Object-Oriented Programming for Scientific Computing

Scientific Computing and Best Practices

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

Summer Semester 2018

Prerequisites and Objectives

Prerequisites

- Advanced knowledge of a programming language
- At least procedural programming in C / C++
- Willingness to program in practice

Objectives

- Improved programming skills
- Introduction of modern programming models
- Strong focus on topics of relevance to Scientific Computing

Course Outline

- Scientific Computing and best practices
- Short recapitulation of basics of object-oriented programming in C++ (classes, inheritance, methods and operators)
- Smart Pointers and Constness
- Error handling (exceptions)
- Dynamic polymorphism (virtual inheritance)
- Static polymorphism (templates)
- The C++ Standard Template Library (STL)
- Traits and Policies
- Design Patterns
- Template Metaprogramming
- C++ Threads

Throughout the lectures changes due to the C++{11,14,17} standards will be taken into account.

Lecture Website

Link: `conan.iwr.uni-heidelberg.de/teaching/ooprogram_ss2018/`

Content:

- Link to draft of official standard
- Literature recommendations and quick reference
- GitLab instructions (for exercises)
- Lecture slides (updated after each lecture)
- Exercise sheets (same as above)

MUESLI links (also listed on the lecture website):

- Registration: `muesli.mathi.uni-heidelberg.de/user/register`
- Course page: `muesli.mathi.uni-heidelberg.de/lecture/view/866`

Exercises and Exam

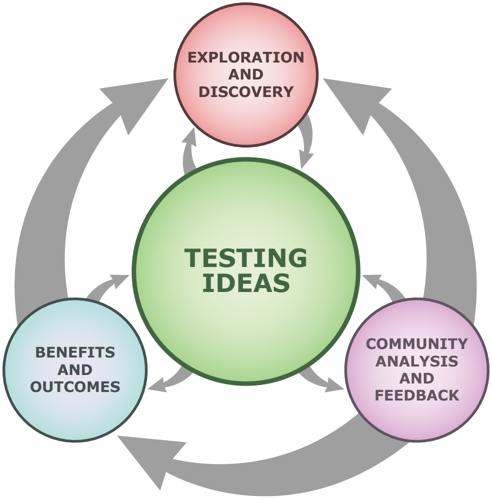
Exercises:

- Mo and Fr, 11:00–13:00, INF205, Computer Pool (third floor)
- New exercises every week: on the website after the lecture
- To be handed in right before the lecture on Wednesday
- Geared towards g++ and Linux
- Correction, grading etc. depends on course size

Exam:

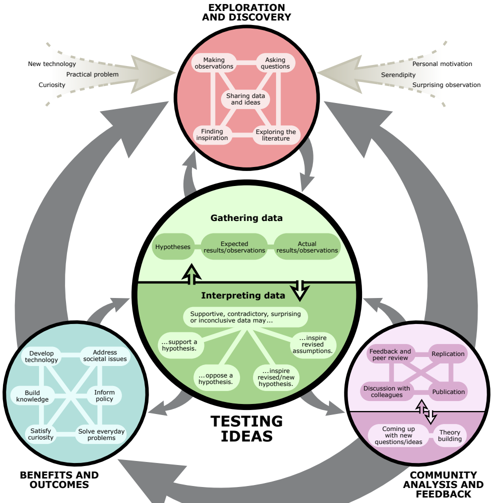
- Will take place in the last week of the semester (25.7.)
- Mandatory for all participants
- 50% of the points in the exercises required for admission

How science works



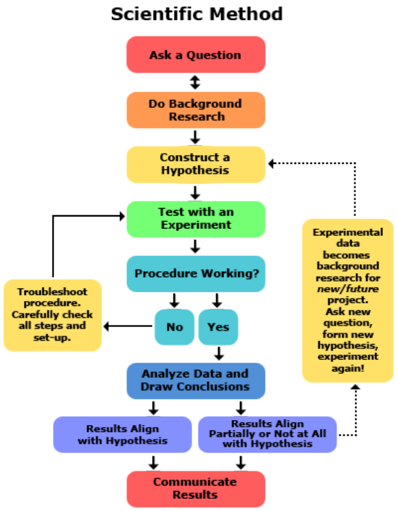
www.understandingscience.org
© 2008 The University of California Museum of Paleontology, Berkeley, and the Regents of the University of California

