Programming & Software Engineering for Scientists

Emanuel Gull

University of Michigan

Simons Foundation / Stony Brook Summer School on the Many-Electron Problem

The problem

- We are scientists, first and foremost. Our education and training is not in programming or software engineering.
- Many of us write code for a living.
- Few of us have had more than a basic ‘programming’ training. Few of us have taken lectures at a CS department. And even if: few of the techniques taught there are useful for us.
- Our algorithms get more and more complex.
  (no more 200 line codes)
- A lot of our time is spent debugging, not writing code.
- Codes survive from student to student, from year to year, but ‘problem at the transition’.
- As time progresses, codes get more and more messy. Finally we throw them out and start from scratch.

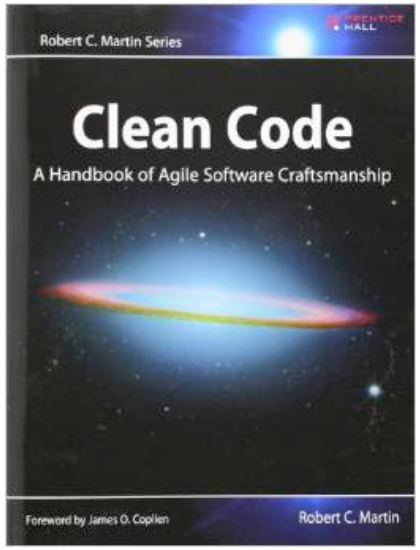
The promise

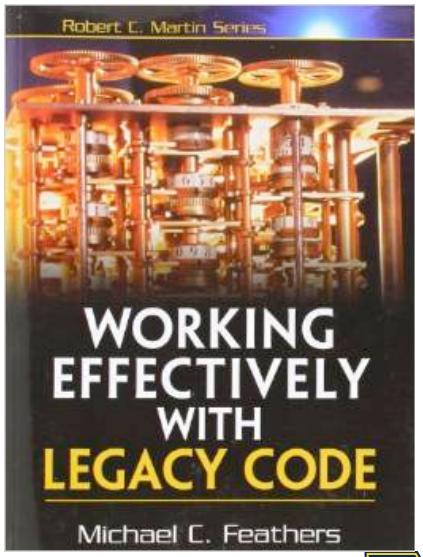
- If a little bit of investment in time, tools, and techniques could save you a lot of time programming, debugging, and maintaing code…
- If you could make your code easier to read and understand…
- If you could make your code easier to extend and reuse…
- …wouldn't that be worth it?

This lecture aims to present a basic overview of useful tools. Some of it you know. Some of it may not apply to your situation. Use whatever is useful: this is engineering!

…A big part of your university's CS department worries about these things, so there's much out there that I will not touch!

Good intro books for practitioners:





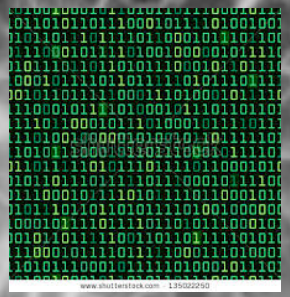
Overview

- Challenges
- Libraries
- Tools and infrastructure
- Testing done properly: Unit tests
- TDD: Philosophy and examples
- OO: why we should care

Common Challenges for Scientific Codes

Short term mission
(get papers out)

knowledge/tools/education
of the programmer
(backup via e-mail, if at all)



Complexity of the problem
(if we knew what we were
doing…)

Long term goals

Cutting edge / nonstandard
infrastructure
(cray, blue gene, …)

massive
parallelization

How codes decay

- Remember your last big project?
  - …clean design
  - …fast progress
  - …or 'just a bit of code to try something'?

- And then reality happened!

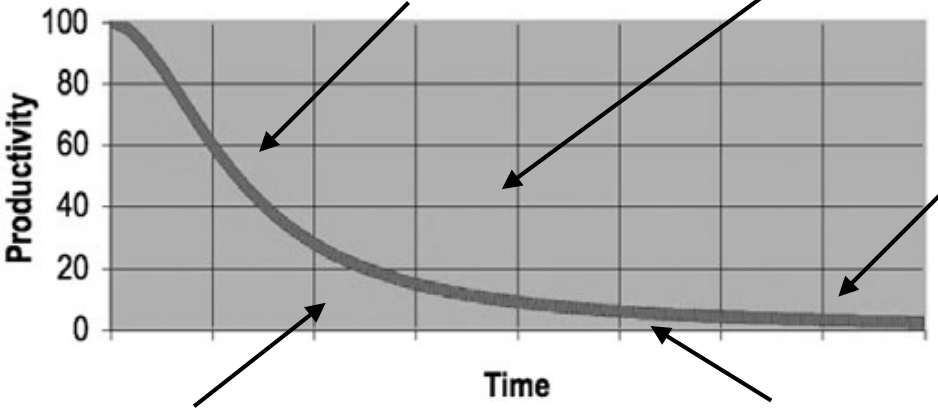


Figure 1-1
Productivity vs. time

Emanuel Gull

SIMONS FOUNDATION

MICHIGAN

How codes decay

- As time progresses
  - What used to be quick and easy is now hard and slow
  - Lots of code, lots of functionality,
    - not sure what actually still works
    - dead code
  - How to progress without breaking anything?
  - How to adapt to new environments?
  - How to clean without rewriting everything?

