How to write good code and why it matters

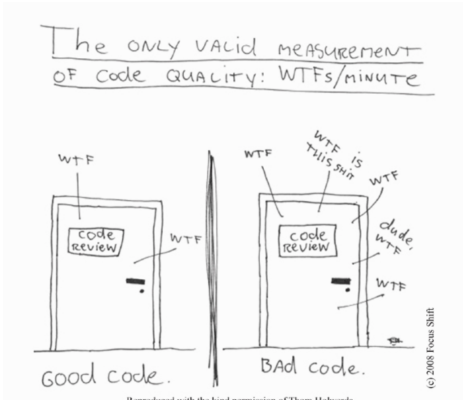
Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no[1,2]

[1]National Superconducting Cyclotron Laboratory and Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, USA
[2]Department of Physics, University of Oslo, Oslo, Norway

Aug 13, 2020

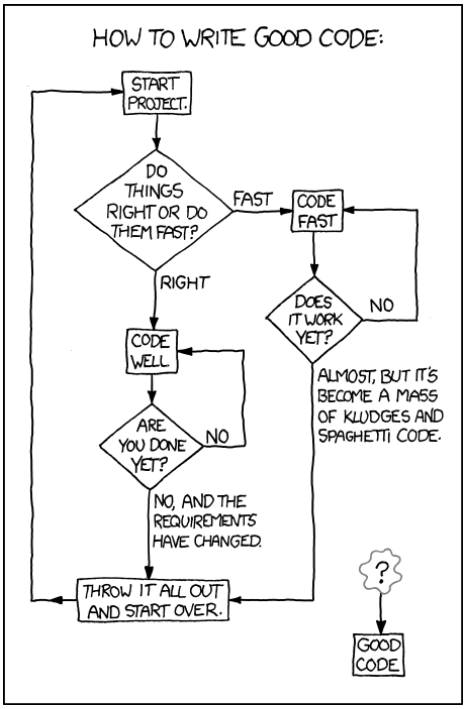
What is the only test of a properly written code?

The number of WTFs/minute



© 1999-2020, Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no. Released under CC Attribution-NonCommercial 4.0 license

How to write good code



Disasters attributable to bad numerical computing

Have you been paying attention in your numerical analysis or scientific computation courses? If not, it could be a costly mistake. Here are some real life examples of what can happen when numerical algorithms are not correctly applied.

1. The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.

2. The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple overflow.

3. The sinking of the Sleipner A offshore platform in Gandsfjorden near Stavanger, Norway, on August 23, 1991, resulted in a loss of nearly one billion dollars. It was found to be the result of inaccurate finite element analysis.

Why is computing competence important and clean code important?

Definition of computing. With computing I will mean solving scientific problems using computers. It covers numerical, analytical as well as symbolic computing. Computing is also about developing an understanding of the scientific process by enhancing algorithmic thinking when solving problems. Well written code is a piece of art by itself and expresses clarity.

Computing competence, what is it?

Computing competence has always been a central part of the science and engineering education. Traditionally, such competence meant mastering mathematical methods to solve science problems - by pen and paper.

Today **we** are expected to use all available tools to solve scientific problems; computers primarily, but also pen and paper.

I will use the term/word algorithms in the broad meaning: methods (for example mathematical) to solve science problems, with and without computers.

Continuous versus discrete

Algorithms involving pen and paper are traditionally aimed at what we often refer to as continuous models.

Application of computers calls for approximate discrete models.

Much of the development of methods for continuous models are now being replaced by methods for discrete models in science and industry, simply because much larger problem classes can be addressed with discrete models, often also by simpler and more generic methodologies. However, verification of algorithms and understanding their limitations requires much of the classical knowledge about continuous models.

Key principle in scientific modeling

The power of the scientific method lies in identifying a given problem as a special case of an abstract class of problems, identifying general solution methods for this class of problems, and applying a general method to the specific problem (applying means, in the case of computing, calculations by pen and paper, symbolic computing, or numerical computing by ready-made and/or self-written software). This generic view on problems and methods is particularly important for understanding how to apply available, generic software to solve a particular problem.