How science works



www.understandingscience.org
© 2008 The University of California Museum of Paleontology, Berkeley, and the Regents of the University of California

Scientific Method


Scientific Method
Ask a Question
Do Background Research
Construct a Hypothesis
Test with an Experiment
Procedure Working?
No
Yes
Troubleshoot procedure. Carefully check all steps and set-up.
Analyze Data and Draw Conclusions
Results Align with Hypothesis
Results Align Partially or Not at All with Hypothesis
Experimental data becomes background research for new/future project. Ask new question, form new hypothesis, experiment again!
Communicate Results

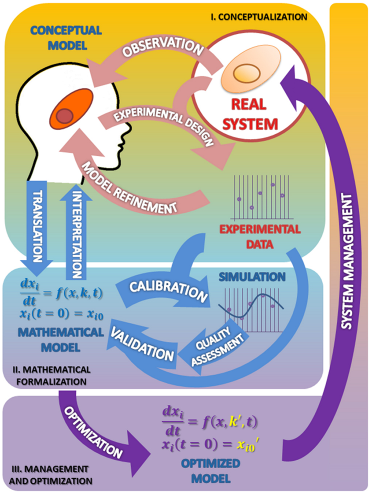
Source: sciencebuddies.org

In given context (e.g. master thesis):

- Question/Hypothesis: Thesis topic, provided by supervisor or proposed by student
- Experiment: Modeling, programming, simulation
- Communication: Written thesis, presentation, project repository (!)

Depending on findings and results, thesis scope may have to be extended, shrunk or shifted — requires additional feedback loop and steady communication between supervisor and student

Modeling Circle



Source: Frontiers, frontiersin.org

Three central topics in Scientific Computing: Modeling, Simulation, Optimization (MSO)

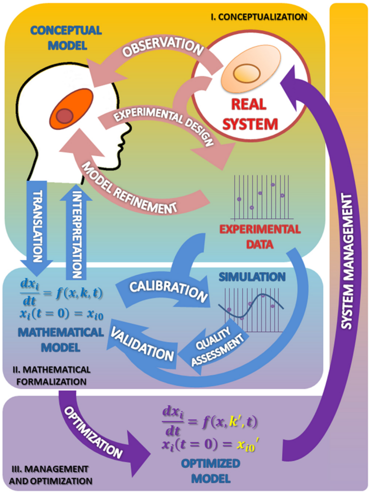
Modeling: Representation of systems and processes in the real world using mathematics

Simulation: Generation of (discretized) model outcomes in a computer

Optimization: Find optimal system state/evolution for given objective

Scientific Computing in the strict sense focuses on simulation and optimization

Modeling Circle



Source: Frontiers, frontiersin.org

Real system: Part of the observable world around us, closed or with (more or less) known boundary conditions

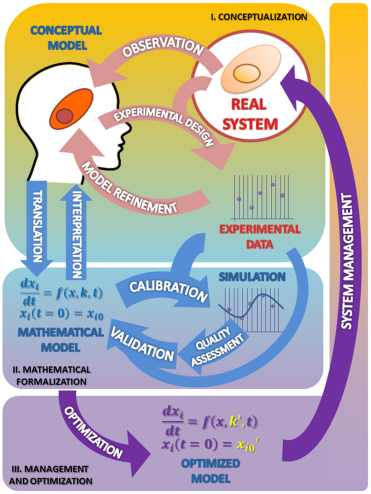
Example: Our planet's climate

Abstraction:
real system $\rightarrow$ conceptual model

Conceptual, “physical” model:
Representation of system in terms of actors (atoms, molecules), quantities (pressure, temperature) and interactions

The climate as interaction between temperature, pressure, wind velocity, humidity, . . .