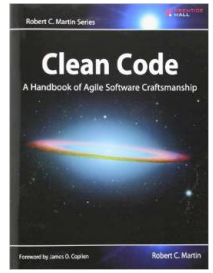
What is clean code?

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

> *I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.*

Grady Booch, author of *Object Oriented Analysis and Design with Applications*

> *Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.*



Environment & Libraries

- before we go over some easy techniques, let's
  - go over the computational environment
  - look at libraries
  - look at tools

L A P A C K
L -A P -A C -K
L A P A -C -K
L -A P -A -C K
L A -P -A C K
L -A -P A C -K

Languages

…are to a large extent a personal choice. Here are some guidelines:

- Use a modern programming language in wide use in industry. Good choices are:
  - **C++** for HPC codes
    - Personal opinion: do not use C++-11: not supported everywhere, not every feature supported by all compilers (others strongly disagree).
  - **Python** for scripting
- Whatever you use, it should be:
  - general
  - compilers are available everywhere
  - tools are available everywhere
- Stay away from ‘new’ languages and exotic features. The more standard, the easier it is for you to run your code 10 years down the road.
- The more standard, the easier it will be for someone else to take over.
- If it doesn’t compile on (insert exotic platform/machine/etc here), it probably will limit your choices down the road.
- If you use fortran: use the modern fortran language standards (fortran 03/08)
  - don’t use fortran unless required by community

Libraries

Physics is based on math. Many of the mathematics building blocks are available, tested, optimized, fool proof.

- Disadvantage of using a library:
  - Additional dependencies
  - You have to learn how to use it
  - It may be buggy

- Advantages of using a library:
  - Learn it once, keep using it!
  - Debugging and development, maintenance delegated to people who know what they are doing (and do this as their job)
  - Keep your code short and clean, don't reinvent the wheel!

What follows are a bunch of low level libraries and tools that are standard enough that you should **UNDER NO CIRCUMSTANCE** reinvent / reimplement their functionality

Basic Linear Algebra System

- BLAS: Basic Linear Algebra System
- de facto standard interface for low level linear algebra routines
- Available on any machine, for any compiler, often hand-optimized for best possible speed:
  - OSX: -framework accelerate or -framework vecLib
  - Intel compiler: icpc -mkl …
  - Automatically tuned library (ATLAS)
  - AMD Core Math Library on AMD machines
  - Intel Math Kernel Library on Intel machines
  - ESSL on IBM Blue Gene
- Separated in 3 levels:
  - Blas Level 1: scalar-vector and vector-vector interaction
  - Blas Level 2: matrix-vector operations
  - Blas Level 3: matrix-matrix operations
- About factor of 20 faster than naive nested loop for matrix multiplications

Basic Linear Algebra System

Level 1 BLAS

dim scalar vector vector scalars 5-element array		prefixes
SUBROUTINE xROTG (A, B, C, S)	Generate plane rotation	S, D
SUBROUTINE xROTMG(D1, D2, A, B, PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT (N, X, INCX, Y, INCY, C, S)	Apply plane rotation	S, D
SUBROUTINE xROTM (N, X, INCX, Y, INCY, PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP (N, X, INCX, Y, INCY)	$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL (N, ALPHA, X, INCX)	$x \leftarrow \alpha x$	S, D, C, Z
SUBROUTINE xCOPY (N, X, INCX, Y, INCY)	$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY (N, ALPHA, X, INCX, Y, INCY)	$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT (N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU (N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC (N, X, INCX, Y, INCY)	$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT (N, X, INCX, Y, INCY)	$dot \leftarrow \alpha + x^T y$	SDS
FUNCTION xNRM2 (N, X, INCX)	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC,
FUNCTION xASUM (N, X, INCX)	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC,
FUNCTION IxAMAX(N, X, INCX)	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

options dim b-width scalar matrix vector scalar vector		
xGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xGBMV (TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xHEMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xSYMV (UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSBMV (UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xTRMV (UPLO, TRANS, DIAG, N, A, LDA, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTBMV (UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTPMV (UPLO, TRANS, DIAG, N, AP, X, INCX)	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTRSV (UPLO, TRANS, DIAG, N, A, LDA, X, INCX)	$x \leftarrow A^{\Gamma_1} x, x \leftarrow A^{\Gamma_T} x, x \leftarrow A^{\Gamma_H} x$	S, D, C, Z
xTBSV (UPLO, TRANS, DIAG, N, K, A, LDA, X, INCX)	$x \leftarrow A^{\Gamma_1} x, x \leftarrow A^{\Gamma_T} x, x \leftarrow A^{\Gamma_H} x$	S, D, C, Z
xTPSV (UPLO, TRANS, DIAG, N, AP, X, INCX)	$x \leftarrow A^{\Gamma_1} x, x \leftarrow A^{\Gamma_T} x, x \leftarrow A^{\Gamma_H} x$	S, D, C, Z
options dim scalar vector vector matrix		
xGER (M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha x y^T + A, A - m \times n$	S, D
xGERU (M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha x y^T + A, A - m \times n$	C, Z
xGERC (M, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha x y^H + A, A - m \times n$	C, Z
xHER (UPLO, N, ALPHA, X, INCX, A, LDA)	$A \leftarrow \alpha x x^H + A$	C, Z
xHPR (UPLO, N, ALPHA, X, INCX, AP)	$A \leftarrow \alpha x x^H + A$	C, Z
xHER2 (UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$	C, Z
xHPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, AP)	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$	C, Z
xSYR (UPLO, N, ALPHA, X, INCX, A, LDA)	$A \leftarrow \alpha x x^T + A$	S, D
xSPR (UPLO, N, ALPHA, X, INCX, AP)	$A \leftarrow \alpha x x^T + A$	S, D
xSYR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, A, LDA)	$A \leftarrow \alpha x y^T + \alpha y x^T + A$	S, D
xSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, AP)	$A \leftarrow \alpha x y^T + \alpha y x^T + A$	S, D

