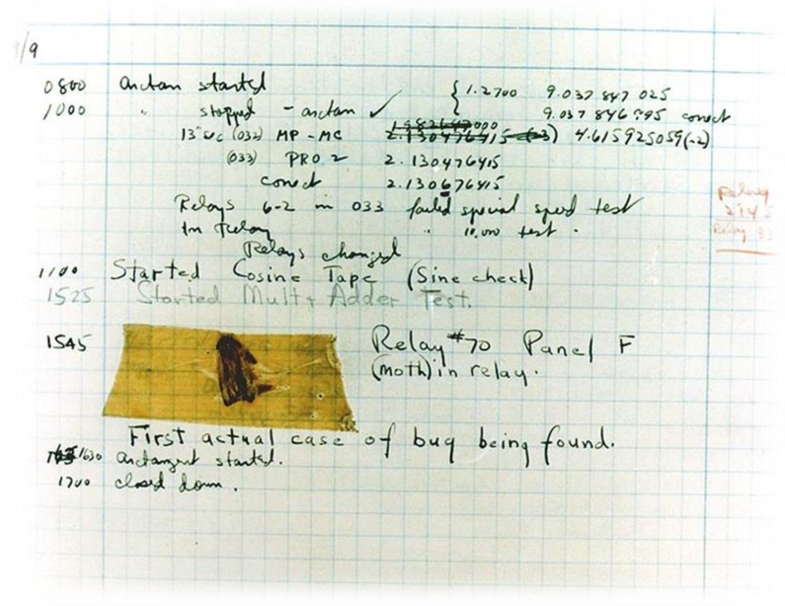
- Software testing is important for having trust in computer programs

- The simplest kind of test, a regression test, will check that the program output does not change

- Feature tests and unit tests that test specific features and parts of the software give the expected output

- Testing of fixed bugs to make sure they do not reappear

- More advanced tests include verification and validation



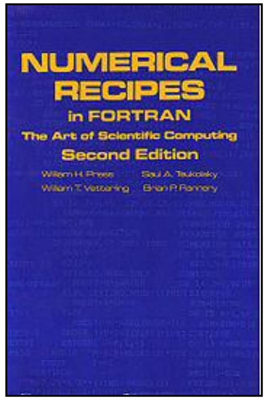
First computer bug, Harvard Mk. II, 1947

Regression testing

	Change structure	New functionality	Change functionality	Change resource use
Add feature	X	X		X
Fix bug	X		X	
Refactor	X			
Optimize	X			X

- Software development can be split into four categories: add feature, fix bug, refactor, optimize.
  - Program output should only change when fixing a bug!
- Regression tests make it easy to check that you did not change the expected output
  - Run the program once and store the expected results
  - For every future run, check that the output is identical to the stored version
- Very important to consider your development: you should only perform one task at a time!

Sharing code & software licenses



- A lot of code on the internet is copyrighted and non-free
  - That it is on the internet does not mean you can use it for free
  - Code in books are also typically copyrighted and non-free
- To share your code with others, you should supply them with a license
- Two main types of open source licenses:
  - **Permissive** (MIT, BSD, etc.): Code can be changed and incorporated into closed source (commercial) without having to share changes to the code
  - **Protective** (GPL, etc.): All code changes must be available to anyone who has your program
- Data can often be released under suitable Creative-Commons licenses, http://creativecommons.org/

Inspired by talk by Johan Seland, 2013 winter school

Summary

- Floating point is extremely tricky
  - Often very difficult to check if the error is due to floating point or implementation issues
- Single precision is often sufficient
  - In many cases there are other errors which completely shadow single precision errors
  - Compting more accurately does not always give better results
- You save a huge amount of time by being thorough
  - Trying to take shortcuts often does not pay off
  - It is often better to do it right from the start