Modeling Circle



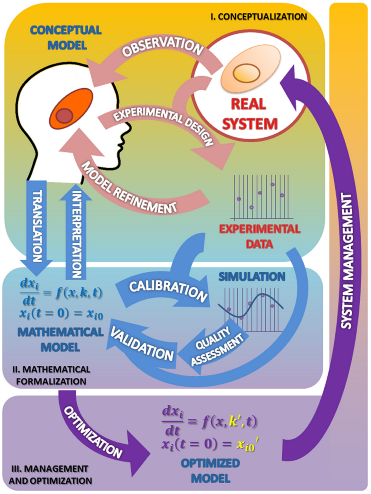
Source: Frontiers, frontiersin.org

Formalization:
conceptual model $\rightarrow$ mathematical model

Formal, "mathematical" model: Based on formulas, especially ordinary and partial differential equations (ODEs and PDEs)

PDE model for convective atmospheric air flow, coupled with other processes and, e.g., oceanic models

Modeling Circle



Source: Frontiers, frontiersin.org

Discretization:
mathematical model → discrete model

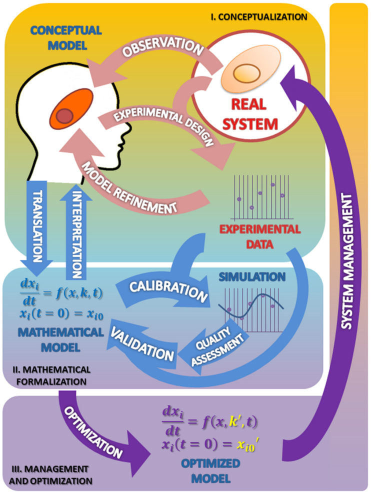
Discrete model: Finite-dimensional approximation that allows simulation on a computer

Finite element or finite volume discretization of the climate model PDEs

Implementation:
discrete model → actual code

Most crucial steps from Scientific Computing viewpoint, but often not included in high-level concept of field experts

Modeling Circle



Source: Frontiers, frontiersin.org

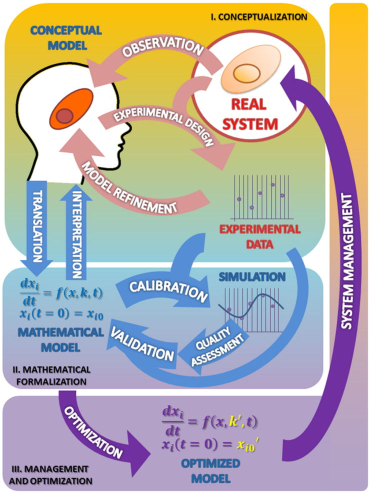
Program code is used as stand-in for real system, which leads to several systematic errors:

- Conceptual model: misconception, oversight
- Formal model: simplification, approximation
- Discrete model: discretization error
- Program code: bug

These errors have to be taken into account (if possible)
$\rightarrow$ necessity of validation

There is no such thing as a perfect model!

Modeling Circle



Source: Frontiers, frontiersin.org

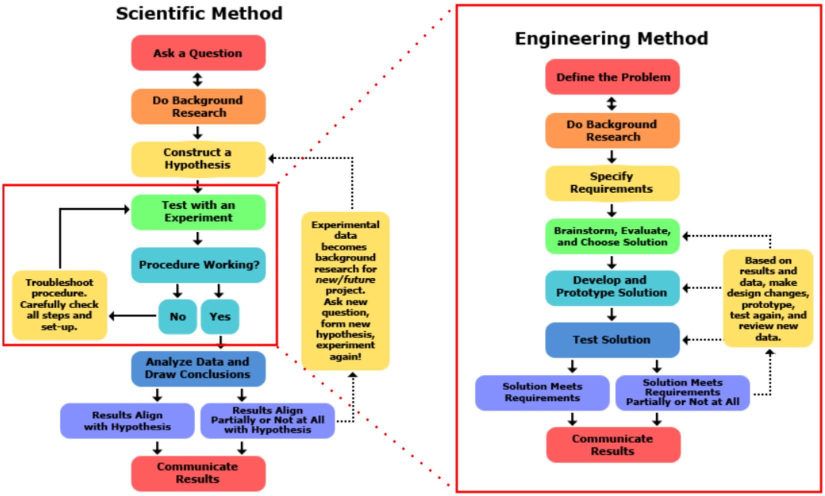
Calibration: Adjust parameters of model to optimize alignment with observed data