Level 3 BLAS

options dim scalar matrix matrix scalar matrix		
xGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha op(A) op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
xSYMM (SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
xHEMM (SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
xSYRK (UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
xHERK (UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
xSYR2K(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$	S, D, C, Z
xHER2K(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$	C, Z
xTRMM (SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
xTRSM (SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A^{\Gamma_1})B, B \leftarrow \alpha Bop(A^{\Gamma_1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

Emanuel Gull

SIMONS FOUNDATION

MICHIGAN

L A P A C K
L -A P -A C -K
L A P A -C -K
L -A P -A -C K
L A -P -A C K
L -A -P A C -K

LAPACK

- LAPACK: Linear Algebra Package
- de facto standard interface for dense matrix linear algebra solvers
- pmmaaa: p for precision, mm for matrix type, and aaa for the algorithm. For example: dsysv: double precision symmetric solver. sgetrf: single precision general matrix factorization.
- depends on BLAS, these days mostly packaged/shipped with BLAS
- …link as you would link BLAS…
- Contains:
  - Triangular factorization
  - Orthogonal factorization (standard and generalized)
  - Eigenvalues
  - Linear System Solvers
  - Linear Least Squares minimizers (with and w/o constraints)

- …all of this for general/symmetric/hermitian/orthogonal/unitary/diagonal/triangular matrices!

HDF5

- HDF5: Hierarchical data format, version 5
- de facto standard for binary data storage
- Machine independent efficient storage

- Especially useful for large data sets
- Parallel adaptations (layered on top of MPI) exist

- Language independent: C, C++, Fortran interface
- check out alps hdf5 library

- Use the same way you would use directories and files on your computer:
  - Create nodes
  - Write data to nodes
  - Write attributes to data

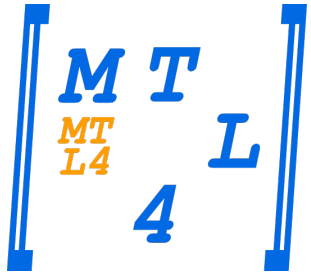
- Automatically translates endianness (byte ordering)

Eigen (C++)

- One of many matrix libraries. So why this one?

- Efficient
- Most of the linear algebra functionality you might want
- Elegant and straightforward API
- very efficient code generation

- …turned out to be the most natural of all matrix libraries I've seen so far. Alternatives are:
  - Boost UBLAS (do NOT use, very slow and clumsy, huge mess)!
  - MTL (the Matrix Template Library): somewhat clumsy interface, not well thought through.
  - Armadillo



Boost (C++)

- C++ general purpose libraries
- Difficult to use
- Know what to use and what to avoid. If in doubt: avoid!

- All sorts of convenience libraries:
  - program options (beautiful command line parsing)
  - random
  - filesystem
  - chrono

- Advanced C++ programming libraries:
  - MPL (template meta programming)

- MPI and Python dependencies pulled in for boost mpi/boost python

- It's easy to overuse techniques like meta programming. Beware of compile time issues. Too much boost and your code will become unreadable / unmaintainable!

Domain specific libraries: ALPS

- ALPS: Applications and Libraries for Physics Simulations
- Simulation applications:
  - spin systems
  - exact diagonalization
  - sparse diagonalization
  - drmg
  - tensor networks
  - dmft

- Basic Monte Carlo libraries
- Basic HPC Libraries
- Versatile C++ HDF5 interface

- ALPSCore: redeveloped ALPS for HPC applications: www.alpscore.org

Domain specific libraries: iTensor

See Miles' talk & tutorial
today & tomorrow &
Friday

Domain specific libraries: TRIQS

See Olivier's talks &
tutorials today & tomorrow

Emanuel Gull

SIMONS FOUNDATION

MICHIGAN

Tools and Infrastructure

Who here is:

- Using an IDE? (eclipse? xcode? kdevelop?) Who is using just editors? (vi? emacs?)
- Using a documentation tool?
- Using a revision control system? (git? svn? cvs? mercurial?)
- Using a unit test system? (which one?)
- Using a build script? Makefile? CMake?
- Using a debugger? ‘cout/printf’?
- Using a memory debugger every now and then?

