Computational Physics and Quantum Mechanical Systems

Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo, Norway[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA[2]

May 16-20, 2016

Topics covered in this course

This course aims at giving you an overview of important numerical methods for solving quantum mechanical problems. In particular, in this course we will focus on

- Quantum Monte Carlo methods applied to few-electron systems like quantum dots, electrons confined to move in two or three dimensions. The system we will study will contain up to 20 interacting electrons in an oscillator trap.
- Linear algebra and eigenvalue problems with applications to the same physical systems, simple two-electron systems and Hartree-Fock theory.
- Object orientation, code optimization and High-performance computing aspects

The main programming language will be c++ (and Fortran), with Python examples in the slides.

Overarching aims of this course, how to develop and finalize a numerical project

While developing the mathematical tools and a numerical project we aim at giving you an insight in how to:

- Develop a critical approach to all steps in a project, which methods are most relevant, which natural laws and physical processes are important. Sort out initial conditions and boundary conditions etc.
- To teach you structured scientific computing and learn to structure a project using for example object orientation.
- To give you a critical understanding of central mathematical algorithms and methods from numerical analysis. In particular their limits and stability criteria.
- To show you how to use version control software to keep track of different versions of your normal project, how to verify and validate your codes and how to make your science reproducible. We will demonstrate this using git and github

Learning outcomes

We hope we can give you some useful hints and guidance on

- how to develop a thorough understanding of how computing is used to solve scientific problems
- getting to know some central algorithms used in science
- developing knowledge of high-performance computing elements: memory usage, vectorization and parallel algorithms
- understanding approximation errors and what can go wrong with algorithms
- getting experience with programming in a compiled language (Fortran, C, C++)
- getting experience with test frameworks and procedures
- being able to critically evaluate results and errors
- understanding how to increase the efficiency of numerical algorithms and pertinent software
- understanding tools to make science reproducible and to develop a sound ethical approach to scientific problems

Why is computing competence 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.

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 you 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.

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

Continuous verse 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.

Shift from classical mathematics to modern computing

1. The impact of the computer on mathematics is tremendous: science and industry now rely on solving mathematical problems through computing.
2. Computing increases the relevance in education by solving more realistic problems earlier.
3. Computing through programming is excellent training of creativity.
4. Computing enhances the understanding of abstractions and generalization.
5. Computing decreases the need for special tricks and tedious algebra, and shifts the focus to problem definition, visualization, and "what if" discussions.

The result is a deeper understanding of mathematical modeling. Not only is computing via programming a very powerful tool, it also a great pedagogical aid. For the mathematical training, there is one major new component among the arguments above: understanding abstractions and generalization. While many of the classical

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).

Modeling and computations

In this course we will try to focus on how to

- Introduce research elements as early as possible
- Trigger further insights in math and other disciplines
- Validation and verification of scientific results, with the possibility to emphasize ethical aspects as well. Version control is central.
- Introduce good project handling practices from day one.

Symbolic calculations and numerical calculations in one code

The following simple example here illustrates many of the previous points. 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 math import *
from sympy import *
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
```

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
```

Integrating numerical mathematics with calculus

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

- Validate and verify your algorithms.
- Include concepts like unit testing. It gives you the possibility to test and validate several or all parts of the code.
- Validation and verification are then included *naturally* and you can develop a better attitude to what is meant with an ethically sound scientific approach.
- The above example allows you to test the mathematical error of the algorithm for the trapezoidal rule by changing the number of integration points. You get to think error analysis from day one.

Additional benefits: A structured approach to solving problems

In this process we easily bake in

1. How to structure a code in terms of functions
2. How to make a module
3. How to read input data flexibly from the command line
4. How to create graphical/web user interfaces
5. How to write unit tests (test functions or doctests)
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)

Additional benefits: A structure approach to solving problems

The conventions and techniques outlined here 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.

Literature

- Lecture notes on Computational Physics by Morten Hjorth-Jensen, C++, Fortran, and Python
- A Primer on Scientific Programming with Python by Hans Petter Langtangen

Installing C++ and Fortran

If you wish to use C++ or Fortran, we recommend normally

- C++: Use an Integrated Development Environment (IDE) like Qt which works on all platforms
- C++: Use the IDE Visual C++ for Windows as operating system or Dev C++.
- For Mac (OSX) and linux users the GNU gcc and gfortran family of compilers are widely used.
- The Intel C++ and Fortran compilers are excellent. Free versions exist for students.