Validation: Test model prediction on other (!) observed data

Interpretation: Draw conclusions about the real system based on simulation results

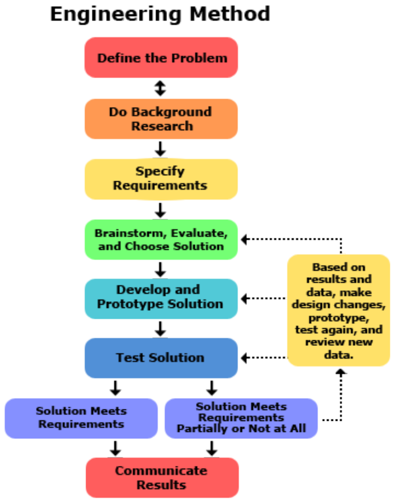
. . . closes loop by allowing new observations based on deeper understanding of system

Scientific Computing: Science and Engineering


Scientific Method
Ask a Question
Do Background Research
Construct a Hypothesis
Test with an Experiment
Procedure Working?
No
Yes
Troubleshoot procedure. Carefully check all steps and set-up.
Analyze Data and Draw Conclusions
Results Align with Hypothesis
Results Align Partially or Not at All with Hypothesis
Communicate Results
Experimental data becomes background research for new/future project. Ask new question, form new hypothesis, experiment again!
Engineering Method
Define the Problem
Do Background Research
Specify Requirements
Brainstorm, Evaluate, and Choose Solution
Develop and Prototype Solution
Test Solution
Solution Meets Requirements
Solution Meets Requirements Partially or Not at All
Communicate Results
Based on results and data, make design changes, prototype, test again, and review new data.

Source: sciencebuddies.org

Engineering Method


Engineering Method
Define the Problem
Do Background Research
Specify Requirements
Brainstorm, Evaluate, and Choose Solution
Develop and Prototype Solution
Test Solution
Solution Meets Requirements
Solution Meets Requirements Partially or Not at All
Communicate Results
Based on results and data, make design changes, prototype, test again, and review new data.

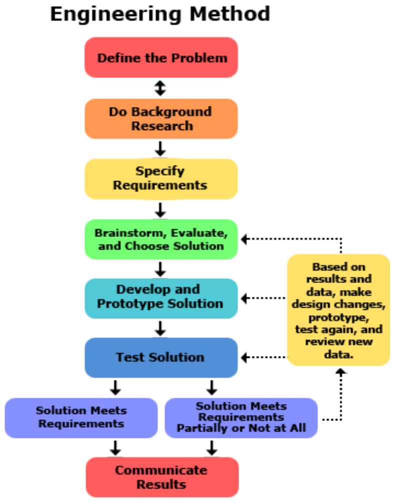
Source: sciencebuddies.org

In given context:

- Problem: write code for simulation/optimization
- Requirements: accuracy, efficiency, readability (!)
- Choose solution: Find/develop algorithms for subproblems

Programming loop is embedded in research loop — unforeseen changes in project definition may require further iterations, and loops will tend to become concurrent due to code reuse

Engineering Method


Engineering Method
Define the Problem
Do Background Research
Specify Requirements
Brainstorm, Evaluate, and Choose Solution
Develop and Prototype Solution
Test Solution
Solution Meets Requirements
Solution Meets Requirements Partially or Not at All
Communicate Results
Based on results and data, make design changes, prototype, test again, and review new data.