- IT professionals have developed countless tools over the last four decades that make developing code easier, faster, and safer. Use them! This is just a short overview with some pointers.

vi? emacs?
seriously?

IDEs

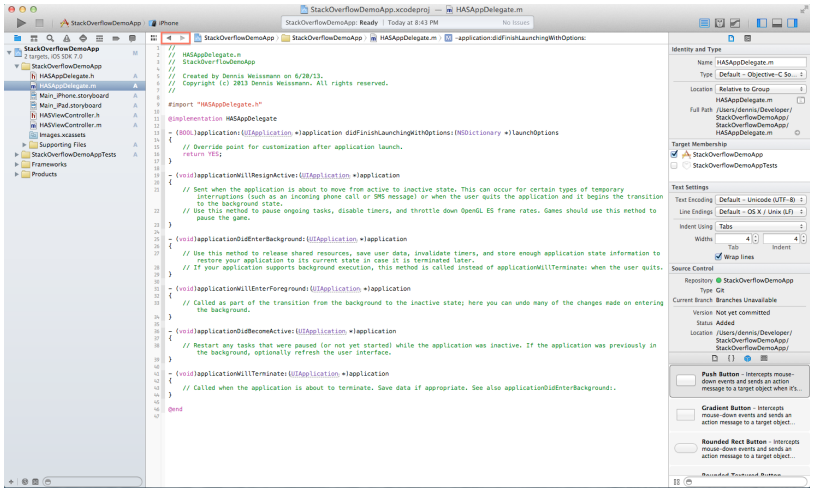
Eclipse / Eclipse CDT: written in java. Used to be very slow. Useful these days also for C++. If you've tried it last time a few years back: try again.

Default on OSX. Weak refactoring tools in C++, good integration with compilers.

KDevelop: KDE standard ide.

Visual studio on windows

IDEs: why bother?



- Your IDE shows additional information:
  - Class hierarchies
  - Function completion
  - compilation / syntax check as you go

- Is integrated:
  - debug, compile, run, within seconds
    - we'll come back to this in TDD
  - integration with versioning system

- It takes about a day for you to set up an IDE, a week to get comfortable, a month to get faster – but you will be much faster.

Version control

- You should use version control. No excuses.
- You should use git.
- For public, open source codes: get a github repository
- For closed scientific codes: register for an educational github repo
  - Do NOT stand up your own svn/git servers!
  - Who worries about backups & maintenance five years down the road?

- If you're using cvs or subversion: convert to git

- Do you know how to branch, merge, rebase, resolve conflicts, etc?
  - We will have a practice session on Friday.
- Use a graphical user interface to see what's going on on the repo:
  - Sourcetree
  - GitHub

Why git

- Decentralized / Independent of central repository
- Very easy to branch, very easy to merge:
  - Use branch per issue:
    1. create branch
    2. fix bug / implement feature
    3. merge back to master
- Commit often
- Never commit if it does not compile
- All tests should always pass
  - more about tests and TDD at the end of this talk & Next slide

Test framework: Google test (more about unit tests later)

- tests for a class

```
#include"gtest/gtest.h"
#include <IsingSimulation.h>

TEST(IsingSimulation, CreateIsingSimulation){
  int L=4;
  double T=2.1;
  IsingSimulation sim(L,T);
}
TEST(IsingSimulation, CanGetLT){
  int L=4;
  double T=2.1;
  IsingSimulation sim(L,T);
  EXPECT_EQ(L,sim.L());
  EXPECT_EQ(T,sim.T());
}
TEST(IsingSimulation, FlipProbabilitiesOfAllNeighborConfigs){
  int L=4;
  double T=1;
  IsingSimulation sim(L,T);
  EXPECT_DOUBLE_EQ(sim.flipProbability(0,0),std::exp(-8/T));
  sim.flip(0,0);
  EXPECT_DOUBLE_EQ(sim.flipProbability(0,1),std::exp(-4/T));
  sim.flip(0,1);
  sim.flip(0,0);
  sim.flip(0,L-1);
  EXPECT_DOUBLE_EQ(sim.flipProbability(0,0),1);
  sim.flip(1,L-1);
  EXPECT_DOUBLE_EQ(sim.flipProbability(0,0),1);
  sim.flip(1,0);
  EXPECT_DOUBLE_EQ(sim.flipProbability(0,0),1);
}
```

- main program for tests

```
#include"gtest/gtest.h"
#include"IsingSimulation.h"
int main(int argc, char **argv){

  ::testing::InitGoogleTest(&argc, argv);
  exit(RUN_ALL_TESTS());
```

- typical test output

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from SpinConfig
[ RUN      ] SpinConfig.CreateSpinConfig
[       OK ] SpinConfig.CreateSpinConfig (0 ms)
[ RUN      ] SpinConfig.ComputeEnergyAllSpinsUp
IsingTest.cpp:15: Failure
Value of: config.totalEnergy()
  Actual: 0
Expected: 2*4*4
Which is: 32
[  FAILED  ] SpinConfig.ComputeEnergyAllSpinsUp (0 ms)
```

- Make it easy to run unit tests

Documentation: Doxygen

- Generate documentation directly from code
- See doxygen.org
- Typical output: html and latex
- Doxygen comments delineated in C++ by:
  - /// Doxygen comment line
  - /** Comment
    …. block */
- In Science: use LaTeX to type the equation a function implements
- Configuration by a Doxyfile
- User interfaces to configure doxygen

- Careful with templates: usually does not work well