Installing and running python

- For Linux and OSX users python is included by default. Check which version by typing in a terminal window

```
python -v
```

Additional packages may have to be installed. For OSX users I highly recommend installing python and additional packages using brew. This makes it extremely simple to add packages and install additional software. With brew there is no point for OSX users to use virtual machines.

- You can also install platform independent IDE versions like the Enthought Python Distribution or Anaconda.
- For Windows users, you may consider a virtual machine like VMware Ubuntu
- You can also go directly to the python website and download the Windows version.

If you want to run a python program in terminal mode, simply write

```
python nameofprog.py
```

Optimization and profiling

If you use C++ or Fortran without an IDE, you would compile as

```
c++  -c  mycode.cpp
c++  -o  mycode.exe  mycode.o
```

For Fortran replace with for example **gfortran** or **ifort**. This is what we call a flat compiler option and should be used when we develop the code. It produces normally a very large and slow code when translated into machine instructions. We use this option for debugging and for establishing the correct program output because every operation is done precisely as the user specified it.
It is instructive to look up the compiler manual for further instructions by writing for example

```
man c++
```

More on optimization

We have additional compiler options for optimization. These may include procedure inlining where performance may be improved, moving constants inside loops outside the loop, identify potential parallelism, include automatic vectorization or replace a division with a reciprocal and a multiplication if this speeds up the code.

```
c++  -O3 -c  mycode.cpp
c++  -O3 -o  mycode.exe  mycode.o
```

This (other options are -O2 or -Ofast) is the recommended option.

Optimization and profiling

It is also useful to profile your program under the development stage. You would then compile with

```
c++  -pg -O3 -c  mycode.cpp
c++  -pg -O3 -o  mycode.exe  mycode.o
```

After you have run the code you can obtain the profiling information via

```
gprof mycode.exe >  ProfileOutput
```

When you have profiled properly your code, you must take out this option as it slows down performance. For memory tests use valgrind. An excellent environment for all these aspects, and much more, is Qt creator.

Optimization and debugging

Adding debugging options is a very useful alternative under the development stage of a program. You would then compile with

```
c++  -g -O0 -c  mycode.cpp
c++  -g -O0 -o  mycode.exe  mycode.o
```

This option generates debugging information allowing you to trace for example if an array is properly allocated. Some compilers work best with the no optimization option **-O0**.

Other optimization flags

Depending on the compiler, one can add flags which generate code that catches integer overflow errors. The flag **-ftrapv** does this for the CLANG compiler on OS X operating systems.

How do I run MPI on a PC/Laptop? MPI

To install MPI is rather easy on hardware running unix/linux as operating systems, follow simply the instructions from the OpenMPI website. See also subsequent slides. When you have made sure you have installed MPI on your PC/laptop,

- Compile with mpicxx/mpic++ or mpif90

```
# Compile and link
mpic++ -O3 -o nameofprog.x nameofprog.cpp
#  run code with for example 8 processes using mpirun/mpiexec
mpiexec -n 8 ./nameofprog.x
```

Can I do it on my own PC/laptop? OpenMP installation

If you wish to install MPI and OpenMP on your laptop/PC, we recommend the following:

- For OpenMP, the compile option **-fopenmp** is included automatically in recent versions of the C++ compiler and Fortran compilers. For users of different Linux distributions, siply use the available C++ or Fortran compilers and add the above compiler instructions, see also code examples below.
- For OS X users however, use for example

```
brew install clang-omp
```

Installing MPI

For linux/ubuntu users, you need to install two packages (alternatively use the synaptic package manager)

```
sudo apt-get install libopenmpi-dev
sudo apt-get install openmpi-bin
```

For OS X users, install brew (after having installed xcode and gcc, needed for the gfortran compiler of openmpi) and then install with brew

```
brew install openmpi
```

When running an executable (code.x), run as

```
mpirun -n 10 ./code.x
```

where we indicate that we want the number of processes to be 10.

Installing MPI and using Qt

With openmpi installed, when using Qt, add to your .pro file the instructions here
You may need to tell Qt where openmpi is stored.

