Slides from FYS3150 Lectures

Morten Hjorth-Jensen

Department of Physics and Center of Mathematics for Applications
University of Oslo, N-0316 Oslo, Norway

Fall 2011

Week 34

- Monday: First lecture: Presentation of the course, aims and content
- Monday: Second Lecture: Introduction to C++ programming and numerical precision.
- Wednesday: Numerical precision and C++ programming, continued
- Numerical differentiation and loss of numerical precision (chapter 3 lecture notes)
- Computer-Lab: Tuesday, Wednesday and Thursday. First time: Thursday this week, Presentation of hardware and software at room FV329. Exercises 1 and 2.

Lectures and ComputerLab

- Lectures: Monday (12.15pm-2pm) and Wednesday (12.15pm-2pm)
- Detailed lecture notes, exercises, all programs presented, projects etc can be found at the homepage of the course.
- Computerlab: Tuesday (12pm-6pm), Wednesday (9am-12am), and Thursday (9am-7pm) room FV329.
- Weekly plans and all other information are on the official webpage.

Course Format

- Several computer exercises, 5 compulsory projects. Electronic reports only.
- Evaluation and grading: The last project (50% of final grade) and a final written exam (50% of final grade). Dates to be settled, most likely start Monday December 12, but I need your constraints.
- The computer lab (room FV329)consists of 16 Linux PCs. C/C++ is the default programming language, but Fortran95 and Python are also used. All source codes discussed during the lectures can be found at the webpage of the course. We recommend either C/C++, Fortran95 or Python as languages.

ComputerLab

day	teacher
Tuesday 12pm-6pm	NN
Wednesday 9am-12pm	NN
Thursday 9am-1 pm	NN
Thursday 1pm-5pm	NN

Set up your preferred lab time today, see separate list.

Topics covered in this course

- Numerical precision and intro to C++ programming
- Numerical derivation and integration
- Random numbers and Monte Carlo integration
- Monte Carlo methods in statistical physics
- Quantum Monte Carlo methods
- Linear algebra and eigenvalue problems
- Non-linear equations and roots of polynomials
- Ordinary differential equations
- Partial differential equations
- Parallelization of codes
- Programming av GPUs (optional)

Syllabus FYS3150

Linear algebra and eigenvalue problems, chapters 6 and 7

- Know Gaussian elimination and LU decomposition
- How to solve linear equations
- How to obtain the inverse and the determinant of a real symmetric matrix
- Cholesky and tridiagonal matrix decomposition

Syllabus FYS3150

Linear algebra and eigenvalue problems, chapters 6 and 7

- Householder's tridiagonalization technique and finding eigenvalues based on this
- Jacobi's method for finding eigenvalues
- Singular value decomposition
- Qubic Spline interpolation

Syllabus FYS3150

Numerical integration, standard methods and Monte Carlo methods (chapters 4 and 11)

- Trapezoidal, rectangle and Simpson's rules
- Gaussian quadrature, emphasis on Legendre polynomials, but you need to know about other polynomials as well.
- Brute force Monte Carlo integration
- Random numbers (simplest algo, ran0) and probability distribution functions, expectation values
- Improved Monte Carlo integration and importance sampling.

Syllabus FYS3150

Monte Carlo methods in physics (chapters 12, 13, and 14)

- Random walks and Markov chains and relation with diffusion equation
- Metropolis algorithm, detailed balance and ergodicity
- Simple spin systems and phase transitions
- Variational Monte Carlo
- How to construct trial wave functions for quantum systems

Syllabus FYS3150

Ordinary differential equations (chapters 8 and 9)

- Euler's method and improved Euler's method, truncation errors
- Runge Kutta methods, 2nd and 4th order, truncation errors
- How to implement a second-order differential equation, both linear and non-linear. How to make your equations dimensionless.
- Boundary value problems, shooting and matching method (chap 9).

Syllabus FYS3150

Partial differential equations, chapter 10

- Set up diffusion, Poisson and wave equations up to 2 spatial dimensions and time
- Set up the mathematical model and algorithms for these equations, with boundary and initial conditions. Their stability conditions.
- Explicit, implicit and Crank-Nicolson schemes, and how to solve them. Remember that they result in triangular matrices.
- How to compute the Laplacian in Poisson's equation.
- How to solve the wave equation in one and two dimensions.