```
150
151 ///human readable verion of write_single_freq. Puts a bosonic Green's function into a text file.
152 void bosonic_matsubara_function::write_single_freq_hr(const std::string &out_file_name_gf) const{
```

Building: CMake

- How do you compile your codes?
  - g++ main.cc?
  - make?
  - ./configure, make, make install?
  - What if dependencies change?

- What if the machine changes?
  - New cluster
  - New compiler
  - New libraries?
  - Write a CMakeLists.txt file
  - Run:
    - cmake

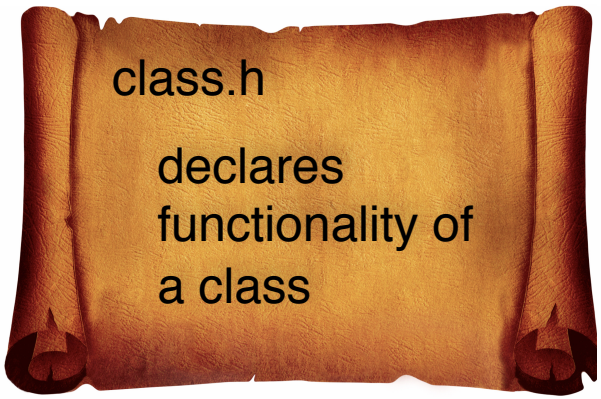
      …followed by
    - make
    - Dependencies automatically updated. See tutorial on Sunday

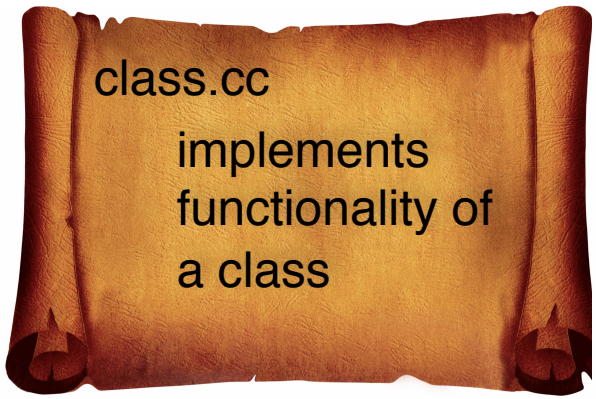
Automatic Building and Testing

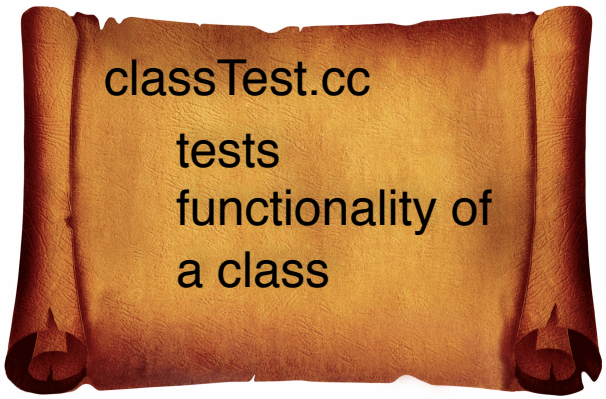
- How do you know that your code works?
- Two or more people with commit rights: How do you know that nobody breaks your code?
- If you're writing a library:
  - how do you know it works for other people?
  - how do you know it works on exotic machines?
- If you're using a library that frequently changes:
  - how do you know you're compatible with the newest version?
  - how do you know that updates don't break the code?
- Continuous build:
  - Make a change, test it locally
  - Check it in
  - Compile and run tests on a range of platforms to verify that it behaves as expected

Testing your code

- 2 types of tests for codes:
  - Integration / acceptance tests: code gives an expected non-trivial result.
  - Unit tests: small tests that check that a function or a class behaves as intended
- Unit tests: One of the big paradigm shifts in software engineering in the last 10 years
- All of us do regular and detailed testing to ‘make sure that the code works’. Let’s formalize this:







Unit tests: Philosophy

- Imagine: what if all the important lines of code were tested?
- Imagine: what if it was very easy to rerun these tests?

- If you trust that your tests check that a program is correct, you can always run them and make sure:
  - complete confidence in code
  - Bugs breaking old code while extending new code automatically detected
  - …and immediately fixed!

- Unit tests cut down the debug time, at a small cost of increased development time and more code.

- Pay off whenever we spend a long time debugging and little time writing code.

Unit tests: Benefits

- Imagine: what if all the important lines of code were tested?
- Imagine: what if it was very easy to rerun these tests?

- Imagine how fearless you would be in changing your code:
  - you know that you're going to detect all problems after a change
  - if you break something, you can immediately fix it!

- This allows you to refactor: change your code so that the structure is better, easier to read, easier to maintain.

- Stops the decay of your code!

- 'Boy scouts rule': Leave your campground cleaner than you found it!

Unit tests: decoupling

- Unit tests test small features
