Reproducible Science and Modern Scientific Software Development

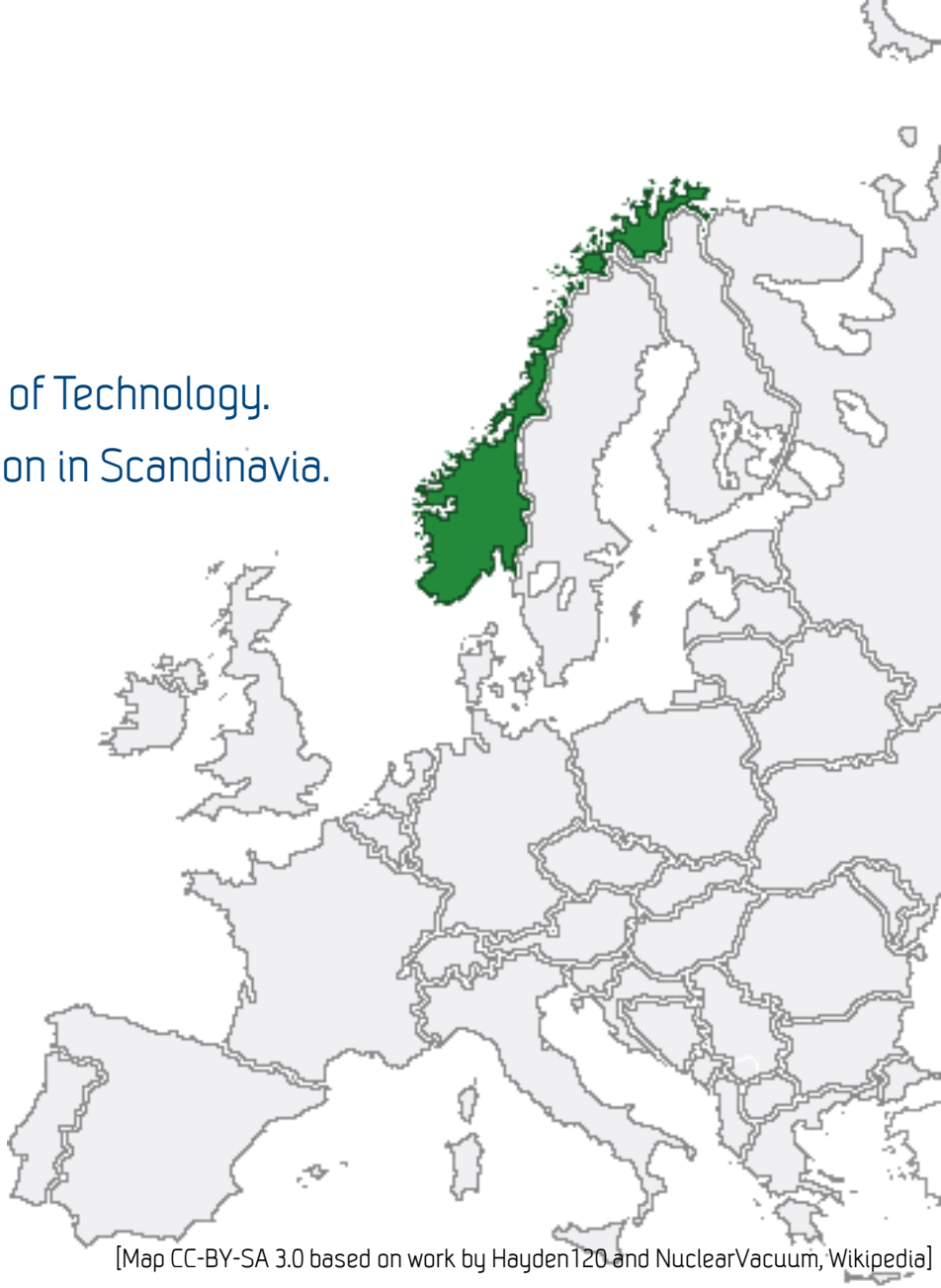
Master Course, 2013,
University of Granada, Spain

2013-04-09

André R. Brodtkorb, Ph.D., Research Scientist
SINTEF ICT, Dept. of Appl. Math.

About SINTEF

- Established 1950 by the Norwegian Institute of Technology.
- The largest independent research organisation in Scandinavia.
- A non-profit organisation.
- Motto: "Technology for a better society".
- Key Figures*
  - 2123 Employees from 67 different countries.
  - 2755 million NOK in turnover
    (about 340 million EUR / 440 million USD).
  - 7216 projects for 2200 customers.
  - Offices in Norway, USA, Brazil, Macedonia, United Arab Emirates, and Denmark.



** Data from SINTEF's 2009 annual report*

[Map CC-BY-SA 3.0 based on work by Hayden120 and NuclearVacuum, Wikipedia]

Outline

- Part 1:
  - What is reproducible science and why should I care?
  - What does software development have to do with it?
- Part 2:
  - Why **not** to share code
  - Best practices for reproducible research
  - Limits of reproducibility
- Part 3:
  - Floating point: It's fun!
  - Parallel computing: It's n times as fun!
  - Reporting performance

Reproducible Research

Reproducible Research

- Are you able to reproduce each graph / result you ever made?
- Reproducible research is about being able to reproduce ones own research *and* that of others.

Who am I, and why am I speaking about this?

- Ph.D. from the University of Oslo
  - Have been working with hyperbolic conservation laws on GPUs, shallow water in particular
    - Getting it 90% correct takes "two weeks"
    - Getting it 99% correct takes "two years"
    - Getting it 100% correct is "impossible"
  - Three month stay at the National Center for Computational Hydroscience and Engineering
- Research Scientist at SINTEF ICT since 2010
  - Getting a research prototype to a commercial code is a major challenge!
- Attended ICERM (Brown University) workshop on reproducible research December 2012
  - Learned a lot from the real experts
- Organized Winter School on Reproducible Science in 2013

Acknowledgements

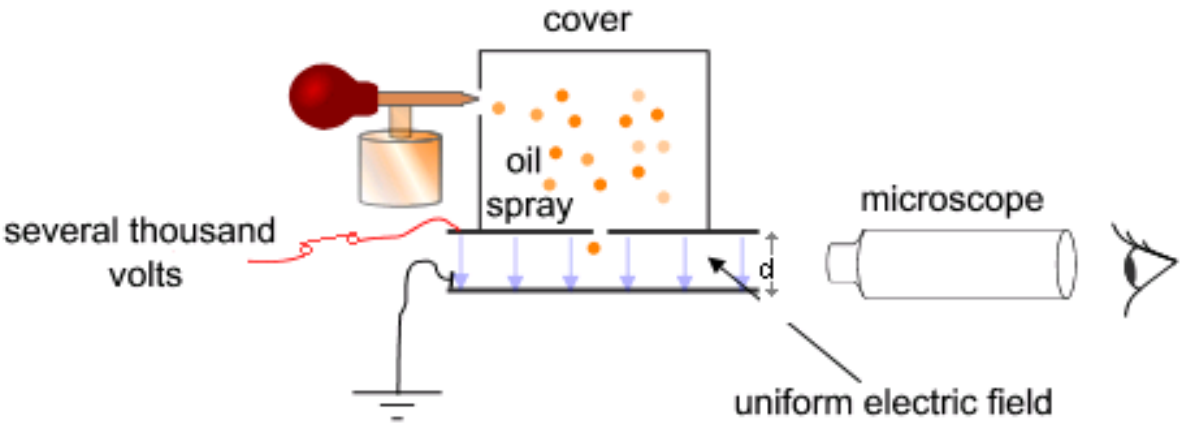
- A large portion of the slides I am presenting are inspired by the ICERM workshop Organizers: David H. Bailey, Jon Borwein, Randall J. Leveque, Bill Rider, William Stein, Victoria Stodden

What is reproducible science?

- Science is a very wide spectrum of disciplines:
  - Chemistry
  - Physics
  - Biology
  - Mathematics
  - Computer science
  - Medicine
  - …
- Reproducible research for is most certainly not the same in any two given disciplines!

Measuring the charge of an electron

- Robert Millikan held a famous experiment published in 1910.
  - Part of the reason for his 1923 Nobel prize in physics



- By varying the charge of the electric field, the (charged) oil particles would rise or fall.
  - Millikan discovered that that charge was discrete, and had a value of $1.5924(17)\times10^{-19}$ C with a very small margin of error
  - The value is today believed to be $1.602176487(40)\times10^{-19}$ C

Oil-drop experiment image, CC-BY-SA 3.0, Theresa Knott

Measuring the charge of an electron

- In 1978 Gerald Holton criticized Millikan, claiming he had manipulated the data
  - Millikan had 175 measurements taken over five months
  - 75 measurements taken over two months published

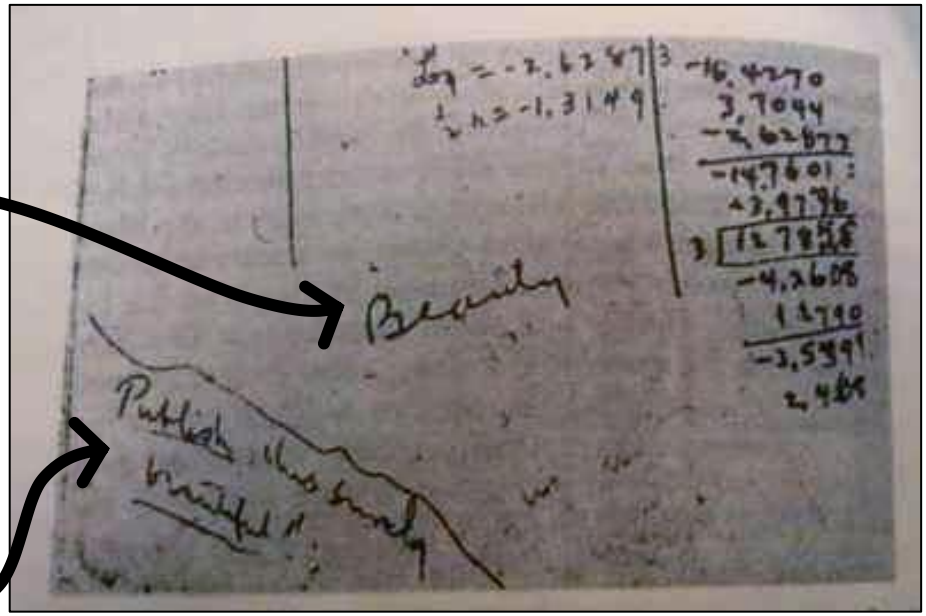
> "It is to be remarked, too, that this is not a selected group of drops, but represents all the drops experimented upon during 60 consecutive days"
> --Millikan, 1913

- Millikan kept a journal of his experiments, which includes notes on the measurements:
  - "Very low, something wrong", "Publish this beautiful one", "Error high will not use", "Too high by 1 ½ %", …

Measuring the charge of an electron

Beauty

Publish this surely Beautiful!!



Measuring the charge of an electron

- Millikan had almost exactly the correct value, but extremely small error margins in published result.

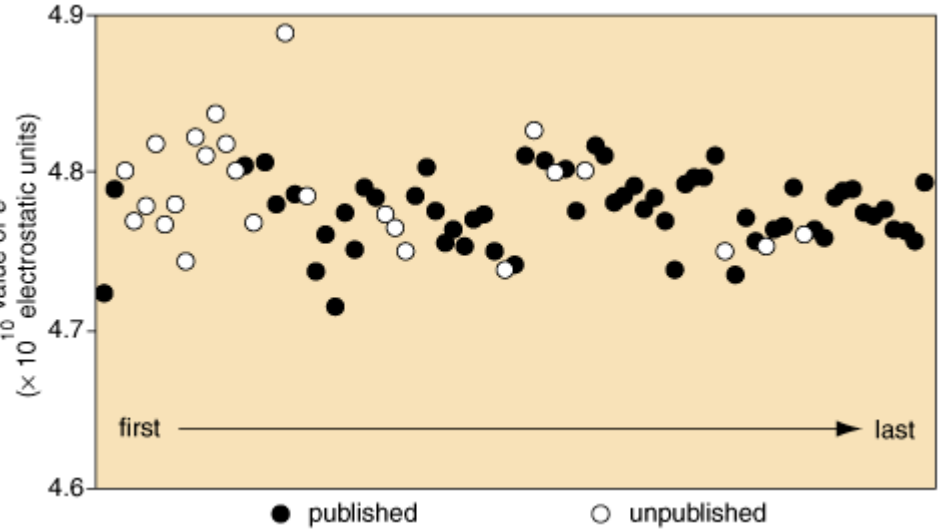


Figure by David Schneider

- A lot of data excluded from publication

Measuring the charge of an electron

We have learned a lot from experience about how to handle some of the ways we fool ourselves. One example: **Millikan measured the charge on an electron** by an experiment with falling oil drops**, and got an answer which we now know not to be quite right**. It's a little bit off because he had the incorrect value for the viscosity of air. **It's interesting to look at the history of measurements of the charge of an electron, after Millikan. If you plot them as a function of time, you find that one is a little bit bigger than Millikan's, and the next one's a little bit bigger than that, and the next one's a little bit bigger than that, until finally they settle down to a number which is higher.**

...