Computing competence is central to solving scientific problems

Definition of computing. Computing competence represents a central element in scientific problem solving, from basic education and research to essentially almost all advanced problems in modern societies. Computing competence is simply central to further progress. It enlarges the body of tools available to students and scientists beyond classical tools and allows for a more generic handling of problems. Focusing on algorithmic aspects results in deeper insights about scientific problems.

Today's project in science and industry tend to involve larger teams. Tools for reliable collaboration must therefore be mastered (e.g., version control systems, automated computer experiments for reproducibility, software and method documentation). **In order to be efficient and to have code which can be extended upon, clean code matters even more**.

What is computing competence about?

Computing competence is about

1. derivation, verification, and implementation of algorithms
2. understanding what can go wrong with algorithms
3. overview of important, known algorithms
4. understanding how algorithms are used to solve mathematical problems
5. reproducible science and ethics
6. algorithmic thinking for gaining deeper insights about scientific problems
7. computing competence is also about writing clear code

Computing is understanding.

Getting started: How to use Symbolic tools to verify your code

Integration by Trapezoidal Rule.

- The algorithm for computing the integral vha the Trapezoidal rule for an interval $x \in [a, b]$

$$\int_a^b (f(x)dx \approx \frac{1}{2}\left[f(a) + 2f(a+h) + \cdots + 2f(b-h) + f(b)\right]$$

The quantity h is the so-called step length defined as

$$h = \frac{b-a}{n}$$

with n being the number of integration points. Ideally we want h as small as possible. The mathematical error goes like $\sim O(h^2)$ (global error).

Typical implementation

Integration by Trapezoidal Rule.

```
from math import exp, log
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s

def f1(x):
    return exp(-x*x)*log(1+x*sin(x))

a = 1;  b = 3; n = 1000
result = Trapez(a,b,f1,n)
print result
```

Symbolic calculations and numerical calculations in one code

Python offers an extremely versatile programming environment, allowing for the inclusion of analytical studies in a numerical program. Here we show an example code with the **trapezoidal rule** using **SymPy** to evaluate an integral and compute the absolute error with respect to the numerically evaluated one of the integral $4\int_0^1 dx/(1+x^2) = \pi$:

```
from sympy import Symbol, integrate
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s

#  function to compute pi
def function(x):
    return 4.0/(1+x*x)

a = 0.0;  b = 1.0; n = 100
result = Trapez(a,b,function,n)
print "Trapezoidal rule=", result
# define x as a symbol to be used by sympy
x = Symbol('x')
exact = integrate(function(x), (x, 0.0, 1.0))
print "Sympy integration=", exact
# Find relative error
print "Relative error", abs((exact-result)/exact)
```

Error analysis

The following extended version of the trapezoidal rule allows you to plot the relative error by comparing with the exact result. By increasing to 10^8 points one arrives at a region where numerical errors start to accumulate.

```
from math import log10
import numpy as np
from sympy import Symbol, integrate
import matplotlib.pyplot as plt
# function for the trapezoidal rule
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s
#  function to compute pi
def function(x):
    return 4.0/(1+x*x)
# define integration limits
a = 0.0;  b = 1.0;
# find result from sympy
# define x as a symbol to be used by sympy
x = Symbol('x')
#exact = integrate(function(x), (x, a, b))
exact = pi
# set up the arrays for plotting the relative error
n = np.zeros(9); y = np.zeros(9);
# find the relative error as function of integration points
for i in range(1, 8, 1):
    npts = 10**i
    result = Trapez(a,b,function,npts)
    RelativeError = abs((exact-result)/exact)
    n[i] = log10(npts); y[i] = log10(RelativeError);
plt.plot(n,y, 'ro')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.show()
```

Integrating numerical mathematics with calculus

The last example shows the potential of combining numerical algorithms with symbolic calculations, allowing you to

- Validate and verify your algorithms.
- Including concepts like unit testing, one has the possibility to test and validate several or all parts of the code.
- Validation and verification are then included *naturally* and one can develop a better attitude to what is meant with an ethically sound scientific approach.