Matrix Handling in C/C++ and Fortran, Static and Dynamical allocation

Static

We have an $N \times N$ matrix A with $N = 100$ In C/C++ this would be defined as

```
int N = 100;
double A[100][100];
//   initialize all elements to zero
for(i=0 ; i < N ; i++) {
   for(j=0 ; j < N ; j++) {
      A[i][j] = 0.0;
```

Note the way the matrix is organized, row-major order.

Matrix Handling in C/C++

Row Major Order, Addition

We have $N \times N$ matrices A, B and C and we wish to evaluate $A = B + C$.

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij},$$

In C/C++ this would be coded like

```
for(i=0 ; i < N ; i++) {
   for(j=0 ; j < N ; j++) {
      a[i][j] = b[i][j]+c[i][j]
```

Matrix Handling in C/C++

Row Major Order, Multiplication

We have $N \times N$ matrices A, B and C and we wish to evaluate $A = BC$.

$$\mathbf{A} = \mathbf{BC} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj},$$

In C/C++ this would be coded like

```
for(i=0 ; i < N ; i++) {
   for(j=0 ; j < N ; j++) {
      for(k=0 ; k < N ; k++) {
         a[i][j]+=b[i][k]*c[k][j];
```

Matrix Handling in Fortran 90/95, dynamic allocation and deallocation

Column Major Order

```
ALLOCATE (a(N,N), b(N,N), c(N,N))
DO j=1,  N
   DO i=1, N
      a(i,j)=b(i,j)+c(i,j)
   ENDDO
ENDDO
...
DEALLOCATE(a,b,c)
```

Fortran 90 writes the above statements in a much simpler way

```
a=b+c
```

Multiplication

```
a=MATMUL(b,c)
```

Fortran contains also the intrinsic functions TRANSPOSE and CONJUGATE.

Dynamic memory allocation in C/C++

At least three possibilities in this course

- Do it yourself with plain C/C++ functions like **new** and **delete**.
- Use the functions provided in the library package lib.cpp
- Use Armadillo `http://arma.sourceforgenet` (a C++ linear algebra library, discussion next two weeks, both here and at lab).
- Write your own vector-matrix class or use the examples in the program folder

Matrix Handling in C/C++, Dynamic Allocation

Do it yourself

```
int N;
double **  A;
A = new double*[N]
for ( i = 0; i < N; i++)
    A[i] = new double[N];
```

Always free space when you don't need an array anymore.

```
for ( i = 0; i < N; i++)
    delete[] A[i];
delete[] A;
```

Armadillo, recommended!!

- Armadillo is a C++ linear algebra library (matrix maths) aiming towards a good balance between speed and ease of use. The syntax is deliberately similar to Matlab.
- Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries).
- A delayed evaluation approach is employed (at compile-time) to combine several operations into one and reduce (or eliminate) the need for temporaries. This is accomplished through recursive templates and template meta-programming.
- Useful for conversion of research code into production environments, or if C++ has been decided as the language of choice, due to speed and/or integration capabilities.
- The library is open-source software, and is distributed under a license that is useful in both open-source and

Armadillo, simple examples

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv)
  {
  mat A = randu<mat>(5,5);
  mat B = randu<mat>(5,5);

  cout << A*B << endl;

  return 0;
```

Armadillo, how to compile and install

For people using Ubuntu, Debian, Linux Mint, simply go to the synaptic package manager and install armadillo from there. You may have to install Lapack as well. For Mac and Windows users, follow the instructions from the webpage http://arma.sourceforge.net. For Mac users we strongly recommend using brew.

To compile, use for example

```
c++ -O2 -o program.x program.cpp  -larmadillo -llapack -lblas
```

where the -l option indicates the library you wish to link to.

Armadillo, simple examples

```
#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
  {
  // directly specify the matrix size (elements are uninitialised)
  mat A(2,3);
  // .n_rows = number of rows    (read only)
  // .n_cols = number of columns (read only)
  cout << "A.n_rows = " << A.n_rows << endl;
  cout << "A.n_cols = " << A.n_cols << endl;
  // directly access an element (indexing starts at 0)
  A(1,2) = 456.0;
  A.print("A:");
  // scalars are treated as a 1x1 matrix,
  // hence the code below will set A to have a size of 1x1
  A = 5.0;
  A.print("A:");
  // if you want a matrix with all elements set to a particular value
  // the .fill() member function can be used
  A.set_size(3,3);
  A.fill(5.0);  A.print("A:");
```

Armadillo, simple examples

```
mat B;