Measuring the charge of an electron

...
Why didn't they discover the new number was higher right away? It's a thing that scientists are ashamed of - this history - because it's apparent that people did things like this: **When they got a number that was too high above Millikan's, they thought something must be wrong - and they would look for and find a reason why something might be wrong. When they got a number close to Millikan's value they didn't look so hard. And so they eliminated the numbers that were too far off, and did other things like that...**

--Richard Feynman

Making up data

- The Sudbø case (2006)
  - Made up 900 persons with metical history for cancer research (the use of anti-inflamatory drugs were claimed to reduce risk of mouth cancer).
  - The data was supposedly from a named patient database (which had not yet opened…)
  - Articles retracted (including in The Lancet), his wife and brother were co-authors on several retracted papers, lost his Ph.D., …
- Lancet editor: the biggest scientific fraud conducted by a single researcher

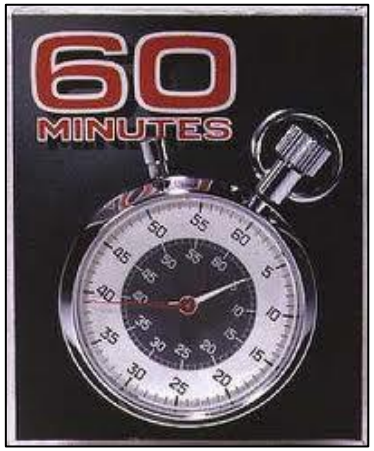
Duke University: Breakthrough in cancer research gone bad

"Anil Potti is accused of **falsifying data** regarding the use of microarray genetic analysis for personalized cancer treatment, which was published in various prestigious scientific journals."
[Wikipedia on Anil Potti]

- "As of February 2012, of more than 120 peer-reviewed publication "Published Papers", ten scientific papers authored by Potti and others retracted"

- Patients treated with experimental treatments (personalized treatment)
  - Based on their genes, one would find the most suitable drug for cancer treatment

- 60 minutes documentary: http://www.cbsnews.com/video/watch/?id=7398476n



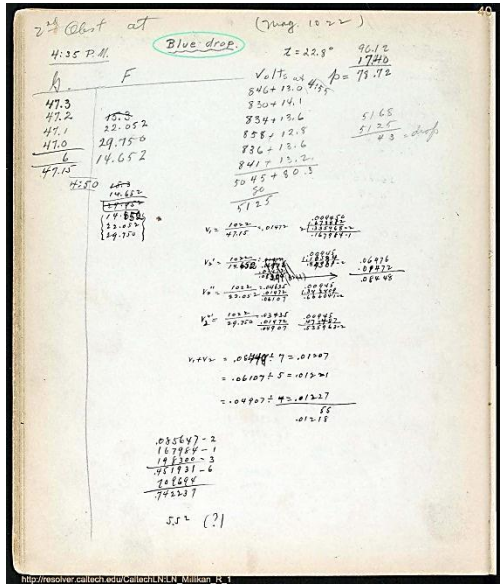
Irreproducible science

What is the difference between Potti, Sudbø and Millikan?

- All manipulated data used in publications in some way
- Millikan was right, the others were wrong

What does this have to do with reproducible science?

- Millikans notebook was important to document his published and unpublished results
- His results have been reproduced again and again
- Other researchers discovered the errors in Potti and Sudbø by studying their data



Tenfold increase in scientific research papers retracted for fraud since 1975: Two thirds retracted for scientific misconduct, not error

Findings in six of 53 landmark studies in cancer research can be confirmed

Source:

- Tenfold increase in scientific research papers retracted for fraud, Alok Jha, The Guardian, Monday 1 October 2012
- Drug development: Raise standards for preclinical cancer research, C. Glenn Begley and Lee M. Ellis, Nature 483, 2012

The scientific method and reproducible science

Roger Bacon
1214-1294

1267:
Alchemist who proposes ideas of observation, hypothesis, experimentation, and external verification

Francis Bacon
1561-1626

1620:
Important for the idea of the "scientific method"

Robert Boyle
1626-1691

1665:
"enough information must be included to allow others to independently reproduce the finding"

Inspired by slides presented by Victoria Stodden at ICERM, 2012

Reproducible Research Movement

- 1991: Professor Jon Claerbout (Stanford) requires theses of his students to be reproducible (geophysics)

- A lot of researchers at other institutions see similar problems, and the idea spreads
  - Randy Leveque, Sergey Fomel, David Donoho, Kai Diethelm, …

- Scandals in some disciplines (cancer research, genomics,…) cause policy changes and huge (local) awareness

- Special issue on in Computing in Science and Engineering (2009)

Reproducible Research Movement

Increasing awareness around 2010 with a large number of workshops:

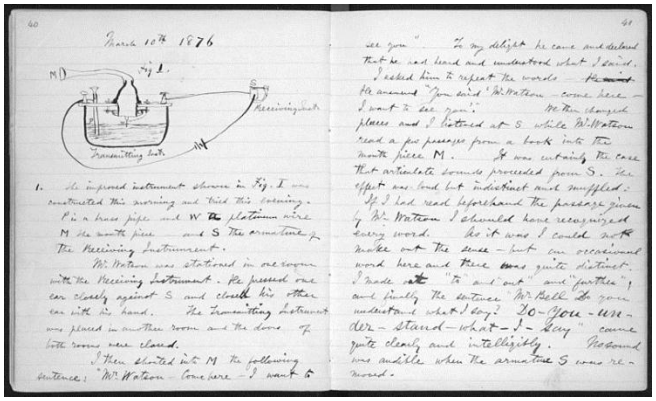
- **2009:**
  Yale 2009: Roundtable on Data and Code Sharing in the Computational Sciences
- **2011:**
  SIAM CSE 2011: Verifiable, Reproducible Computational Science
  AAAS 2011: The Digitization of Science: Reproducibility and Interdisciplinary …
  ENAR International Biometric Society 2011: Panel on Reproducible Research
  SIAM Geosciences 2011 Reproducible and Open Source Software in the Geosciences
  AMP / ICIAM 2011 Community Forum on Reproducible Research Policies
  AMP 2011 Reproducible Research: Tools and Strategies for Scientific Computing
- **2012:**
  ICERM 2012 Reproducibility in Computational and Experimental Mathematics
  Supercomputing 2012, Reproducibility of Results
- **2013:**
  eVITA Winter School on Reproducible Science and Modern Scientific Software Development

Reproducible Research Movement

- Important journals start changing their policies
  - Allow for supplementary material
  - Require statements on reproducibility
  - …
- Image Processing On Line
  - All published algorithms are available as an interactive web-service
  - http://www.ipol.im/

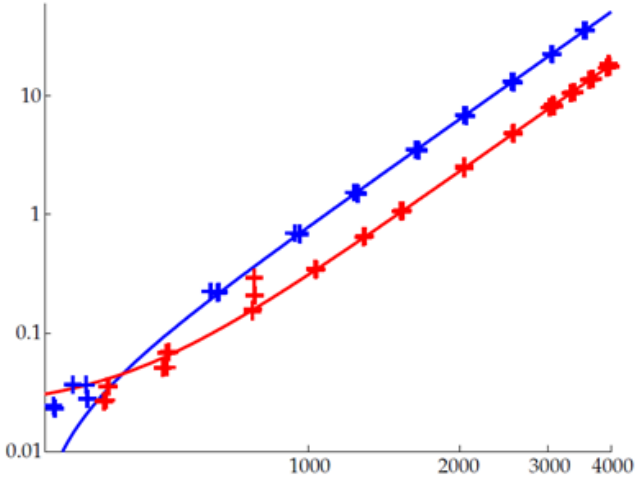
Reproducible Research Movement

- Reproducible science is thoroughly embedded in the scientific work-flow in many disciplines
  - An experimental scientist will keep a lab notebook over all experiments.
  - The notebook will enable him to reproduce the experiment

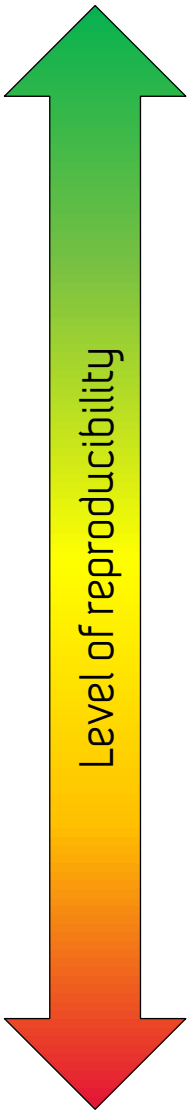


Lab notebook of Graham Bell, 1876

- In computational science, however, this lab notebook has lost its place in the natural work-flow:
  - Possible workflow: write a program, create a graph, change the program, create a graph, etc.
    - Changes can be small: a parameter change, domain size change, …
  - The graphs are kept for publication
  - The program changes are forgotten…



50 Shades of Reproducible Research



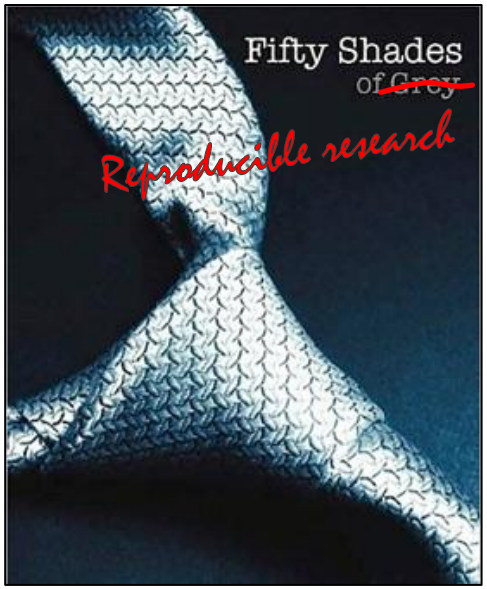
Interactively reproducible: All figures, tables, and data in a paper can be reproduced with the original data, or I can supply my own data through a web service and get new graphs and results.

Turn-key reproducible: All figures, tables, and data in a paper can be reproduced by compiling and running the program at the click of a button.

Publicly reproducible: All figures, tables, and data in the paper would be possible to reproduce for someone else, but they'd have to manually compile the program and all of its dependencies.

Privately reproducible: All figures, tables, and data in a paper would be possible to reproduce, albeit with a great deal of effort, by myself or one of my co-authors.

Irreproducible: It would not be possible for me to recreate the results I published.



"The person most likely to reproduce
your work is your own future self"
-- Sergey Fomel at ICERM workshop

- The most convincing reason for me to be reproducible,
  is that somewhere down the line:
  - I will have to re-do the graph with different axes because a reviewer / supervisor asked,
  - I will have to reinterpret the data for an updated conclusion,
  - I will write a journal paper based on a conference paper,
  - I will (hopefully☺) write a book or book chapter based on previous results,
  - …

More reasons for reproducible research

- Reproducible papers are cited almost 5 times as often!
- Easier to get collaborations started
- Start-up-time for Masters and Ph.D. students cut down from months/years to weeks.
- Fraudulent research is not reproducible☺, reproducible research should not be fraudulent.
- ACM journals starting with "stamps of approval" for reproducible research
- …

[1] Code Sharing Is Associated with Research Impact in Image Processing, Patrick Vandewalle, Computing in Science & Engineering, 2012

The importance of reproducible research

> Computational science cannot be elevated to a third branch of the scientific method until it generates routinely verifiable knowledge.
> --Donoho, et al. 2009

Science is the art of building pyramids of knowledge, one small block of knowledge at a time

"If I have seen further it is by standing on the shoulders of giants."
-- Isaac Newton

All Gizah Pyramids, CC-BY-SA 2.0, Ricardo Liberato

Inspired by slides presented by Sergey Fomel and Victoria Stodden at ICERM, 2012

A chemist treats the laboratory with thorough respect, having strict security procedures, careful note taking during experiments, etc. So should we also treat our computers, as it is the laboratory of computational science.