- The above example allows you to also test the mathematical error of the algorithm for the trapezoidal rule by changing the number of integration points. From day one start to think error analysis.

How to automatize and autogenerate code with sympy

For the computation of various derivatives with different types of wave functions, you will find it useful to use python with symbolic python, that is sympy, see online manual. Using sympy allows you autogenerate both Latex code as well c++, python or Fortran codes. Here you will find some simple examples. We want to compute various derivatives of hydrogen-like single-particle state functions like the $2s$ hydrogen-orbital

$$\phi_{2s}(\boldsymbol{r}) = (Zr - 2)\exp -(\frac{1}{2}Zr),$$

with $r^2 = x^2 + y^2 + z^2$.

```
from sympy import symbols, diff, exp, sqrt
x, y, z, Z = symbols('x y z Z')
r = sqrt(x*x + y*y + z*z)
r
phi = (Z*r - 2)*exp(-Z*r/2)
phi
diff(phi, x)
```

We can improve our first attempt

We can improve our output by factorizing and substituting expressions

```
from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing
x, y, z, Z = symbols('x y z Z')
r = sqrt(x*x + y*y + z*z)
phi = (Z*r - 2)*exp(-Z*r/2)
R = Symbol('r') #Creates a symbolic equivalent of r
#print latex and c++ code
print printing.latex(diff(phi, x).factor().subs(r, R))
print printing.ccode(diff(phi, x).factor().subs(r, R))
```

And second derivatives

We can in turn look at second derivatives

```
from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing
x, y, z, Z = symbols('x y z Z')
r = sqrt(x*x + y*y + z*z)
phi = (Z*r - 2)*exp(-Z*r/2)
R = Symbol('r') #Creates a symbolic equivalent of r
(diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().subs(r, R)
# Collect the Z values
(diff(diff(phi, x), x) + diff(diff(phi, y), y) +diff(diff(phi, z), z)).factor().collect(Z).subs(r,
# Factorize also the r**2 terms
(diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).subs(r
print printing.ccode((diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).facto
```

With some practice this allows one to be able to check one's own calculation and translate automatically into code lines. Saves a lot of time and is much less error prone.

Old dusty Fortran deck, this function is 509 lines long

```
      REAL FUNCTION GMOSH*8(N,L,NC,LC,N1,L1,N2,L2,LR,D)
C GENERALIZED TRANSFORMATION BRACKETS WRITTEN BY XXX!
C REF  M.SOTONA AND M.GMITRO  COMP.PHYS.COMM 3(1972)53
C  D=MASS1/MASS2
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION F(50),G(50),W(50)
      IORD=0
      ZERO=0.0D0
      HALF=0.5D0
      EIN=1.0D0
      IF(IORD-10) 5,6,5
    5 IORD=10
      F(1)=ZERO
      G(1)=DLOG(HALF)
      W(1)=ZERO
      DO 10 I=2,50
      A=I-1
      F(I)=F(I-1)+DLOG(A)
      G(I)=G(I-1)+DLOG(A+HALF)
   10 W(I)=DLOG(A+A+EIN)
    6 GMOSH=ZERO
      IF(N+N+NC+NC+L+LC-N1-N1-N2-N2-L1-L2) 500,12,500
   12 IF(L+LC-LR) 500,13,13
   13 IF(L1+L2-LR) 500,14,14
   14 IF(IABS(L-LC)-LR) 15,15,500
   15 IF(IABS(L1-L2)-LR) 16,16,500
   16 DL=DLOG(D)
      D1L=DLOG(D+EIN)
C  it goes on like this, in total 509 lines.
```