- Being able to test small features means being able to decouple code
- Unit tests force you to write modular codes with clear dependencies and boundaries
- Modular code is easier to maintain…
- Tests are removed from the main code: that's what the unit test framework does for you!
- Production code will never know that they exist
- Unit tests are the best possible documentation for your code: show every single way a class can be accessed and used.

Unit tests: Limitations

- Unit tests will not cure all bugs!
- Logic problems are still there, but the small computing problems can be fixed
- Unit tests can introduce their own problems:
  - too slow to execute (will not be run – typical time < 10 ms)
  - too much overhead and baggage

Unit tests: How to debug?

- Assume you find a bug in your code.
- First step: write a test that fails
- Keep writing more tests. Some will pass, some will fail. Failed tests show where the bug is
- Fix the bug, all tests pass
- Clean up the tests, leave them in your test framework so you will know if the bug comes back. Clean up the code – all tests still pass.

Scientific tests: the inverse problem

- In science, we often have
  - difficult forward problem
  - easy inverse problem

- example: factorization of a number

$$18161 =$$

- solution is difficult, but the validation of the solution is easy:

$$18161 = 11 * 13 * 127$$

- These are ideal cases for tests!

Scientific unit tests: the special cases

- Check your extreme behavior!
- Can you do something analytically?
- special cases for
  - U=0?
  - n=0?
  - T=0?
- Use this as test case.
- [Don’t write tests that don’t test anything]

Test Driven Development (TDD)

- Having tests is very nice: you always know that the code works as intended
- Only we never put them:
  - They clutter up the code
    - use unit test framework to separate them out
  - There really is no need, because once we've tested a routine by hand we know that it works
  - Writing tests takes time that we otherwise would use to code!
- How do we cover our code completely with tests?

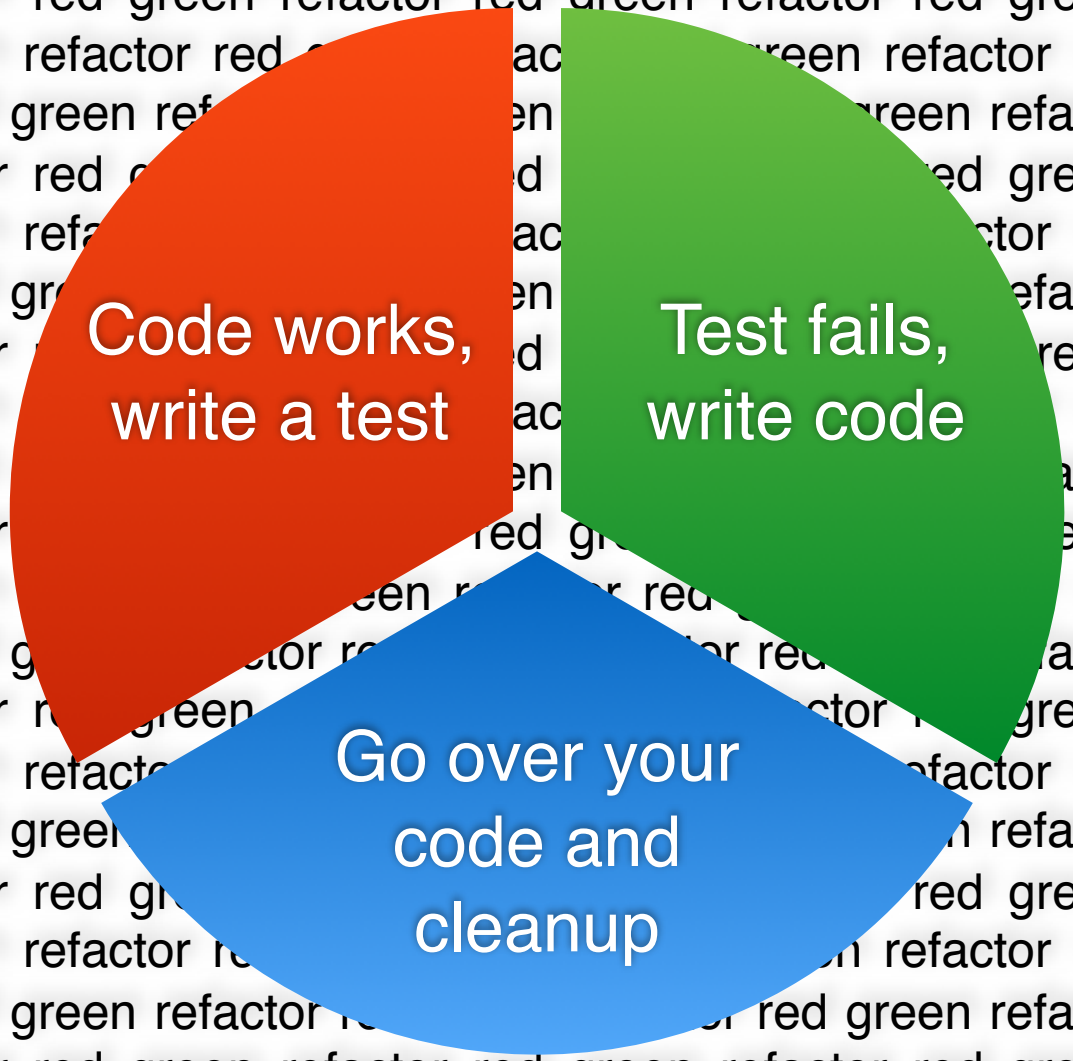
Test Driven Development (TDD): Philosophy

- Discipline! **Write your tests first, then your code**:

- You are not allowed to write any production code unless there is a failing test. Not compiling is failing.
- As soon as you have a failing test, you have to write code to make it pass
- You are not allowed to write more code than is needed to make your test pass.

- …Sounds like useless theory but works very well in practice.
- …will slow you down a little in the short run, but speed you up a lot in the long run!

Test Driven Development (TDD): The Red Green Refactor loop



Test Driven Development (TDD): The Red phase

- Starting point: a working and clean code, all tests pass.
- Think about the feature you would like to have.
- Find the a small step towards that feature, e.g. a new function.
- Write a test for that function