Modern Scientific Software Development

Commercial versus academic software development

- Commercial software development is driven by the motive of **making money**
  - **Inherent requirement to quality**: Customer must feel that the software is worth the money it costs

- Academic software development is driven by the motive of **publishing papers**
  - **Inherent requirement to work at least once**. After the publication has been completed, further research and development can rapidly die out.

The sad truth

"In academia, software quality, user interfaces, documentation, testing and reproducibility, will all be sacrificed at the altar of publications"

Working reproducibly must become a central part of your everyday work cycle: it is not enough to think of it as a post publication step (which never happens, anyway…)

The sad truth

- Many (Most?) scientists have not been taught software development practices
  - Researchers are taught as physicists, chemists, biologists, etc.
  - "**Self-taught** software developers" with **little or no formal training**
  - Even many computer scientists have poor work habits

- Many hold the belief that software development is something you can pick up, whilst {mathematics, chemistry, …} is something that you must study.
  - Might be true for simple Matlab programming
  - But it's also like lab safety: not something you can pick up for advanced and demanding experiments

- Academic software has a tendency to grow into huge monsters over the years
  - Example: NASTRAN: Developed at NASA in late 1960-ies. 1 million lines of code, Fortran, still sold commercially.
  - **Software quality: Pay me now or pay me later**

Scientific Software Development

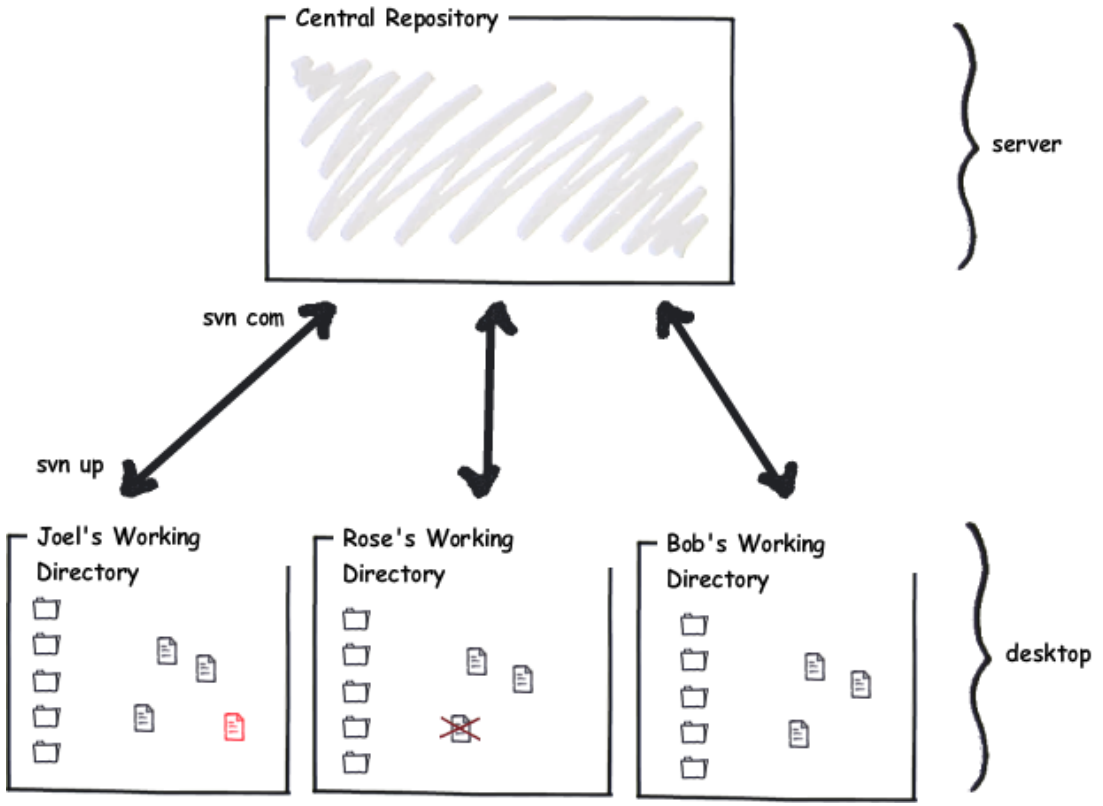
- Commercial software development has progressed rapidly:
  - Version control, issue trackers, …
  - Extreme programming, pair programming, scrum, …
  - Test driven development, continuous builds, …
  - Dedicated testing and quality assurance teams
  - …
- Many, but perhaps not all, of these tools and methodologies are great aids in reproducible research.
  - Some tools are also specifically created for scientific software: lab notebooks ala Ipython, Sage, etc.

Version control

- **Version control**
  - Subversion, mercurial, git, …
  - A simple kind of backup system.
  - Stores what a file looks like at a given time
  - By regularly checking in changes to your program, you get a history of the source code development.
  - Also a tool for collaboration

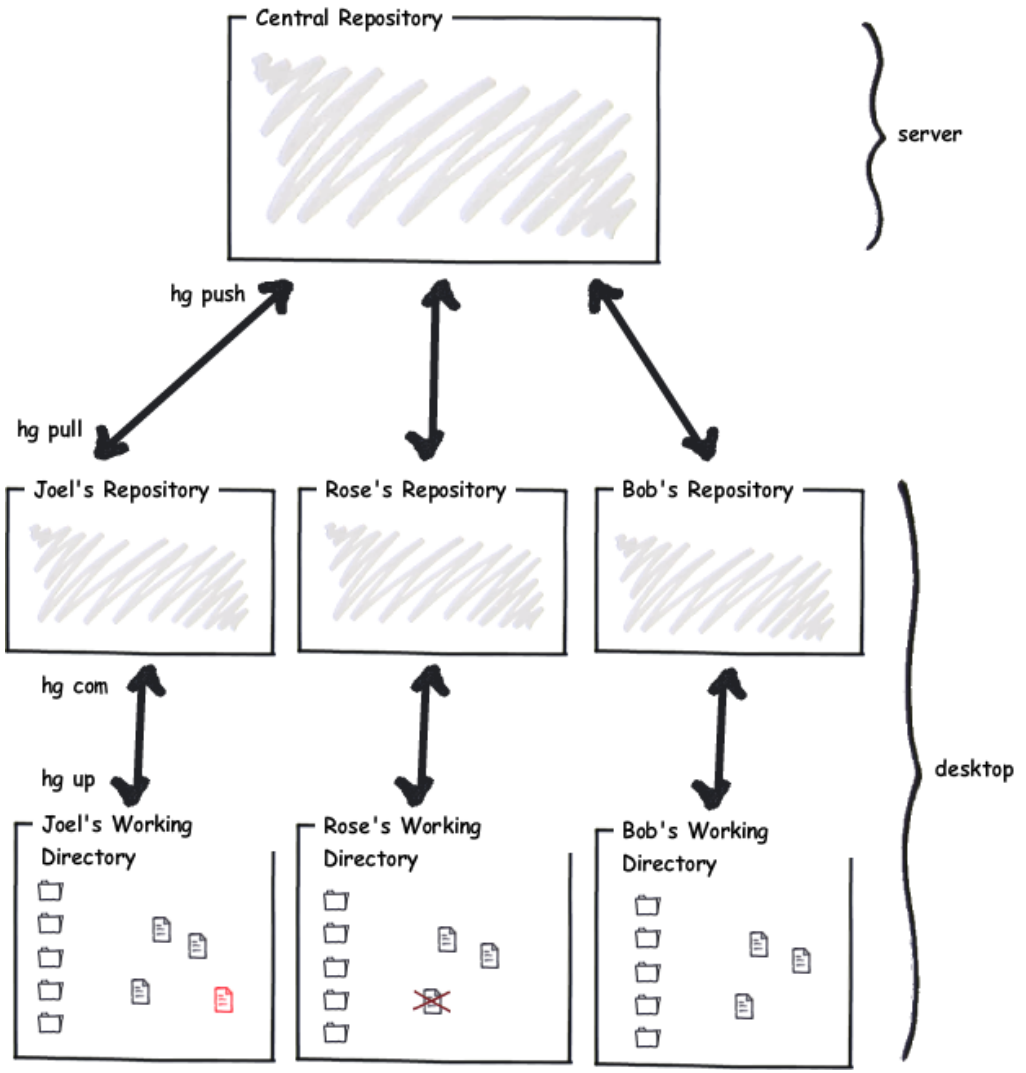
Centralized version control

- Traditional CVS, subversion, etc.
- A central repository is used which everyone synchronizes against
- Breaks down every time you are offline
- Notoriously difficult to merge differences from different users



Distributed Version Control

- Distributed version control is like a layered version control: mercurial, git, bazaar, etc.
- Commit often to your local repository
- Synchronize with the central repository
- Much better at handling merge conflicts (also because work habits change)



Version control and provenance

- Provenance is imperative for reproducible research
  - Provenance is the history that describes what has happened to the data and source code used to produce results

- In its simplest form:
  What version of the software and what version of the data produced the results?

- The use of version control software gives provenance for source code
  - More difficult with dependencies (version of operating system, libraries, etc.)
  - More difficult with large data-sets, as most version control systems are not really designed for it

- Important: must be able to match output results to a given version of the data and source code

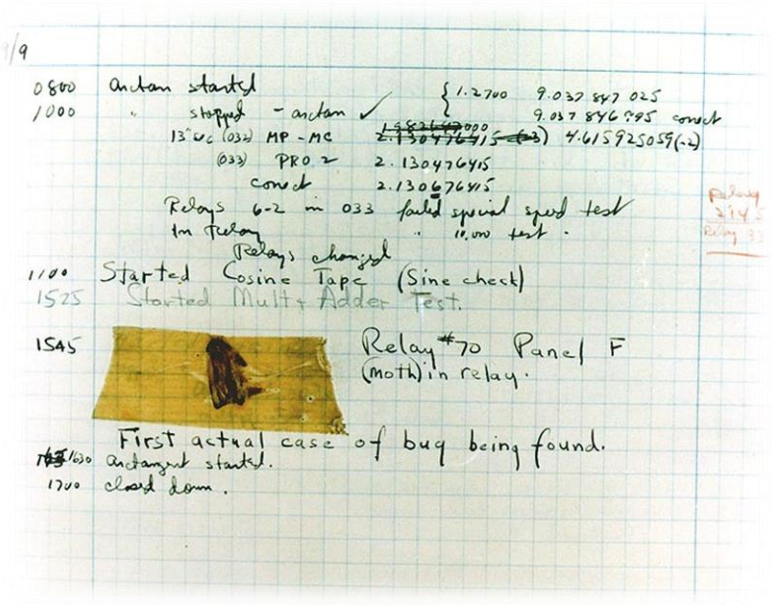
Issue Trackers & wikis

- **Issue trackers**
  - Bugzilla, trac, …
  - A kind of registry for bugs and planned developments for your source code
  - Makes sure you have a history of fixed issues, and that you don't forget an important bug.

- **Personal wiki**
  - Mediawiki, …
  - Enables you to keep track of ideas and thoughts in a structured way

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!

Continuous builds

- One danger of using testing is that tests quickly are disabled or ignored as the paper deadline approaches…

- Continuous builds tackle this by automatically testing each new revision in the source code repository, and creating a report on the results

- Makes it easy to get a history of development, and discover when and why something suddenly changed.

- http://jenkins-ci.org

Summary

- Reproducible research does not come for free
  - The pros outweigh the cons: pay me now or pay me later
  - Requires that we include reproducibility in our daily work-flows

- Modern software development practices are important tools
  - Without version control, it is extremely difficult to have reproducible research
  - Many techniques and tools make software development faster, less error prone, and makes it easy to collaborate

Further references

- **ICERM Reproducibility in Computational and Experimental Mathematics**
  http://icerm.brown.edu/tw12-5-rcem
  http://wiki.stodden.net/ICERM_Reproducibility_in_Computational_and_Experimental_Mathematics

- **Reproducible Research: Tools and Strategies for Scientific Computing**
  http://stodden.net/AMP2011/

- **Best Practices for Scientific Computing**
  Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Katy Huff, Ian M. Mitchell, Mark Plumbley, Ben Waugh, Ethan P. White, Paul Wilson
  (Submitted on 1 Oct 2012 (v1), last revised 29 Nov 2012 (this version, v3))
  http://arxiv.org/abs/1210.0530

Reproducible Science and Modern Scientific Software Development

Master Course, 2013,
University of Granada, Spain

2013-04-09

André R. Brodtkorb, Ph.D., Research Scientist
SINTEF ICT, Dept. of Appl. Math.