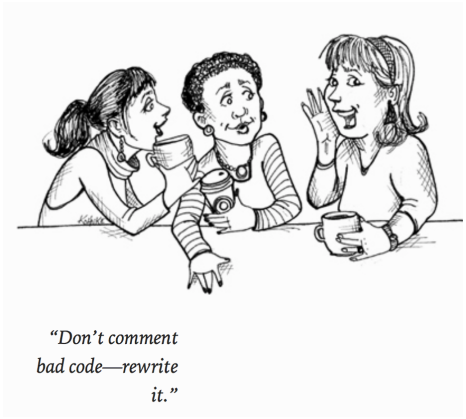
Yet another Fortran babe

```
      SUBROUTINE argonne(j,l,lp,is,v1,v2,v3,r,n)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION v1(500),v2(500),v3(500),r(500)
      data cte/3.72681d0/,xmu/0.6995d0/

      DO i=1,n
         v1(i)=0.d0
         v2(i)=0.d0
         v3(i)=0.d0
      ENDDO
      it=1-MOD(l+is,2)
      xs=DFLOAT(is)
      xj=DFLOAT(j)
      xl=DFLOAT(l)
      xlp=DFLOAT(lp)
      xit=DFLOAT(it)
```

```
 xb=(xj*(xj+1.d0)-xl*(xl+1.d0)-xs*(xs+1.d0))*0.5d0
 xq=xl*(xl+1.d0)
 yspin=4.d0*xs-3.d0
 yisos=4.d0*xit-3.d0
 xqp=xlp*(xlp+1.d0)
 xbp=(xj*(xj+1.d0)-xlp*(xlp+1.d0)-xs*(xs+1.d0))*0.5d0
 IF(l.EQ.j) xten=2.d0
 IF(l.EQ.j+1) xten=-2.d0*(xj+2.d0)/(2.d0*xj+1.d0)
 IF(l.EQ.j-1) xten=-2.d0*(xj-1.d0)/(2.d0*xj+1.d0)
 xten=xten*xs
 vaux=-4.801125d0+yisos*0.798925+1.189325*yspin
&     +0.182875*yisos*yspin-0.1575*xten-
&     0.7525*xten*yisos+0.5625*xb+0.0475*xb*yisos
&     +0.070625*xq-0.148125*xq*yisos-0.040625*
&     xq*yspin-0.001875*xq*yisos*yspin
&     -0.5425*xb*xb+0.0025*xb*xb*yisos
  vaux1=2061.5625-477.3125*yisos-502.3125*yspin+
&       97.0625*yisos*yspin+108.75*xten+297.25*xten*yisos
&       -719.75*xb-159.25*xb*yisos+8.625*xq+5.625*xq*yisos+
&       17.375*xq*yspin-33.625*xq*yisos*yspin+391. *xb*xb+145.0
&       *xb*xb*yisos
  DO i=1,n
     xr=r(i)
     xexp=DEXP(-xmu*xr)/(xmu*xr)
     xgauss=1.d0-DEXP(-2.d0*xr*xr)
     y=xexp*xgauss
     t=y*xgauss*(1.d0+3.d0/(xmu*xr)+3.d0/(xmu*xmu*xr*xr))
     tt=t*t
     ws=1.d0/(1.d0+DEXP((xr-0.5d0)/0.2d0))
     v1(i)=vaux*tt+
&          vaux1*ws+cte*yspin*yisos*y+cte*t*xten*yisos
  ENDDO
  IF(l.NE.lp) THEN
     xten=-2.d0*(xj+2.d0)/(2.d0*xj+1.d0)
     xten=xten*xs
     vaux=-4.801125d0+yisos*0.798925+1.189325*yspin
&         +0.182875*yisos*yspin-0.1575*xten-
&          0.7525*xten*yisos+0.5625*xbp+0.0475*xbp*yisos
&         +0.070625*xqp-0.148125*xqp*yisos-0.040625*
&         xqp*yspin-0.001875*xqp*yisos*yspin
&         -0.5425*xbp*xbp+0.0025*xbp*xbp*yisos
     vaux1=2061.5625-477.3125*yisos-502.3125*yspin+
&          97.0625*yisos*yspin+108.75*xten+297.25*xten*yisos
&          -719.75*xbp-159.25*xbp*yisos+8.625*xqp
&          +5.625*xqp*yisos+
&          17.375*xqp*yspin-33.625*xqp*yisos*yspin
&          +391. *xbp*xbp+145.0
&          *xbp*xbp*yisos
     xten2= 6.d0*DSQRT(xj*(xj+1.d0))/(2.d0*xj+1.d0)
     xten2=xten2*xs
     DO i=1,n
        xr=r(i)
        xexp=dexp(-xmu*xr)/(xmu*xr)
        xgauss=1.d0-dexp(-2.d0*xr*xr)
        y=xexp*xgauss
        t=y*xgauss*(1.d0+3.d0/(xmu*xr)+3.d0/(xmu*xmu*xr*xr))
        tt=t*t
        ws=1.d0/(1.d0+dexp((xr-0.5d0)/0.2d0))
        v2(i)=vaux*tt+vaux1*ws+cte*yspin*yisos*y+
&            cte*t*xten*yisos
        v3(i)=cte*t*xten2*yisos-0.1575*xten2*tt-
```

```
&             0.7525*xten2*yisos*tt
&             +108.75*xten2*ws+297.25*xten2*yisos*ws
     ENDDO
  ENDIF

  DO i=1,n
     v1(i)=v1(i)/41.47
     v2(i)=v2(i)/41.47
     v3(i)=v3(i)/41.47
  ENDDO
  RETURN
  END
```