```
13 TEST(SpinConfig, ComputeEnergySpinsFlipped){
14   int L=4;
15   SpinConfig config(L);
16   config.flip(1,1);
17   EXPECT_EQ(-2*L*L+8, config.totalEnergy());
18   config.flip(1,2);
19   EXPECT_EQ(-2*L*L+12, config.totalEnergy());
20   config.flip(3,3);
21   EXPECT_EQ(-2*L*L+8+4+8, config.totalEnergy());
22 }
```

- first iteration of this test stopped here
- later additional tests were added

Test Driven Development (TDD): The Green phase

- Starting point: a test that fails
- What do you need to do to implement that feature?
- Make the test pass!

```
class SpinConfig{
  public:
  SpinConfig(int L){
    L_=L;
    spins_.resize(L*L, up);
    energy_=totalEnergy();
    magnetization_=totalMagnetization();
  };
```

```
int SpinConfig::totalEnergy() const{
  int e=0;
  for(int i=0;i<L_;++i){
    for(int j=0;j<L_;++j){
      e-=(spins_[at(i,j)]*spins_[right(i,j)]);
      e-=(spins_[at(i,j)]*spins_[below(i,j)]);
    }
  }
  return e;
}
```

- As soon as the test passes, you have to stop writing code!
- No more functionality than is needed to make the test pass!

Test Driven Development (TDD): The Refactor (Blue) phase

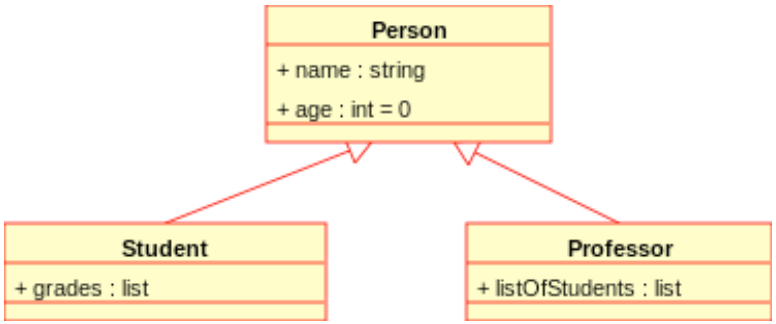
- Refactoring is a fancy word for cleanup
- First step: go to your production code, clean:
  - extract duplicate code into functions
  - remove unused code
  - go over function and variable names, make sure they are descriptive and clear
- Second step: repeat this with your unit tests:
  - cleaning up the tests is as important as cleaning up your code
- During refactoring, your code always remains correct: all of your tests will always pass!
- Refactoring is where the IDE shows its full power!

Consequences of TDD

- Having to test your code early on means it always works.
- Having to write small independent tests means that your code does not develop long-range dependencies: everything stays small and encapsulated.
  - modules stay small.
- Debugging gets simpler: write a failing test, then make it pass

OO and why we care

- Object orientation means:
  - Dynamic dispatch
  - Encapsulation
  - Polymorphism
  - Inheritance
  - Open recursion



- Most Important for scientific codes:
  - Encapsulation (think private and public variables)
  - Polymorphism (think virtual functions)

Encapsulation

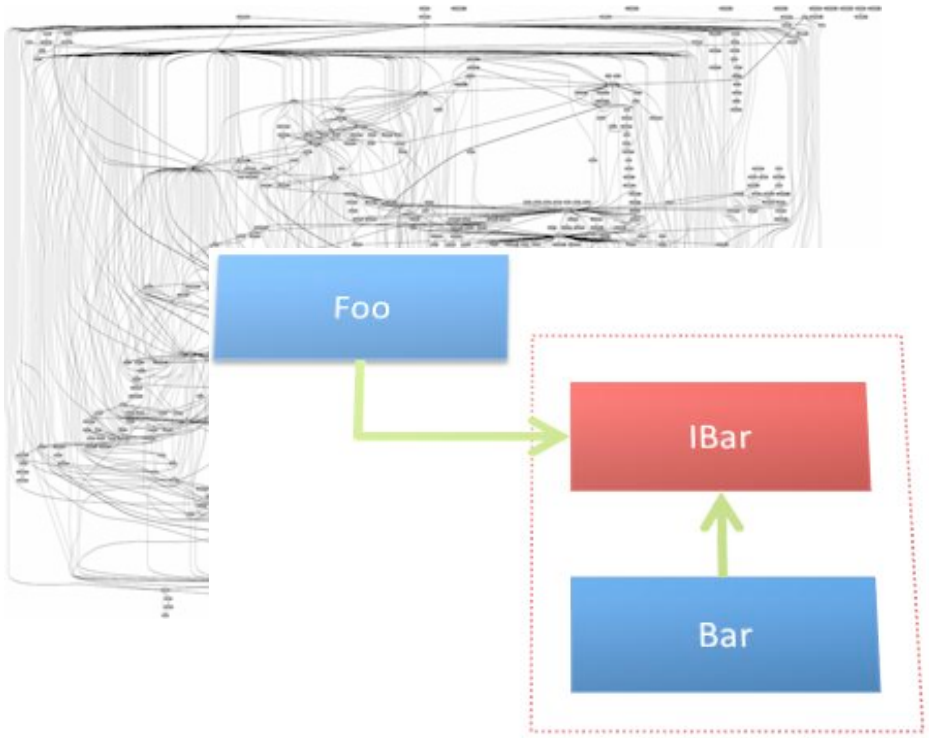
```
class SpinConfig{
  public:
  SpinConfig(int L){
    L_=L;
    spins_.resize(L*L, up);
    energy_=totalEnergy();
    magnetization_=totalMagnetization();
  };
  int totalEnergy() const;
  int energySpinFlip(int i, int j) const;
  void flip(int i, int j);
  int energy() const{return energy_;}
  int totalMagnetization() const;
  int magnetization() const{return magnetization_;}