Outline

- Part 1:
  - What is reproducible science and why should I care?
  - What does software development have to do with it?
- Part 2:
  - Why **not** to share code
  - Best practices for reproducible research
  - Limits of reproducibility
- Part 3:
  - Floating point: It's fun!
  - Parallel computing: It's n times as fun!
  - Reporting performance

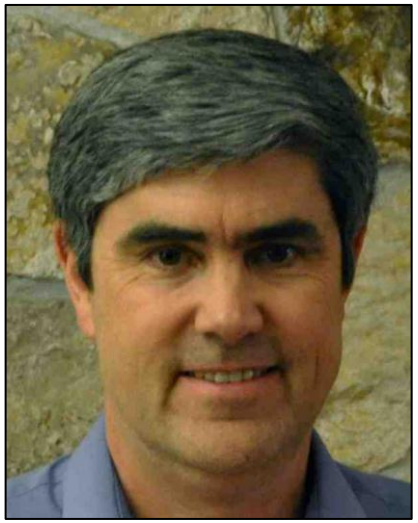
Part 2

This part should equip you with:

- Reasons why you should try to be reproducible and share your code
- A check-point list of best practices to consider
- An overview of some situations in which it is difficult to get reproducible results

Top ten list of why not to share code [1]

- A list that would have been better presented by Randy Leveque



Professor Randy Leveque, University of Washington

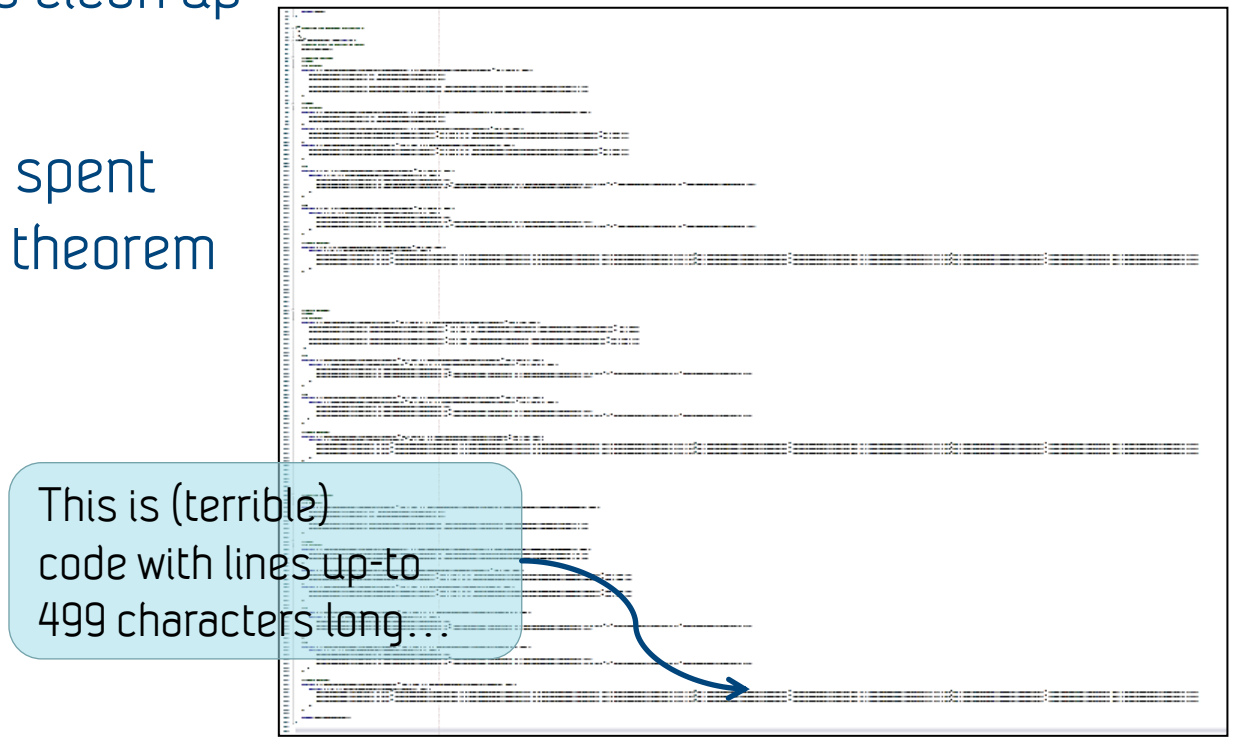
- Imagine a world in which mathematical papers contain the same amount of information as computational papers have today:
  - Papers contain lemmas, theorems, corollaries
  - No proofs are required or expected

- Then some people start demanding proofs to be published.
  What would people say?

[1] from Top Ten Reasons to Not Share Your Code, Randall J. Leveque, 2012
http://jarrodmillman.com/talks/siam2011/ms148/leveque.pdf

Top ten list of why not to share code

1. The proof is too ugly to show anyone else.
   - It's a waste of time to clean up the proof: it's very specific and not worth while to clean up
   - My time is better spent publishing a new theorem



Top ten list of why not to share code

2. I didn't work out all the details.
   - It applies for the examples I use in the paper, that's enough
   - It won't really work for all corner cases, but that's not important

3. I didn't actually create the proof myself, my student did.
   - And the student went into industry so I don't really have it anyway
   - But he was a good student, and I'm pretty sure it's correct

Top ten list of why not to share code

4. Giving the proof to my competitors would be unfair to me.
   - If I give out my proof, anyone can do further research
   - I should be the one to get papers out of this: after all, it's my proof

5. The proof is valuable intellectual property.
   - I would be stupid to give it away:
     I might be able to commercialize it some time in the future

Top ten list of why not to share code

6. Including proofs would make the paper much longer.
   - Journals wouldn't want to publish it, and who would want to read it?

7. Referees would never agree to check proofs.
   - It's already difficult to get reviewers and reviews on time

Top ten list of why not to share code

8. The proof uses sophisticated hardware/software that most readers and referees don't have.
   - If they can't execute the proof, why should they care to get it?

9. My proof relies on other unpublished (proprietary) proofs.
   - It doesn't really help that they have my proof, they don't know if the dependencies are correct anyway

Top ten list of why not to share code

10. Readers who have access to my proof will want user support.
    - And I really don't want to be pestered by people actually using my work

- Are computer codes fundamentally different from mathematical proofs?
- Are computer codes exempt from the scientific method?

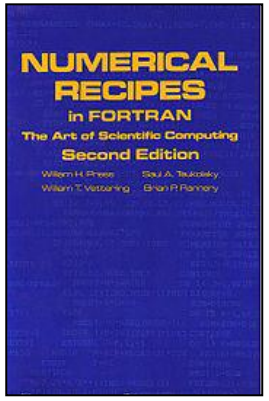
Why you should share your code anyway

"An article about computational result is advertising, not scholarship. The actual scholarship is the full software environment, code and data, that produced the result."
--Jon Claerbout [1]

"Personally I think people get hung up too much on the fact that it's hard to insure others can run the code, and should focus more on providing a full record of the research methodology."
-- Randy Leveque [2]

[1] WaveLab and Reproducible research, J. B. Buckheit and D. L. Donoho, 1995
[2] Top Ten Reasons to Not Share Your Code, Randy Leveque, 2012

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

Inspired by talk by Johan Seland, 2013 winter school

Best Practices

Best Practices Countdown [1]

1. **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
   - Human memory is extremely limited: "a program should not require its readers to hold more than a handful of facts in memory at once"

Bad: Order of arguments
```
def rect_area(x1, y1, x2, y2):
```

Good: Use natural syntax
```
def rect_area(point1, point2):
```

Bad: Too many parameters
```
def complex_func(x1, x2, x3, …, xn):
```

Good: Use structures
```
def rect_area(arguments):
```

   - Human effort is limited: "all aspects of software development should be broken down into tasks roughly an hour long"

[1] Best Practices for Scientific Computing, Greg Wilson et al., 2012, arXiv:1210.0530

Best Practices Countdown

2. Automate repetitive tasks

- even the most careful researcher will lose focus while doing this and make mistakes.
- "use a build tool to automate the scientific workflows"
- write (python) scripts for repetitive tasks

Best Practices Countdown

3. 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
- Automatically record versions of software and data, and parameters used to produce results (see also point 2)

4. Make incremental changes

- Do not plan for months or years of development: Plan for one week or so, partitioned into small tasks (which can be solved using one hour long sessions at a time)

Best Practices Countdown

5. Use version control

- Learn how to see the difference (diff) between two versions of the software, and how to revert changes
- Learn how to use version control for collaboration
- "everything that has been created manually should be put in version control"
- Use meta-data to describe binary data

6. Don't repeat yourself

- "every piece of data must have a single authoritative representation in the system"
- "code should be modularized rather than copied and pasted"
- "re-use code instead of rewriting it" (matrix inversion, etc.)

Best Practices Countdown

7. Plan for mistakes (1/2)

- "add assertions to programs to check their operation"

```
def bradford_transfer(grid, point, smoothing):
        assert grid.contains(point),
                'Point is not located in grid'
        assert grid.is_local_maximum(point),
                'Point is not a local maximum in grid'
        assert len(smoothing) > FILTER_LENGTH,
                'Not enough smoothing parameters'
        ...do calculations...
        assert 0.0 < result <= 1.0,
                'Bradford transfer value out of legal range'
        return result
```

- Assertions are "executable documentation, i.e., they explain the program as well as checking its behaviour"

Best Practices Countdown

7. Plan for mistakes (2/2)

- Use automated testing
  - Regression testing => has something changed
  - Verification testing => does the code produce known correct/analytical solutions?
  - Use an interactive debugger instead of print-statements

8. 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"

Best Practices Countdown

9. Document design and purpose, not mechanics.

- Code should be written for humans and not require any documentation by itself: "document interfaces and reasons, not implementations"
- Remove unreadable code: "refactor code instead of explaining how it works"

10. Collaborate.

- Make others read your code: "code reviews are the most cost-effective way of finding bugs in code"
- "use pair programming when bringing someone new up to speed and when tackling particularly tricky problems"
- Collaborators can interpret results in completely different ways

"**Research suggests** that the **time cost** of
implementing these kinds of tools and approaches
in scientific computing **is almost immediately**
offset by the gains in productivity of
the programmers involved"

Best practices overload

11. Take notes

- Use an issue tracker, blog, wiki or physical lab notebook for notes and ideas
- Ideas come up all the time, and are written on post-its etc. and easily forgotten.

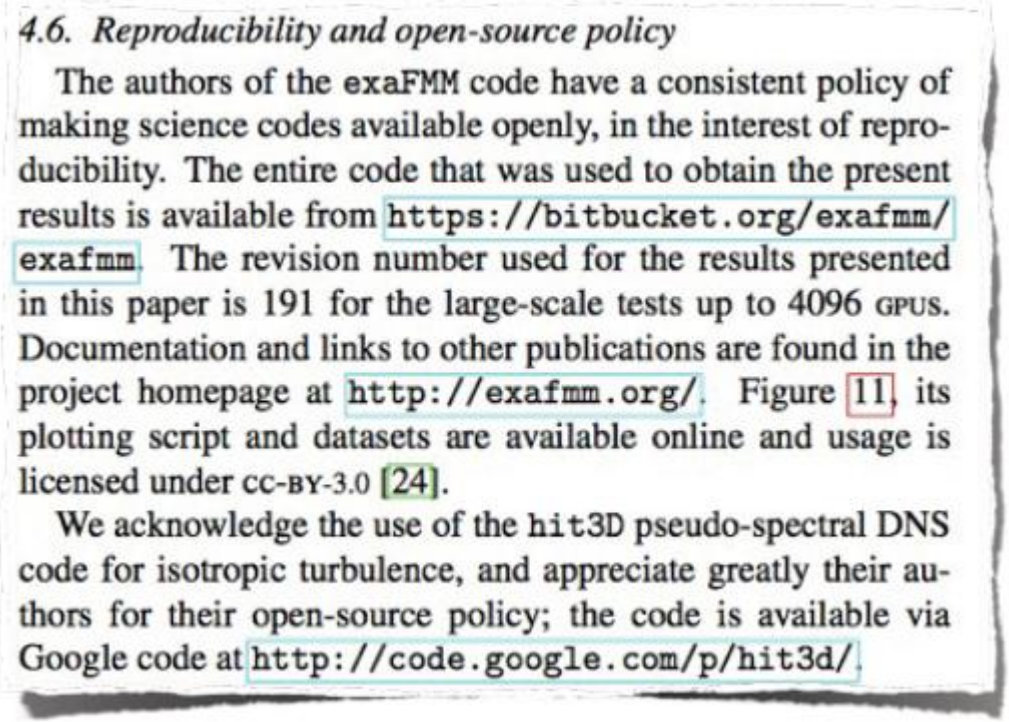
12. Keep it simple, stupid

- 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

Best practices overflow

13. **Write statements on reproducibility** [1]

- At the end of papers you publish, try to write if and how the results are reproducible.