// endr indicates "end of row"
B << 0.555950 << 0.274690 << 0.540605 << 0.798938 << endr
  << 0.108929 << 0.830123 << 0.891726 << 0.895283 << endr
  << 0.948014 << 0.973234 << 0.216504 << 0.883152 << endr
  << 0.023787 << 0.675382 << 0.231751 << 0.450332 << endr;

// print to the cout stream
// with an optional string before the contents of the matrix
B.print("B:");

// the << operator can also be used to print the matrix
// to an arbitrary stream (cout in this case)
cout << "B:" << endl << B << endl;
// save to disk
B.save("B.txt", raw_ascii);
// load from disk
mat C;
C.load("B.txt");
C += 2.0 * B;
C.print("C:");
```

Armadillo, simple examples

```
// submatrix types:
//
// .submat(first_row, first_column, last_row, last_column)
// .row(row_number)
// .col(column_number)
// .cols(first_column, last_column)
// .rows(first_row, last_row)

cout << "C.submat(0,0,3,1) =" << endl;
cout << C.submat(0,0,3,1) << endl;

// generate the identity matrix
mat D = eye<mat>(4,4);

D.submat(0,0,3,1) = C.cols(1,2);
D.print("D:");

// transpose
cout << "trans(B) =" << endl;
cout << trans(B) << endl;

// maximum from each column (traverse along rows)
cout << "max(B) =" << endl;
cout << max(B) << endl;
```

Armadillo, simple examples

```
// maximum from each row (traverse along columns)
cout << "max(B,1) =" << endl;
cout << max(B,1) << endl;
// maximum value in B
cout << "max(max(B)) = " << max(max(B)) << endl;
// sum of each column (traverse along rows)
cout << "sum(B) =" << endl;
cout << sum(B) << endl;
// sum of each row (traverse along columns)
cout << "sum(B,1) =" << endl;
cout << sum(B,1) << endl;
// sum of all elements
cout << "sum(sum(B)) = " << sum(sum(B)) << endl;
cout << "accu(B)     = " << accu(B) << endl;
// trace = sum along diagonal
cout << "trace(B)    = " << trace(B) << endl;
// random matrix -- values are uniformly distributed in the [0,1] in
mat E = randu<mat>(4,4);
E.print("E:");
```

Armadillo, simple examples

```
// row vectors are treated like a matrix with one row
rowvec r;
r << 0.59499 << 0.88807 << 0.88532 << 0.19968;
r.print("r:");

// column vectors are treated like a matrix with one column
colvec q;
q << 0.81114 << 0.06256 << 0.95989 << 0.73628;
q.print("q:");

// dot or inner product
cout << "as_scalar(r*q) = " << as_scalar(r*q) << endl;

  // outer product
cout << "q*r =" << endl;
cout << q*r << endl;


// sum of three matrices (no temporary matrices are created)
mat F = B + C + D;
F.print("F:");

  return 0;
```

Armadillo, simple examples

```
#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
  {
  cout << "Armadillo version: " << arma_version::as_string() << endl;

  mat A;

  A << 0.165300 << 0.454037 << 0.995795 << 0.124098 << 0.047084 << end
    << 0.688782 << 0.036549 << 0.552848 << 0.937664 << 0.866401 << end
    << 0.348740 << 0.479388 << 0.506228 << 0.145673 << 0.491547 << end
    << 0.148678 << 0.682258 << 0.571154 << 0.874724 << 0.444632 << end
    << 0.245726 << 0.595218 << 0.409327 << 0.367827 << 0.385736 << end

  A.print("A =");

  // determinant
  cout << "det(A) = " << det(A) << endl;
```

Armadillo, simple examples

```
// inverse
cout << "inv(A) = " << endl << inv(A) << endl;
double k = 1.23;

mat    B = randu<mat>(5,5);
mat    C = randu<mat>(5,5);

rowvec r = randu<rowvec>(5);
colvec q = randu<colvec>(5);


// examples of some expressions
// for which optimised implementations exist
// optimised implementation of a trinary expression
// that results in a scalar
cout << "as_scalar( r*inv(diagmat(B))*q ) = ";
cout << as_scalar( r*inv(diagmat(B))*q ) << endl;

// example of an expression which is optimised
// as a call to the dgemm() function in BLAS:
cout << "k*trans(B)*C = " << endl << k*trans(B)*C;

  return 0;
```