Source: sciencebuddies.org

- Base code on external libraries and packages, since many subproblems have already been solved
- Start with the simplest possible model and gradually increase complexity
- Test output in each intermediate step, and check for obvious errors

Need for testing leads to frequent context switches: develop theory, program code, test solution, fix code, refine theory, update code, etc. — and in-between, write the first parts of the thesis

Summary

- Scientific progress is based on feedback loops (with room for variation in their implementation)
- In Scientific Computing, several such loops exist and interact with one another
- This requires flexibility and the ability to “see the big picture”
- Models and programs should both start simple and become increasingly more complex
- Frequent pauses for error control are important (validation for models, tests for programs)

Best Practices for Scientific Computing

G. Wilson, D.A. Aruliah, C.T. Brown, N.P.C. Hong, M. Davis, R.T. Guy, S.H.D. Haddock, K.D. Huff, I.M. Mitchell, M.D. Plumbley, B. Waugh, E.P. White, P. Wilson:
Best Practices for Scientific Computing, PLOS Biology

1. Write programs for people, not computers
2. Let the computer do the work
3. Make incremental changes
4. Don't repeat yourself (or others)
5. Plan for mistakes
6. Optimize software only after it works correctly
7. Document design and purpose, not mechanics
8. Collaborate

Write Programs for People, not Computers

Software must be easy to read and understand by other programmers (especially your future self!)

- Programs should not require their readers to hold more than a handful of facts in memory at once
  - Short-term memory: The Magical Number Seven, Plus or Minus Two
  - Chunking (psychology): binding individual pieces of information into meaningful whole
  - Good reason for encapsulation and modularity
- Make names and identifiers
  - consistent
  - distinctive
  - meaningful
- Make coding style and formatting consistent

Minimum Style Requirements

Programming style conventions can be inconvenient, but there is a bare minimum that should be followed in any case:

- Properly indent your code, and use meaningful names for variables and functions
- Comment your code, but don't comment every line: main idea/purpose of class or function, hidden assumptions, critical details
- Don't just fix compilation errors, also make sure that the compiler doesn't issue warnings

This makes it easier for other people to understand your work, including:

- Colleagues you may ask for help or input
- Other researchers extending or using your work
- Yourself in a few weeks or months (!)

Let the Computer do the Work

Typing the same commands over and over again is both time-consuming and error-prone

- Make the computer repeat tasks
- Save recent commands in a file for re-use
- Use a build tool to automate workflows (e.g. makefiles, cmake)

But make sure you don't waste time building unnecessarily intricate automation structures!

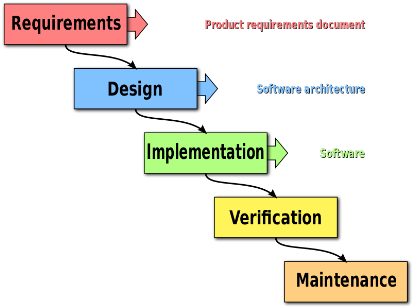
Make Incremental Changes

Requirements of scientific software typically aren't known in advance

- Work in small steps with frequent feedback and course correction (e.g. agile development)
- Use a version control system (e.g. Git, GitLab)
- Put everything that has been created manually under version control
    - The source code, of course
    - Source files for papers / documents
    - Raw data (from field experiments or benchmarks)

Large chunks of data (data from experiments, important program results, figures) can be stored efficiently using Git-LFS (large file storage) extension

Project Planning

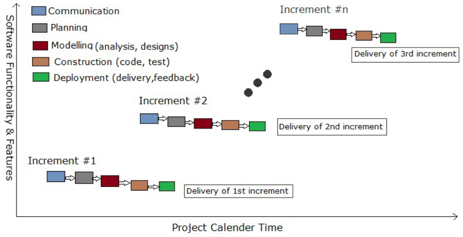


Source: Peter Kemp / Paul Smith, wikipedia.org

The waterfall model is the oldest software development process:

- Originates in manufacturing and construction industries, where changes to the original plan are prohibitively expensive
- Doesn't allow for changes after completion of the initial planning step
- Unsuitable for the investigative nature of Scientific Programming

Project Planning


Communication
Planning
Modelling (analysis, designs)
Construction (code, test)
Deployment (delivery,feedback)
Increment #1
Delivery of 1st increment
Increment #2
Delivery of 2nd increment
Increment #n
Delivery of 3rd increment
Software Functionality & Features
Project Calender Time

Source: RahulT, wikipedia.org

The incremental build model (design, implement, test, repeat) applies the waterfall model recursively, which makes it suitable for Scientific Computing.