4.6. Reproducibility and open-source policy

The authors of the exaFMM code have a consistent policy of making science codes available openly, in the interest of reproducibility. The entire code that was used to obtain the present results is available from `https://bitbucket.org/exafmm/exafmm`. The revision number used for the results presented in this paper is 191 for the large-scale tests up to 4096 GPUs. Documentation and links to other publications are found in the project homepage at `http://exafmm.org/`. Figure 11, its plotting script and datasets are available online and usage is licensed under CC-BY-3.0 [24].

We acknowledge the use of the hit3D pseudo-spectral DNS code for isotropic turbulence, and appreciate greatly their authors for their open-source policy; the code is available via Google code at `http://code.google.com/p/hit3d/`.

[1] Reproducibility PI Manifesto, Lorena A. Barba, http://faculty.washington.edu/rjl/icerm2012/Lightning/Barba_Manifesto.pdf

Limits of reproducible research

Definitions of reproducible research

[1]Reproducibility in Computational and Experimental Mathematics, Workshop report, 2012

Reviewed Research:

The descriptions of the research methods have been independently assessed and the **results judged credible.** (This includes both traditional peer review and community review, and **does not necessarily imply reproducibility.**)

Replicable Research:

Tools are made available that would allow one to **duplicate the results of the research, for example by running the authors' code** to produce the plots shown in the publication. (Here tools might be limited in scope, e.g., only essential data or executables, and might only be made available to referees or only upon request.)

Confirmable Research:

The **main conclusions** of the research **can be obtained independently** without the use of software provided by the author. (But using the complete description of algorithms and methodology provided in the publication and any supplementary materials.)

Auditable Research:

Sufficient records (including data and software) have been archived so that the **research can be defended later if necessary**. The archive might be private, as with traditional laboratory notebooks.

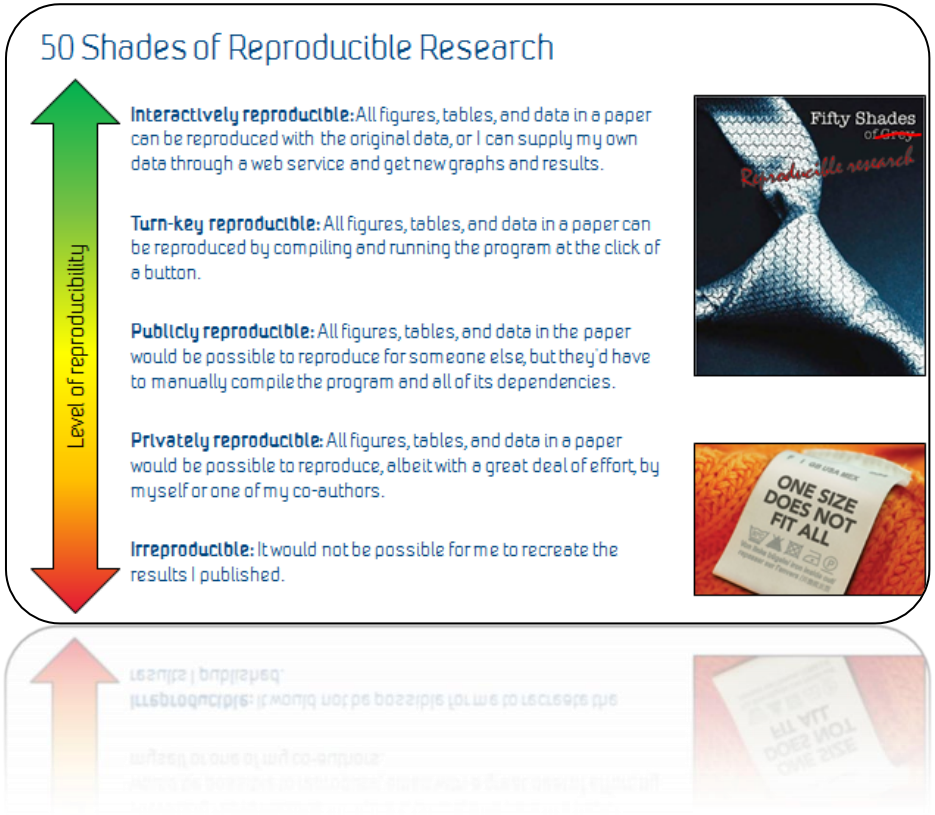
Open Research: Well-documented and fully open tools are publicly available (e.g., all data and open source software) that would allow one to (a) fully audit the computational procedure, (b) duplicate the results of the research, and (c) extend the results or apply the method to new problems.

Saint Graal de la légende du Roi Arthur et des Chevaliers de la Table Ronde, Alfred W Pollard, 1917

The limits of reproducible research

- The limits of reproducible research depends on the type of reproducible research we are discussing

- What hinders
  - "Private reproducibility"?
  - "Public reproducibility"?
  - "Turn-key reproducibility"?
  - "Interactive reproducibility"?



Problems in reproducibility

Supercomputer simulations

- Requires special hardware
- Rerunning experiments not always feasible
- Changing number of nodes changes the domain decomposition, which affects the answer…
- Parallel computing is terribly irreproducible (floating point)
- High performance computing often comes at the expense of reproducibility

Argonne National Laboratory, IBM Blue Gene P, CC-BY-SA 2.0

Problems in reproducibility

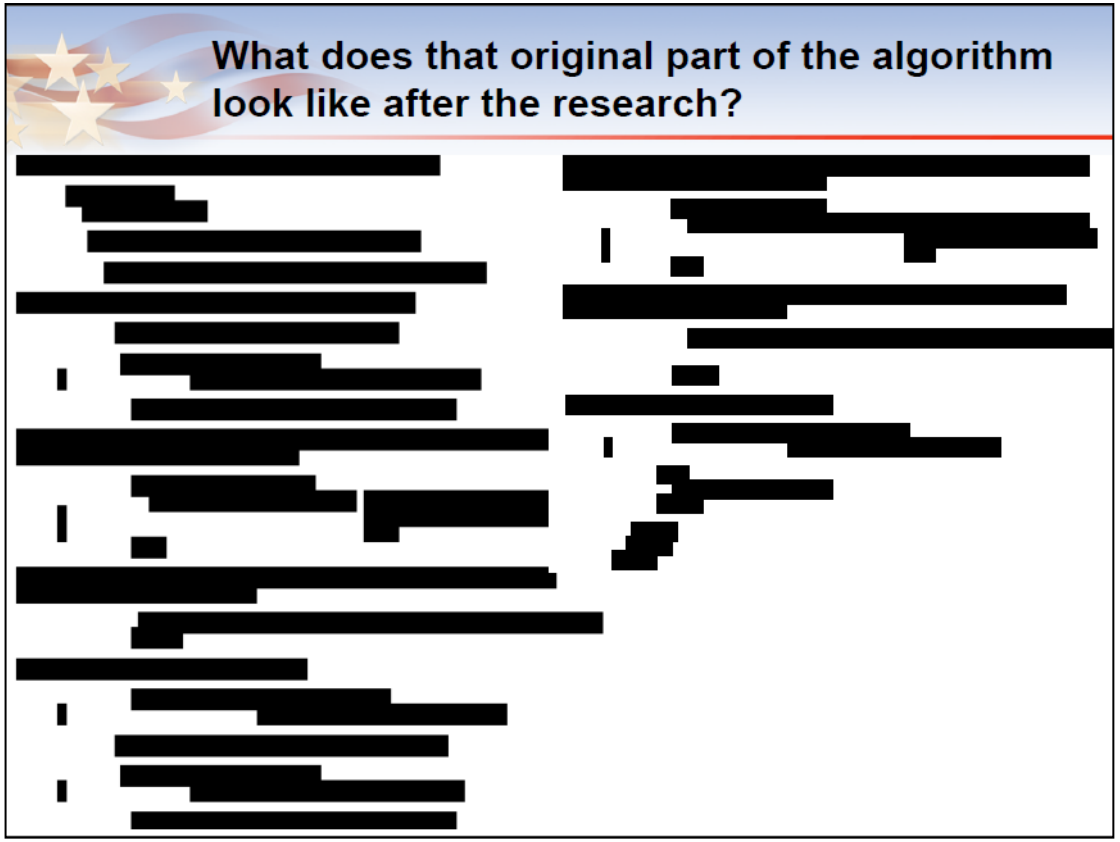
Graphics Processing Units

- GPUs are extremely parallel processors with all the pitfalls of parallel computing
- GPUs change rapidly, and I can't get a five year old GPU anymore
- Programming languages and tools change extremely fast
- A different floating point model than many CPUs (more accurate)

Problems in reproducibility

Legal concerns

- Patent laws
- Export limitations
- Intellectual property rights
- Licenses
- Research performed at commercial institutions
- …

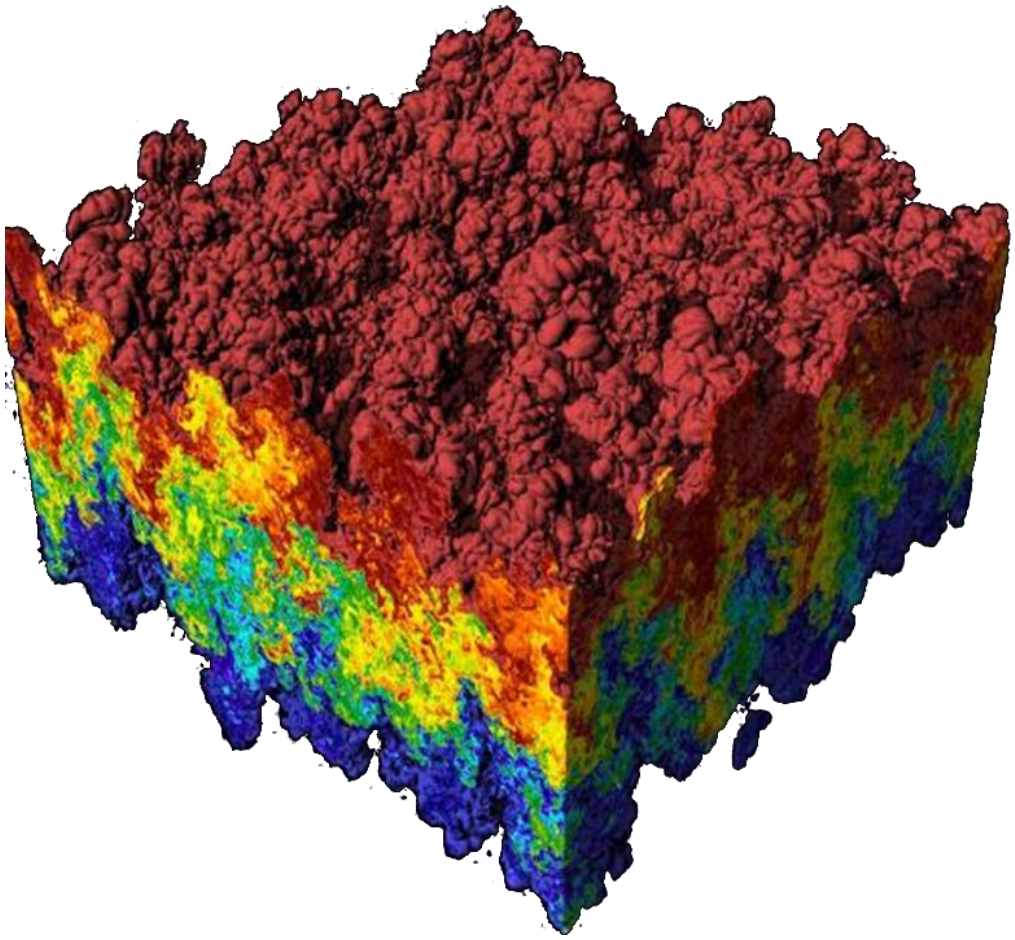


Censored slide from Bill Rider, "What does it take to do reproducible computational science? What stands in our way?", ICERM 2012.