Would more comments help here?



"Don't comment bad code—rewrite it."

Use proper names for variables, comments may not be needed

```
// Main program begins here

int main(int argc, char* argv[])
{
  string filename;
  int NumberSpins, MonteCarloCycles;
  double InitialTemp, FinalTemp, TempStep;
  .....
  // Start Monte Carlo sampling by looping over the selected Temperatures
  for (double Temperature = InitialTemp; Temperature <= FinalTemp; Temperature+=TempStep){
    vec ExpectationValues = zeros<mat>(5);
    // Start Monte Carlo computation and get expectation values
    MetropolisSampling(NumberSpins, MonteCarloCycles, Temperature, ExpectationValues);
    WriteResultstoFile(NumberSpins, MonteCarloCycles, Temperature, ExpectationValues);
```

```
  }
  ofile.close();  // close output file
  return 0;
}
```

Functions should be short and do as few operations as possible

Don't overload functions by transfering gazillions of variables and returning ditto, even this function has too many variables.

```
void InitializeLattice(int NumberSpins, mat &SpinMatrix,  double& Energy, double& MagneticMoment)
{
  // setup spin matrix and initial magnetization
  for(int x =0; x < NumberSpins; x++) {
    for (int y= 0; y < NumberSpins; y++){
      SpinMatrix(x,y) = 1.0; // spin orientation for the ground state
      MagneticMoment +=   SpinMatrix(x,y);
    }
  }
  // setup initial energy
  for(int x =0; x < NumberSpins; x++) {
    for (int y= 0; y < NumberSpins; y++){
      Energy -=  SpinMatrix(x,y)*(SpinMatrix(PeriodicBoundary(x,y-1)+SpinMatrix(x,y+1));
    }
  }
}// end function InitializeLattice
```

How could we improve upon this?

Comments, functions, classes, names and much more

- Avoid complicated function animals and transfering too many variables
- Use meaningful names, reduces the amount of clutter and need of comments
- Don't overcomplicate things, write clean and simple classes (important for high-performance computing)
- Avoid transfering many variables to functions, try to stay with as few as possible
- A function should return mainly one thing, that is do one thing.
- Keep as many variables as possible as private in classes

Important ingredients to have in codes

- Be able to validate and verify the algorithms.
- Include concepts like unit testing. Gives the possibility to test and validate several or all parts of the code.
- Validation and verification are then included *naturally* and one can develop a better attitude to what is meant with an ethically sound scientific approach.