There are more sophisticated design processes (agile software development, lean software development), focusing on communication and efficiency. These can be very useful for research teams, but in small personal projects they tend to overcomplicate things.

Don't Repeat Yourself (or Others)

Anything that exists in two or more places is harder to maintain and may introduce inconsistencies

- Every piece of data must have a single authoritative representation in the system
- Modularize code rather than copying and pasting
- Re-use code instead of rewriting it
- Make use of external libraries (as long as it is not too cumbersome)

First point is actually known as DRY principle in general software design and engineering

Plan for Mistakes

Mistakes are human and happen on all levels of software development (e.g. bug, misconception, problematic design choice), even in experienced teams

- Add assertions to programs to check their operation
  - Simple form of error detection
  - Can serve as documentation (that is auto-updated)
- Use an off-the-shelf unit testing library
  - Can assist in finding unexpected side effects of code changes
  - In case of GitLab: GitLab CI (continuous integration)
- Turn bugs into test cases
- Use a symbolic debugger (e.g. GDB, DDD, Gnome Nemiver, LLDB)

Assertions should only be used to catch misconceptions and omissions by programmers, expected runtime errors (file not found, out of memory, solver didn't converge) are handled using exceptions (later)!

Using the GDB Debugger

A debugger like GDB allows you to:

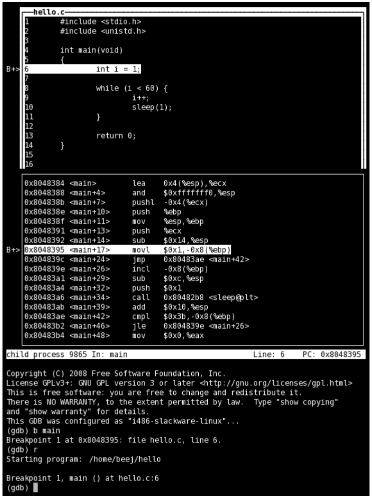
- Interactively run a program and inspect how it behaves / fails
- Attach to an already running (!) program that you want to analyze
- Read in the core dump of a failed program (post mortem analysis)

You can do the following:

- Run one instruction / function at a time, forwards and backwards (!) in time
- Run the program until a specific line is reached (breakpoint)
- Run the program until some variable is used / changed (watchpoint)
- Change values on-the-fly and see how it affects the program (!)

. . . and much more, there is even the theoretical possibility to write scripts that automate part of the debugging process

Using the GDB Debugger



```
hello.c
    1       #include <stdio.h>
    2       #include <unistd.h>
    3
    4       int main(void)
    5       {
B+> 6               int i = 1;
    7
    8               while (i < 60) {
    9                       i++;
    10                      sleep(1);
    11              }
    12
    13              return 0;
    14      }
    15
    16

    0x8048384 <main>        lea     0x4(%esp),%ecx
    0x8048388 <main+4>      and     $0xfffffff0,%esp
    0x804838b <main+7>      pushl   -0x4(%ecx)
    0x804838e <main+10>     push    %ebp
    0x804838f <main+11>     mov     %esp,%ebp
    0x8048391 <main+13>     push    %ecx
    0x8048392 <main+14>     sub     $0x14,%esp
B+> 0x8048395 <main+17>     movl    $0x1,-0x8(%ebp)
    0x804839c <main+24>     jmp     0x80483ae <main+42>
    0x804839e <main+26>     incl    -0x8(%ebp)
    0x80483a1 <main+29>     sub     $0xc,%esp
    0x80483a4 <main+32>     push    $0x1
    0x80483a6 <main+34>     call    0x80482b8 <sleep@plt>
    0x80483ab <main+39>     add     $0x10,%esp
    0x80483ae <main+42>     cmpl    $0x3b,-0x8(%ebp)
    0x80483b2 <main+46>     jle     0x804839e <main+26>
    0x80483b4 <main+48>     mov     $0x0,%eax

child process 9865 In: main                              Line: 6    PC: 0x8048395

Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...
(gdb) b main
Breakpoint 1 at 0x8048395: file hello.c, line 6.
(gdb) r
Starting program: /home/beej/hello

Breakpoint 1, main () at hello.c:6
(gdb) █
```