Problems in reproducibility

Visualization

- Many visualization tools are used interactively, and therefore hard to use reproducibly



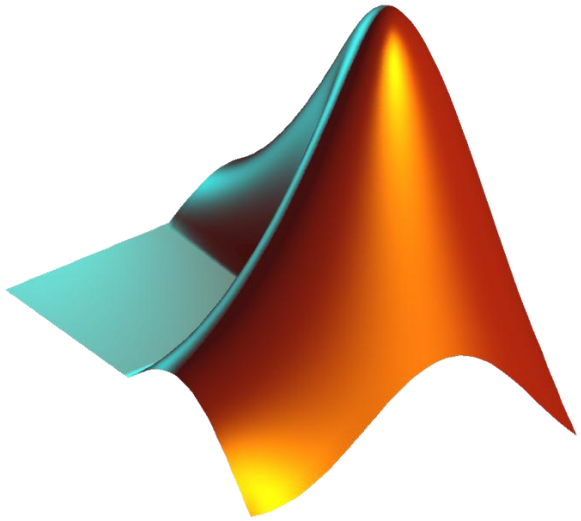
Problems in reproducibility

Data archiving

- We have version control for text-like documents
- We have very little for managing data sets
- Data sets must have meta-data which is manually entered

Archives, Archivo-FSP, CC-BY-SA 3.0

Problems in reproducibility



Software licenses

- Difficult to combine with virtual machines & the cloud

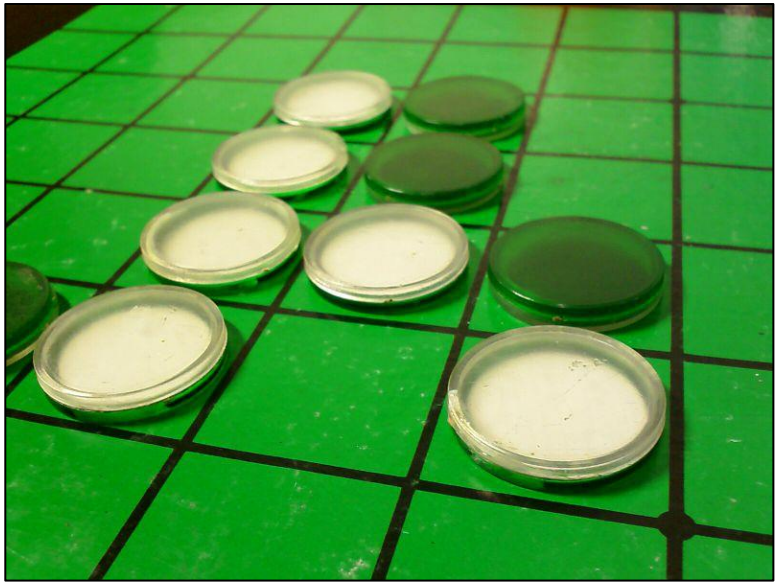
Old software

- My software only run under AIX / Windows NT / … and I can't get hold of hardware that it will install under…
- My software relies on a specific behavior only found in GCC v. 2.81
- My software requires a commercial compiler which only runs on Windows NT / AIX / …

Problems in reproducibility

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

Software versions and compilers

- Versions of software and libraries change, so do their features
- Compiler flags (optimization, --fast-math, etc.) change the program and also the result

Summary

- Computational codes are a lot like mathematical proofs and we should aim at publishing whenever possible
- The essence of the best practices is: "Be methodical, be thorough, be honest".
- Different situations have different requirements to disclosure and reproducibility
- It can be difficult to be reproducible in some situations.

Further Reading

- Randy Leveque, Top Ten Reasons to Not Share Your Code (and why you should anyway), Randy Leveque, 2012, http://faculty.washington.edu/rjl/pubs/topten/

- Reproducibility in Computational and Experimental Mathematics, Workshop report, 2012

- Best Practices for Scientific Computing
  Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Katy Huff, Ian M. Mitchell, Mark Plumbley, Ben Waugh, Ethan P. White, Paul Wilson (Submitted on 1 Oct 2012 (v1), last revised 29 Nov 2012 (this version, v3)) http://arxiv.org/abs/1210.0530

Reproducible Science and Modern Scientific Software Development

Master Course, 2013,
University of Granada, Spain

2013-04-09

André R. Brodtkorb, Ph.D., Research Scientist
SINTEF ICT, Dept. of Appl. Math.

Outline

- Part 1:
  - What is reproducible science and why should I care?
  - What does software development have to do with it?

- Part 2:
  - Why **not** to share code
  - Best practices for reproducible research
  - Limits of reproducibility

- Part 3:
  - Floating point: It's fun!
  - Parallel computing: It's n times as fun!
  - Reporting performance

Outline

- Floating point: It's fun!
- Parallel computing: It's n times as fun!
- Reporting performance

Floating point [1]

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

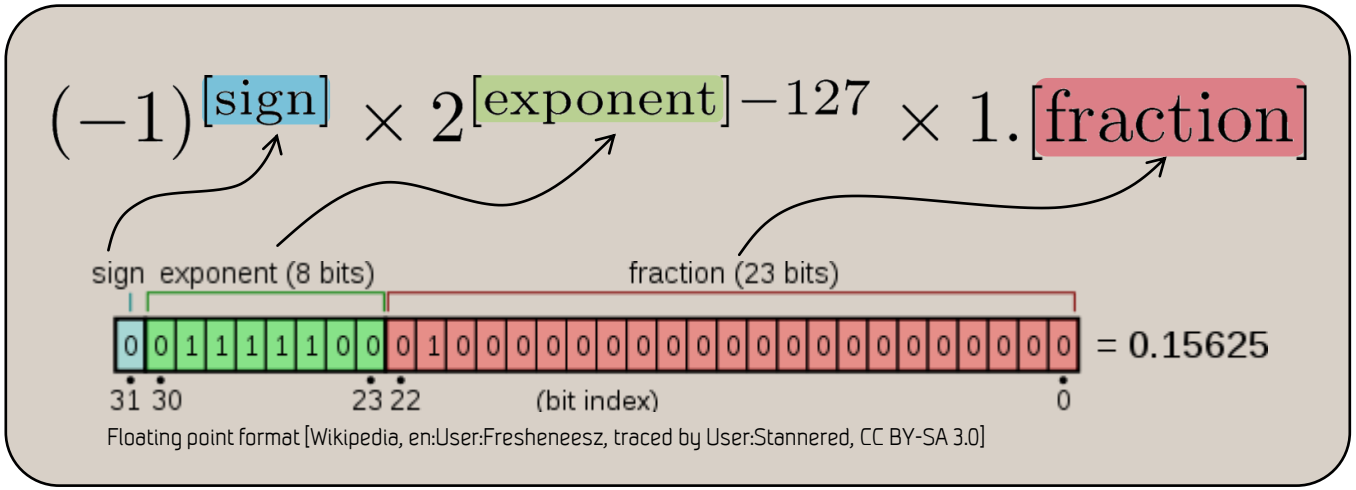
Intel Pentium with FDIV bug,
Wikipedia, user Appaloosa,
CC-BY-SA 3.0

"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

A floating point number on a binary computer

- Floating point numbers are represented using a binary format:



- 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
- 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
```

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

The long double – a real bastard

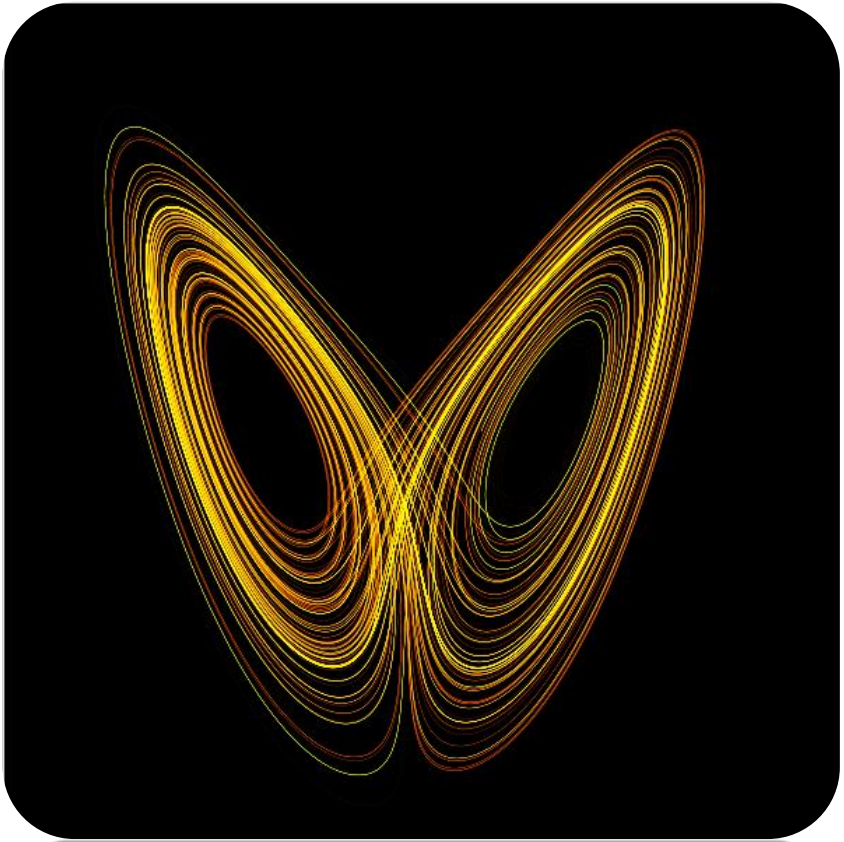
- What is a long double?
  - Defined in C99/C11-standard, its an 80-bit floating point number slightly different than the 32 and 64-bit numbers
  - C99/C11 not implemented in MSVC…
  - Available as __float80 or long double in g++

- Was introduced to give enough accuracy for exponentiation (hardware did not have support for it, and instead computed $x^y = 2^{y \log_2 x}$)

- Extremely unintuitive: when a variable x is in a register, it has 80-bit precision. When it is flushed to the caches or main memory, it can have 128-bit storage.

Floating point and numerical errors

- Some systems are chaotic
  - Is single precision accurate enough for your model?
  - Is double precision --"--?
  - Is quad precision --"--?
  - Is …
- Put another way:
  - What is the minimum precision required for your model?



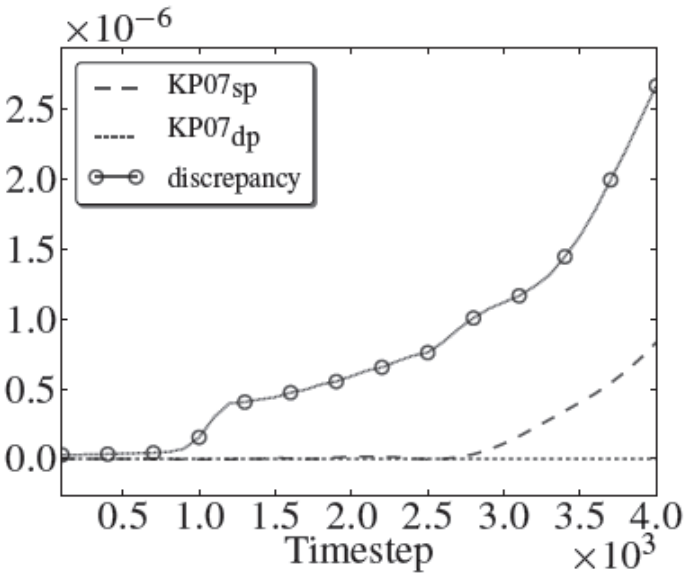
Lorenz strange attractor, Wikimol, wikipedia, CC-BY-SA 3.0

Single versus double precision in shallow water

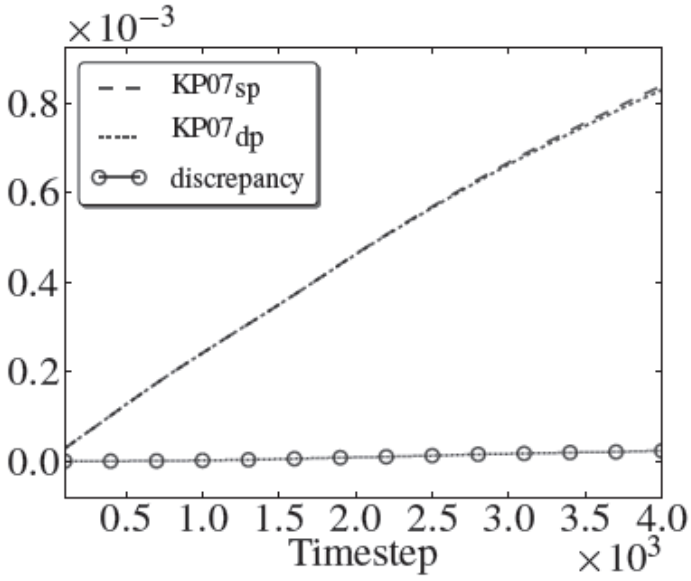
- Shallow water equations: Well studied equations for physical phenomenon
- Difficult to capture wet-dry interfaces accurately
- Let's see the effect of single versus double precision measured as error in conservation of mass

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

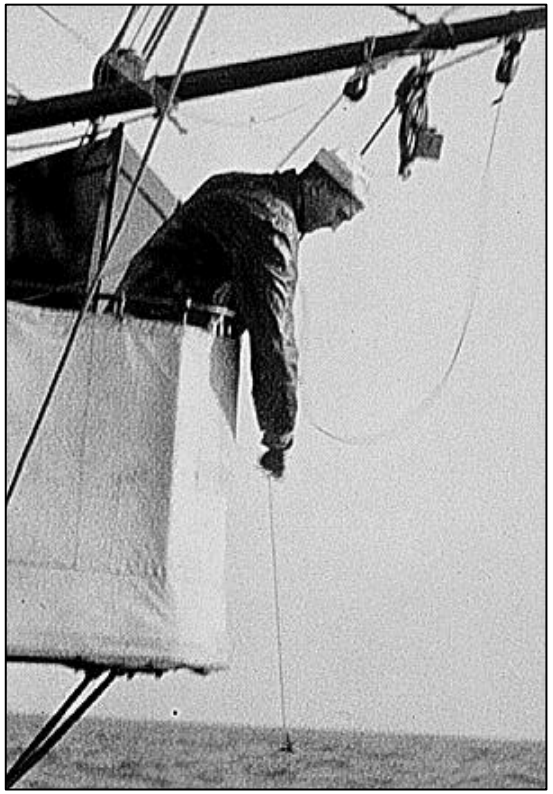


[1] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie and J. R. Natvig, **Simulation and Visualization of the Saint-Venant System using GPUs**, *Computing and Visualization in Science, 2011*

Floating point is often the least problem wrt accuracy

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

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



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

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 from this catastrophic cancellation!

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

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

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

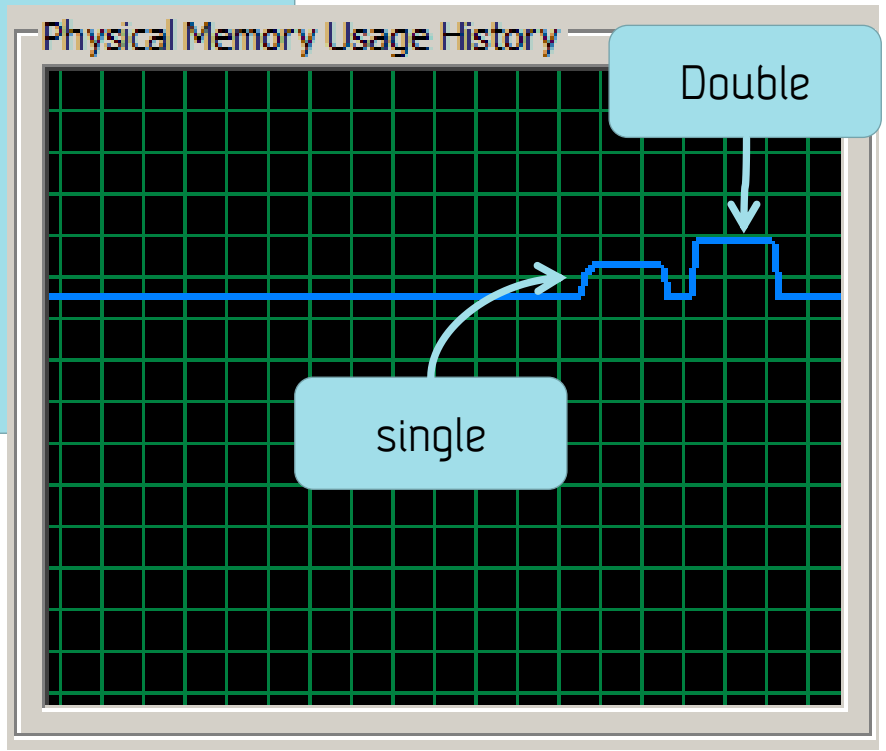
- Double precision is not enough in certain cases
  - Quad precision? Arbitrary precision?
  - Extremely expensive operations (100x+++ time usage)

Demo time

- 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



Demo time rev 2

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 0.018592999999999998417266056094376835972070693969 73 s.

Double:
Floating point bits=64
999999.99983897537458688020706176757812500000000000000000
Completed in 0.023868000000000003268496584496460855007171630859 4 s.

Long double (__float80):
Floating point bits=128
1000000.00000008712743237992981448769569396972656250000000
Completed in 0.020435999999999999304778341979726974386721849441 53 s.

Quad (__float128):
Floating point bits=128
1000000.00000000000000000000000000000000000000000000000000
Completed in 1.39770400000000005746869646827690303325653076171875 s.

The patriot missile…

- Designed by the 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…

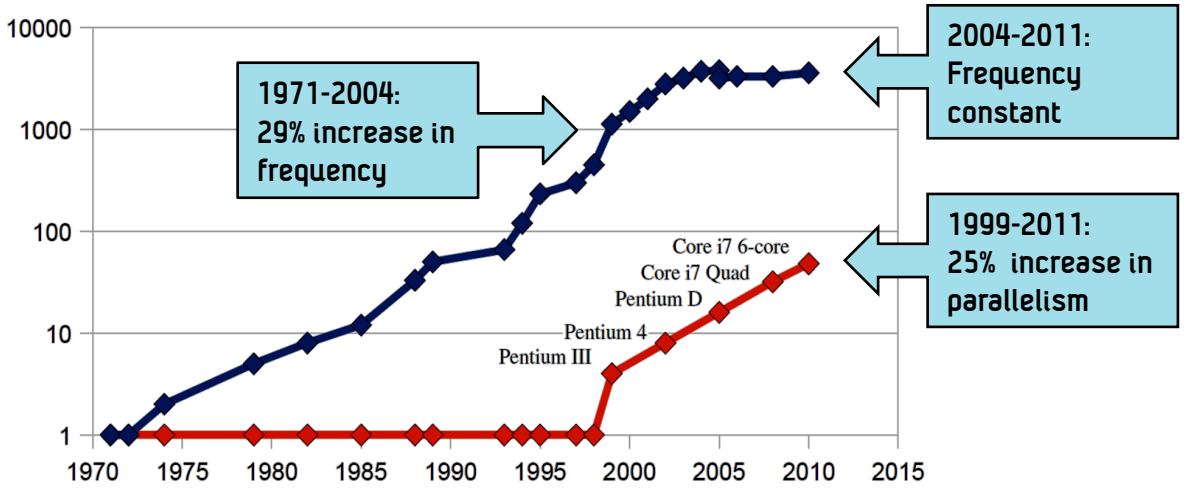
- It appears, that 0.1 seconds is not really 0.1 seconds…
  - Especially if you add a large amount of them

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

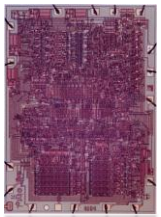
Should I care about parallel computing?



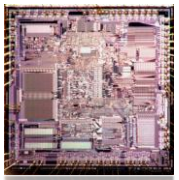
A serial program uses 2% of available resources!

Parallelism technologies:

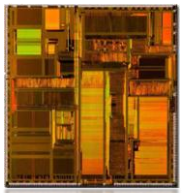
- Multi-core (8x)
- Hyper threading (2x)
- AVX/SSE/MMX/etc (8x)



1971: Intel 4004,
2300 trans, 740 KHz

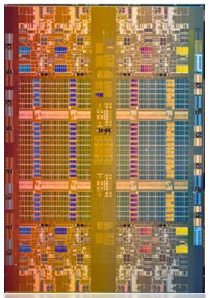


1982: Intel 80286,
134 thousand trans, 8 MHz



1993: Intel Pentium P5,
1.18 mill. trans, 66 MHz

2000: Intel Pentium 4,
42 mill. trans, 1.5 GHz



2010: Intel Nehalem,
2.3 bill. trans, 8 X 2.66 GHz

Floating point and parallelism

- Fact 1: Floating point is non-associative:
  - a*(b*c) != (a*b)*c
  - a+(b+c) != (a+b)+c
  - …

Floating point and parallelism

- 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: 976668.750000000000000000000000000000000000000000000000000000
Run 1: 976759.375000000000000000000000000000000000000000000000000000
Run 2: 976424.875000000000000000000000000000000000000000000000000000
Run 3: 977388.375000000000000000000000000000000000000000000000000000
Run 4: 981089.062500000000000000000000000000000000000000000000000000
Run 5: 976620.250000000000000000000000000000000000000000000000000000

Double:
Floating point bits=64
Run 0: 1000000.000038751800000000000000000000000000000000000000000000
Run 1: 1000000.000038983100000000000000000000000000000000000000000000