A structured approach to solving problems

In the steps that lead to the development of clean code you should think of

1. How to structure a code in terms of functions (use IDEs or advanced text editors like sublime or atom)
2. How to make a module
3. How to read input data flexibly from the command line or files
4. How to create graphical/web user interfaces
5. How to write unit tests
6. How to refactor code in terms of classes (instead of functions only)
7. How to conduct and automate large-scale numerical experiments
8. How to write scientific reports in various formats (LaTeX, HTML, doconce)

Additional benefits

Many of the above aspetcs will save you a lot of time when you incrementally extend software over time from simpler to more complicated problems. In particular, you will benefit from many good habits:

1. New code is added in a modular fashion to a library (modules)
2. Programs are run through convenient user interfaces
3. It takes one quick command to let all your code undergo heavy testing
4. Tedious manual work with running programs is automated,
5. Your scientific investigations are reproducible, scientific reports with top quality typesetting are produced both for paper and electronic devices. Use version control software like git and repositories like github

Code quality

Unit Testing

Unit Testing is the practice of testing the smallest testable parts, called units, of an application individually and independently to determine if they behave exactly as expected.

Unit tests (short code fragments) are usually written such that they can be preformed at any time during the development to continually verify the behavior of the code.

In this way, possible bugs will be identified early in the development cycle, making the debugging at later stages much easier.

Unit Testing, benefits

There are many benefits associated with Unit Testing, such as

- It increases confidence in changing and maintaining code. Big changes can be made to the code quickly, since the tests will ensure that everything still is working properly.

- Since the code needs to be modular to make Unit Testing possible, the code will be easier to reuse. This improves the code design.
- Debugging is easier, since when a test fails, only the latest changes need to be debugged.
  - Different parts of a project can be tested without the need to wait for the other parts to be available.
- A unit test can serve as a documentation on the functionality of a unit of the code.

Simple example of unit test

Look up the guide on how to install unit tests for c++ at course webpage. This is the version with classes.

```
#include <unittest++/UnitTest++.h>

class MyMultiplyClass{
public:
    double multiply(double x, double y) {
        return x * y;
    }
};

TEST(MyMath) {
    MyMultiplyClass my;
    CHECK_EQUAL(56, my.multiply(7,8));
}

int main()
{
    return UnitTest::RunAllTests();
}
```

Simple example of unit test

And without classes

```
#include <unittest++/UnitTest++.h>


double multiply(double x, double y) {
    return x * y;
}

TEST(MyMath) {
    CHECK_EQUAL(56, multiply(7,8));
}

int main()
{
    return UnitTest::RunAllTests();
}
```

For Fortran users, the link at http://sourceforge.net/projects/fortranxunit/ contains a similar software for unit testing.

Unit tests

There are many types of **unit test** libraries. One which is very popular with C++ programmers is Catch

Catch is header only. All you need to do is drop the file(s) somewhere reachable from your project - either in some central location you can set your header search path to find, or directly into your project tree itself!

This is a particularly good option for other Open-Source projects that want to use Catch for their test suite.

Examples

Computing factorials

```
inline unsigned int Factorial( unsigned int number ) {
  return number > 1 ? Factorial(number-1)*number : 1;
}
```

Factorial Example

Simple test where we put everything in a single file

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main()
#include "catch.hpp"
inline unsigned int Factorial( unsigned int number ) {
  return number > 1 ? Factorial(number-1)*number : 1;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(0) == 1 );
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

This will compile to a complete executable which responds to command line arguments. If you just run it with no arguments it will execute all test cases (in this case there is just one), report any failures, report a summary of how many tests passed and failed and return the number of failed tests.

What did we do (1)?

All we did was

```
#define
```

one identifier and

```
#include
```

one header and we got everything - even an implementation of main() that will respond to command line arguments. Once you have more than one file with unit tests in you'll just need to

```
#include "catch.hpp"
```

and go. Usually it's a good idea to have a dedicated implementation file that just has

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp".
```

You can also provide your own implementation of main and drive Catch yourself.

What did we do (2)?

We introduce test cases with the

```
TEST_CASE
```

macro.

The test name must be unique. You can run sets of tests by specifying a wildcarded test name or a tag expression. All we did was **define** one identifier and **include** one header and we got everything.

We write our individual test assertions using the