  private:
  int at(int i, int j) const{ return i*L_+j;}
  int left(int i, int j) const{ return i*L_+(j-1+L_)%L_;}
  int right(int i, int j) const{ return i*L_+(j+1)%L_;}
  int above(int i, int j) const{ return ((i-1+L_)%L_)*L_+j;}
  int below(int i, int j) const{ return ((i+1)%L_)*L_+j;}
  int energy_;
  int magnetization_;
  std::vector<int> spins_;
  int L_;
};
```

- Only the public part is visible to the outside.
- That means all the operations on that class go through the public interface
- If we're just using the class, we don't need to worry about the private stuff!

- In fact only flip() will change the state of this class
- The less code to read through, the faster the coding!
- Unit tests will test the public interface.

Polymorphism

- Polymorphism allows for dependency inversion. Dependency inversion allows to decouple codes!



- This is an actual code – Microsoft IIS (Internet Information Server)
- Everything depends on everything else!
- With polymorphism we can make functions/classes/etc depend on interfaces, and use polymorphic dispatch to implement those interfaces.
- This leads to clean boundaries in the code; rest of code will only need to know about the interface (‘the abstraction’)

How to deal with legacy codes

- Best definition of legacy codes: codes without unit tests
- Typical scenario: you start working on a project. A code exists, but it ‘is a mess’.
- How do you proceed to make changes to the code?
  - add functionality
  - fix bugs
- How do you make sure that you don’t break anything?
- Should you rewrite or should you change the existing code base?
- Assume that we chance the existing code… but it is a mess…

Step one: Integration into a unit test framework

- Download google test (or your favorite unit test framework)

https://code.google.com/p/googletest/

- Introduce a new makefile target that executes the unit test framework, or a way of running the code to invoke the framework.
- Compile and link to it

```
62  if (vm.count("test")){
63    ::testing::InitGoogleTest(&argc, argv);
64    exit(RUN_ALL_TESTS());
```

```
5  #include"g4_phpp.h"
6  #include"gtest/gtest.h"
```

```
///Parameter input
std::string task_type_string;
po::options_description desc( "Allow
desc.add_options()("lattice",po::val
("basename",po::value < std::string>
("sim",po::value < std::string > (&s
("sigma",po::value < std::string > (
("test", "run unit tests")
("n_omega", po::value<int>(&n_omega4
("n_Omega", po::value<int>(&n_omega4
("task", po::value < std::string > (
```

- Change your IDE / environment to compile and execute the tests by default

Step two: find the place where changes need to be done

- …analyze enough of the code to understand the approximate region where changes will need to go.

Step three: cover the part that needs to be extended with unit tests

- Test that part of the code for all imaginable scenarios
- Cover the part of the code with unit tests
- Make sure all the tests pass
- Make sure the tests test something sensible and are actually useful and correct.
- Only once you have confidence in your tests you should start working on the code.

Step four: refactor until the code is clean

- Clean code reads like well written prose

- Go to the part of the code that will need to be changed
- Bring the code into a state where it is easy to read and work with:
  - rename variables
  - decouple codes
  - extract functions
  - extract classes
  - make sure the code is modular and decoupled
- During the entire process all the tests should pass
- This requires confidence in your tests!

Step five: add new functionality using TDD

- Finally you get to code the new functionality
- During the entire process all the tests should pass
- Use TDD: tests first, then code. All tests always pass, code remains correct.
- This requires confidence in your tests!
- Don’t forget the refactoring step: red green refactor cycle, code cleanup etc!

…and if we have time: TDD Ising…

- Homework: write a 2d Ising model using TDD!