Run 2: 1000000.000034328100000000000000000000000000000000000000000000
Run 3: 1000000.000039123900000000000000000000000000000000000000000000
Run 4: 1000000.000038272000000000000000000000000000000000000000000000
Run 5: 1000000.000037564800000000000000000000000000000000000000000000

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

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

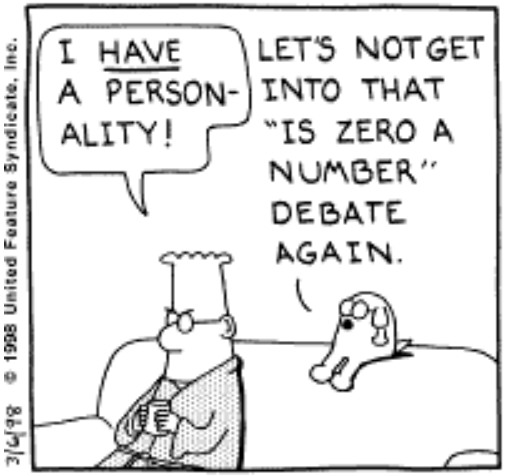
Advanced floating point

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

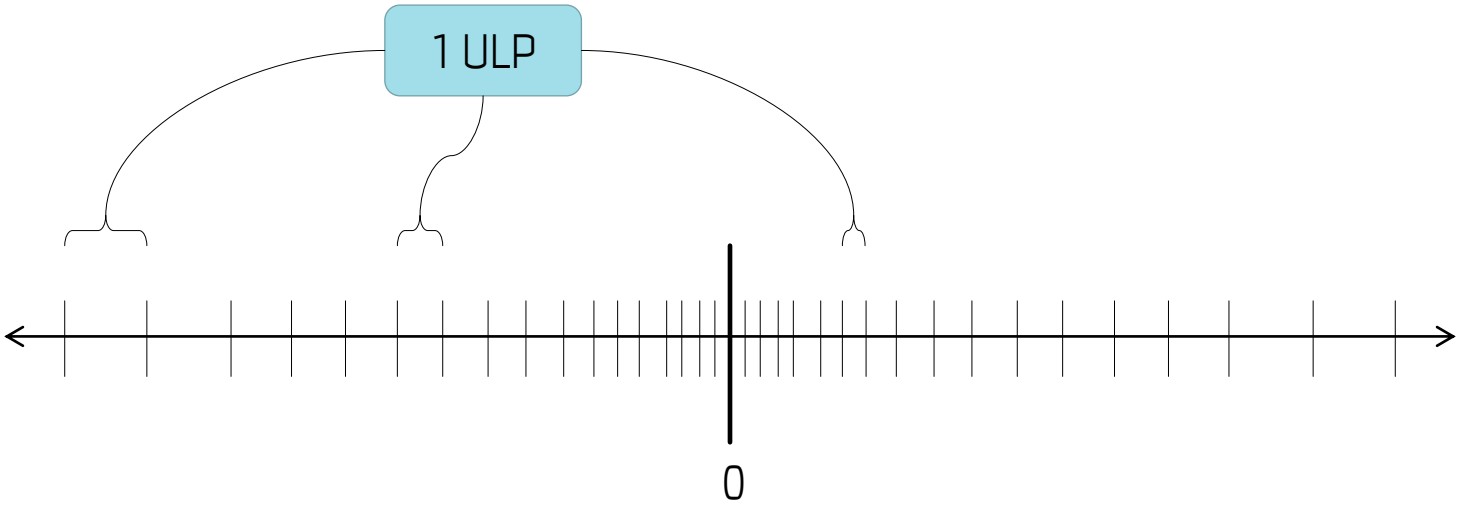
Special floating point numbers

- 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

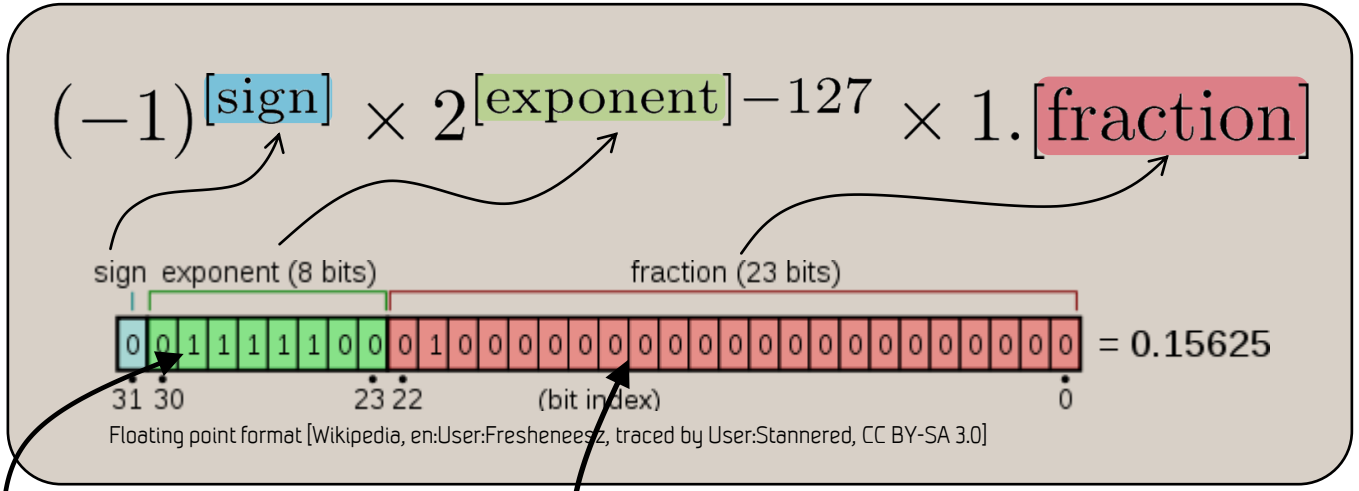


- "The most natural way to measure floating point errors"
- Number of contaminated digits: $\log_2 n$ when the error is n ulps

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

Subnormals

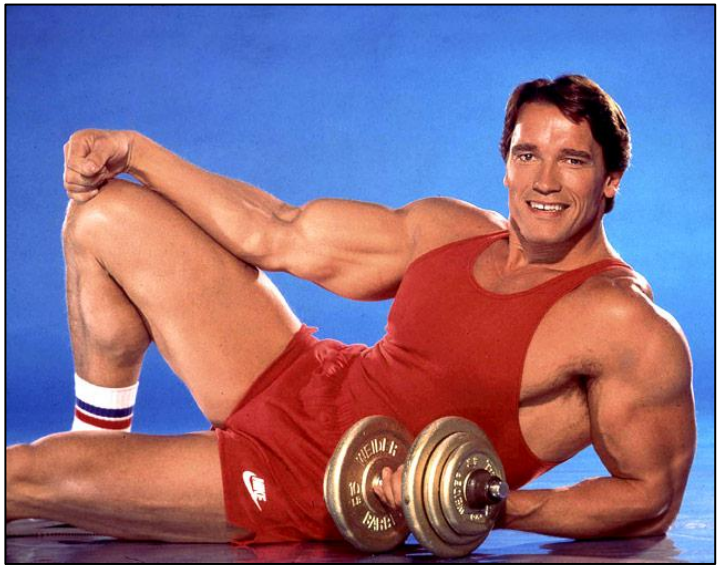
- Subnormals / denormals are gradual underflows
  - Graceful loss of precision instead of flush to zero
  - Can be really, really, expensive

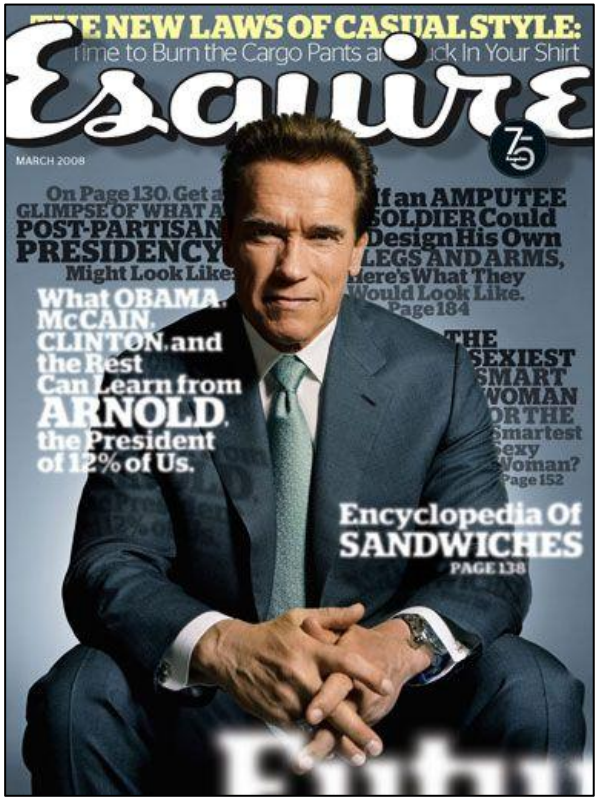


Exponent zero when subnormal

Leading zeros appear in significand / fraction / mantissa when subnormal

Some differences between 1985 and 2008

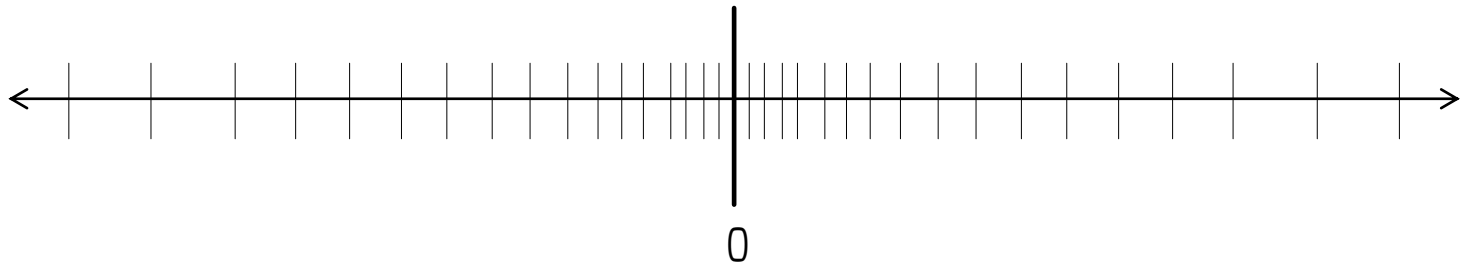




- Floating point multiply-add as a fused operation
  - a = b*c+d with only **one** round-off error
  - GPUs implement this already
- This is basically the same deal as the extended precision.
  - It's a good idea to use this instruction, but it gives "unpredictable" results
  - Users need to be aware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

Floating point best practices

- Floating point has the highest resolution around 0:
  - Lattice Bolzmann intermediate results: subtract 1 when storing to keep resolution
  - Store water elevations in shallow water as depths, or as deviations from mean sea level, not elevations.



Silent Data Corruption [1]

- Silent data corruption happens when a bit is flipped "by itself"…
- Can be handled somewhat with ECC memory (available on servers)
- Can have many causes: Environmental (temperature/voltage fluctuations; particles), manufacturing residues, oxide breakdown, electro-static discharge.
  - Estimate of 1 cosmic-ray-neutron-induced SDC every 1.5 months of operation (RoadRunner)
  - Smaller feature sizes increases frequency of SDC's

[1] Sarah Michalak, Silent Data Corruption and Other Anomalies, ICERM talk, 2012, http://faculty.washington.edu/rjl/icerm2012/Lightning/michalak.pdf

Reporting performance

What do we do in papers we publish?

A. Solve a problem that we previously could not

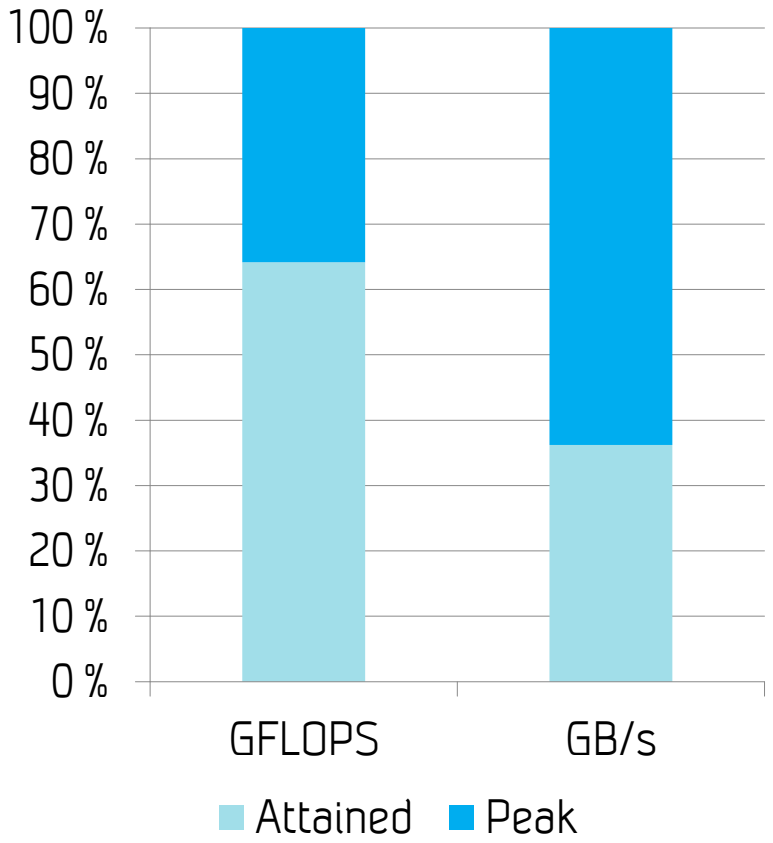
B. Solve an existing problem better than previously
   i. More accurately in the same amount of time
   ii. As accurate as before, but faster
   iii. A more demanding version of an existing problem

C. Perform a case study / write a survey article / …

Performance reporting is often a key element in B

Assessing performance

- Different ways of assessing performance
  - Algorithmic performance, numerical performance, wall clock time, …
- Speedups can be dishonest
  - Comparison of apples to oranges
- Sanity check for performance: Profile your code, and see what percentage of peak performance you attain
  - The aim should be to approach peak performance



Top Ways of Misleading the Masses [1]

1. Quote only 32-bit performance results, not 64-bit results
2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application
3. Quote performance results projected to a full system
4. When direct run time comparisons are required, compare with an old code on an obsolete system
5. If all else fails, show pretty pictures and animated videos, and don't talk about performance

[1] Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers David H. Bailey, 1991

"In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering"

--Donald Knuth

Summary

- Floating point can be devastating when misused
  - But floating point is most often **not** the largest problem
  - Programming errors, model errors, measurement errors…
- Floating point and parallel computing do not work well at all
  - Examine at algorithms that handle summation and parallelism well without affecting performance.
- Tell people that computers are non-deterministic
  Tell people that all results have uncertainties by including error bars
- Be methodical, thorough, and honest; also when reporting performance

Further reading

- Accuracy and Stability of Numerical Algorithms, Nicholas J. Higham
- What every computer scientist should know about floating-point arithmetic, David Goldberg, Computing Surveys , 1991.
- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers, David H. Bailey, Supercomputing Review, 1991.
- Ten Ways to Fool the Masses When Giving Performance Results on GPUs, Scott Pakin, HPC Wire, 2011