Source: Brian Hall, beej.us

GDB has several graphical frontends, e.g. DDD and Nemiver, but also a built-in improvement many don't know: the TUI (Text User Interface)

TUI mode provides you with:

- An editor-like code window (instead of cumbersome "list" commands)
- An optional disassembly window
- Windows for register contents (not shown)
- The default GDB terminal for commands, as usual

More information about GDB and its TUI mode: `https://beej.us/guide/bggdb/`

Common Compiler Flags

Debugging information:

- `-g`: include debug symbols in program

Optimization:

- `-O0`: completely disable optimization (default!)
- `-Og`: flags of `-O1` that don't interfere with debugging
- `-O1`: use optimizations that don't impact on compile time
- `-Os`: flags of `-O2` that don't increase program size
- `-O2`: use most optimizations, increases compile time
- `-O3`: flags of `-O2` plus expensive optimizations
- `-Ofast`: flags of `-O3` plus `-ffastmath`, not standards compliant

Recommendation: `-Og -g` for development, `-O3` or `-Ofast` for application

Others:

- `-march=native`: compile for local processor architecture (fast but non-portable)
- `-Wall`: enable all warning about possible coding errors
- `-Werror`: turn warnings into hard errors

The Importance of Backups

Losing your program code means losing months or even years of hard work. There are three standard ways to guard against this possibility:

- Create automatic backups (but know how the restore process works!)
- Create a manual copy on an external drive or another computer (messy!)
- Use a source code repository to manage the files

It's a good idea to use two approaches as a precaution

Of the above options, using a repository has the largest number of benefits

The Importance of Backups

Advantages of using a repository for code management:

- The repository can be on a different computer, i.e., the repository is automatically also an old-fashioned backup
- Repositories preserve project history, making it easy to retrieve older code or compare versions
- Repositories can be shared with others, making collaboration easy
- When writing a thesis it is often expected that you hand in your source code, which simply amounts to giving access to your repository (or directly hosting it on the research group's infrastructure)

One of the tools most often used for this is Git, which can be used in conjunction with websites like GitHub or Bitbucket. An open source alternative is GitLab, which can also be installed locally to provide repositories for the research group (and is used in the exercises of this lecture).

Optimize Software Only After it Works Correctly

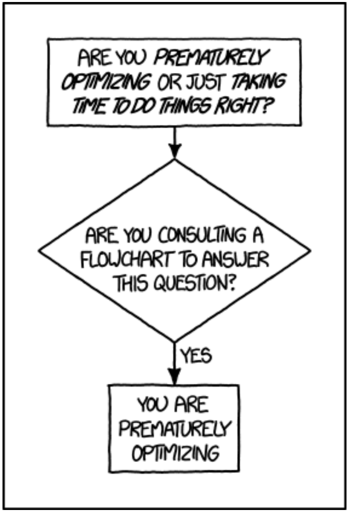
Identifying potential bottlenecks is hard, and optimization of other parts of the program is a waste of time

- Use a profiler to identify bottlenecks
- Write code in the highest-level language possible

This is an optimization with regard to development time

High-level prototypes (e.g. in Python, Matlab) can serve as oracles for low-level high-performance implementations (e.g. in C++)

Code and Coding Efficiency


ARE YOU *PREMATURELY OPTIMIZING* OR JUST *TAKING TIME TO DO THINGS RIGHT?*
ARE YOU CONSULTING A FLOWCHART TO ANSWER THIS QUESTION?
YES
YOU ARE PREMATURELY OPTIMIZING

Source: Randall Munroe, xkcd.com

The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.