```
REQUIRE
```

macro.

Unit test summary and testing approach

Three levels of tests

1. Microscopic level: testing small parts of code, use often unit test libraries
2. Mesoscopic level: testing the integration of various parts of your code
3. Macroscopic level: testing that the final result is ok

How do we time our code?

1. This is not trivial!
2. And why should we care and how is this related to writing good code?

We need to think of how data flows in our code and how we can utilize memory in the most efficient way. And avoid memory problems (which can lead to undesired problems).

Memory management

The main memory contains the program data

- Cache memory contains a copy of the main memory data
- Cache is faster but consumes more space and power. It is normally assumed to be much faster than main memory
- Registers contain working data only
  - Modern CPUs perform most or all operations only on data in register
- Multiple Cache memories contain a copy of the main memory data
  - Cache items accessed by their address in main memory
  - L1 cache is the fastest but has the least capacity
  - L2, L3 provide intermediate performance/size tradeoffs

Loads and stores to memory can be as important as floating point operations when we measure performance.

Memory and communication

- Most communication in a computer is carried out in chunks, blocks of bytes of data that move together
- In the memory hierarchy, data moves between memory and cache, and between different levels of cache, in groups called lines
  - Lines are typically 64-128 bytes, or 8-16 double precision words
  - Even if you do not use the data, it is moved and occupies space in the cache
- This performance feature is not captured in most programming languages

Measuring performance

How do we measure erformance? What is wrong with this code to time a loop?

```
clock_t start, finish;
start = clock();
for (int j = 0; j < i; j++) {
  a[j] = b[j]+b[j]*c[j];
}
finish = clock();
double timeused = (double) (finish - start)/(CLOCKS_PER_SEC );
```

Problems with measuring time

1. Timers are not infinitely accurate
2. All clocks have a granularity, the minimum time that they can measure
3. The error in a time measurement, even if everything is perfect, may be the size of this granularity (sometimes called a clock tick)
4. Always know what your clock granularity is
5. Ensure that your measurement is for a long enough duration (say 100 times the **tick**)

Problems with cold start

What happens when the code is executed? The assumption is that the code is ready to execute. But

1. Code may still be on disk, and not even read into memory.
2. Data may be in slow memory rather than fast (which may be wrong or right for what you are measuring)
3. Multiple tests often necessary to ensure that cold start effects are not present
4. Special effort often required to ensure data in the intended part of the memory hierarchy.

Problems with smart compilers

1. If the result of the computation is not used, the compiler may eliminate the code
2. Performance will look impossibly fantastic
3. Even worse, eliminate some of the code so the performance looks plausible
4. Ensure that the results are (or may be) used.

Problems with interference

1. Other activities are sharing your processor
    - Operating system, system demons, other users
        - Some parts of the hardware do not always perform with exactly the same performance
2. Make multiple tests and report
3. Easy choices include
    - Average tests represent what users might observe over time

Problems with measuring performance

1. Accurate, reproducible performance measurement is hard
2. Think carefully about your experiment:
    (a) What is it, precisely, that you want to measure
    (b) How representative is your test to the situation that you are trying to measure?

Summary and recommendations

Writing clean and clear code is an art and reflects your understanding of

1. derivation, verification, and implementation of algorithms
2. what can go wrong with algorithms
3. overview of important, known algorithms
4. how algorithms are used to solve mathematical problems
5. reproducible science and ethics
6. algorithmic thinking for gaining deeper insights about scientific problems

Computing is understanding and your understanding is reflected in your abilities to write clear and clean code.

Summary and recommendations

Some simple hints and tips in order to write clean and clear code

1. Spell out the algorithm and have a top-down approach to the flow of data
2. Start with coding as close as possible to eventual mathematical expressions
3. Use meaningful names for variables
4. Split tasks in simple functions and modules/classes
5. Functions should return as few as possible variables
6. Use unit tests and make sure your codes are producing the correct results
7. Where possible use symbolic coding to autogenerate code and check results
8. Make a proper timing of your algorithms
9. Use version control and make your science reproducible
10. Use IDEs or smart editors with debugging and analysis tools.
11. Automatize your computations interfacing high-level and compiled languages like C++ and Fortran.
12. .....

The final word?