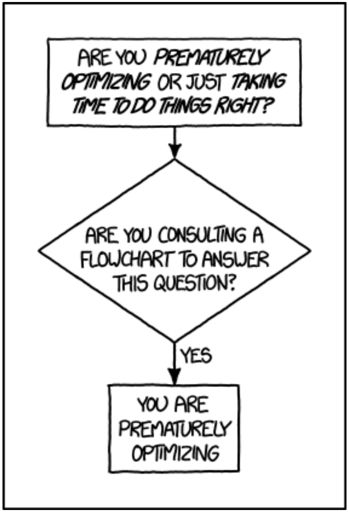
— Tom Cargill, Bell Labs

Even experts are often unable to accurately predict the time requirements for developing a piece of software

This means:

- Project plans should be conservative
- Don't underestimate the work needed for the hard parts of coding

Code and Coding Efficiency


ARE YOU PREMATURELY OPTIMIZING OR JUST TAKING TIME TO DO THINGS RIGHT?
ARE YOU CONSULTING A FLOWCHART TO ANSWER THIS QUESTION?
YES
YOU ARE PREMATURELY OPTIMIZING

Source: Randall Munroe, xkcd.com

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
— Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid

As a result, there are two measures of efficiency, the time your program needs to run, and the time you need to write it

- Don't optimize too soon, get a working prototype first (you will likely need to change optimized parts along the way)
- Look for bottlenecks and only optimize those (your program will spend most of its time in roughly 5% of your code)

Document Design and Purpose, not Mechanics

The main purpose of documentation is assistance of readers who are trying to use an implementation, choosing between different implementations, or planning to extend one

- Document interfaces and reasons, not implementations
    - Function and method signatures
    - Public members of classes
- Refactor code in preference to explaining how it works
- Embed documentation for a piece of software in that software

Refactored code that needs less documentation will often save time in the long run, and documentation that is bundled with code has much higher chance to stay relevant

Collaborate

Scientific programs are often produced by research teams, and this provides both unique opportunities and potential sources of conflict

- Use pre-merge code reviews
- Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems
- Use an issue tracking tool

Pre-merge reviews are the only way to guarantee that code has been checked by another person

Pair programming is very efficient but intrusive, and should be used with care

Numerical Experiments

The development of scientific software requires frequent testing, and it is usually helpful if these tests can be compared at a later point in time, regarding things like:

- Time to solution
- Required number of iterations
- Discretization error (if available)

This can be achieved by storing each test in a separate directory, together with:

- an identifier for your program version (e.g., number of most recent commit)
- the chosen control parameters (e.g., automatically printed by your program)
- the input files, if any
- the results (if space permits)

Documentation and Thesis

If you expect (or would like) your finished program to be used in other projects, then it is important that you document how it is used. This includes:

- Installation/compilation instructions
- List of external dependencies, together with tested versions
- A working example explaining how the program is used

Remarks regarding the written thesis:

- Don't include the program documentation, reference it
- While programming is the largest part of any project, it should be kept to a minimum in the written thesis — only include central algorithms and general framework of your program
- The document can be treated just like program code and kept in a repository for backup and possibly reference

Summary

- Follow at least a bare minimum of style conventions to keep your program code readable
- Optimize only the critical parts of your code, and only once you don't expect any further significant changes (or the code is unbearably slow)
- Don't spend too much time on project planning (instead of programming), but create a coarse initial plan and allow for changes along the way
- Make frequent backups, e.g., by using a repository of a version control system (on a different computer)
- A systematic approach to numerical experiments makes comparison easier and can safe time
- Provide detailed documentation of your software, but include only the most important parts in your thesis