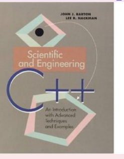
Overarching aims of this course

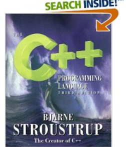
- Develop a critical approach to all steps in a project, which methods are most relevant, which natural laws and physical processes are important. Sort out initial conditions and boundary conditions etc.
- This means to teach you structured scientific computing, learn to structure a project.
- A critical understanding of central mathematical algorithms and methods from numerical analysis. In particular their limits and stability criteria.
- Always try to find good checks of your codes (like solutions on closed form)
- To enable you to develop a critical view on the mathematical model and the physics.

And, there is nothing like a code which gives correct results!!

Selected Texts and lectures on C/C++

- J. J. Barton and L. R. Nackman,*Scientific and Engineering C++*, Addison Wesley, 3rd edition 2000.
- B. Stoustrup, *The C++ programming language*, Pearson, 1997.
- H. P. Langtangen INF-VERK3830 `http://heim.ifi.uio.no/~hpl/INF-VERK4830/`
- D. Yang, *C++ and Object-oriented Numeric Computing for Scientists and Engineers*, Springer 2000.
- More books reviewed at `http:://www.accu.org/` and `http://www.comeaucomputing.com/booklist/`





Other courses in Computational Science at UiO

Bachelor/Master/PhD Courses

- INF-MAT4350 Numerical linear algebra
- MAT-INF3300/3310, PDEs and Sobolev spaces I and II
- INF-MAT3360 PDEs
- INF5620/5630 Numerical methods for PDEs, finite element method
- FYS4411 Computational physics II (Parallelization (MPI), object orientation, quantum mechanical systems with many interacting particles), spring semester
- FYS4460 Computational physics III (Parallelization (MPI), object orientation, classical statistical physics, simulation of phase transitions, spring semester
- INF3331 Problem solving with high-level languages (Python), fall semester
- INF3380 Parallel computing for problems in the Natural Sciences (mostly PDEs), spring semester

A structured programming approach

- Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.
- Always try to choose the simplest algorithm. Computational speed can be improved upon later.
- Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debuging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!

A structured programming approach

- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.
- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice or more.

Getting Started

Compiling and linking

In order to obtain an executable file for a C++ program, the following instructions under Linux/Unix can be used

```
c++ -c -Wall myprogram.cpp
c++ -o myprogram myprogram.o
```

where the compiler is called through the command c++/g++. The compiler option -Wall means that a warning is issued in case of non-standard language. The executable file is in this case *myprogram*. The option $-c$ is for compilation only, where the program is translated into machine code, while the $-o$ option links the produced object file *myprogram.o* and produces the executable *myprogram* .
For Fortran95 we use the Intel compiler, replace c++ with ifort. Also, to speed up the code use compile options like

```
c++ -O3 -c -Wall myprogram.cpp
```

Makefiles and simple scripts

Under Linux/Unix it is often convenient to create a so-called makefile, which is a script which includes possible compiling commands.

```
# Comment lines
# General makefile for c - choose PROG =   name of given program
# Here we define compiler option, libraries and the  target
CC= g++ -Wall
PROG= myprogram
# this is the math library in C, not necessary for C++
LIB = -lm
# Here we make the executable file
${PROG} :          ${PROG}.o
                   ${CC} ${PROG}.o ${LIB} -o ${PROG}
# whereas here we create the object file
${PROG}.o :       ${PROG}.c
                  ${CC} -c ${PROG}.c
```

If you name your file for 'makefile', simply type the command **make** and Linux/Unix executes all of the statements in the above makefile. Note that C++ files have the extension .cpp

Hello world

The C encounter

Here we present first the C version.

```
/* comments in C begin like this and end with */
#include <stdlib.h> /* atof function */
#include <math.h>   /* sine function */
#include <stdio.h>  /* printf function */
int main (int argc, char* argv[])
{
  double r, s;        /* declare variables */
  r = atof(argv[1]);  /* convert the text argv[1] to double */
  s = sin(r);
  printf("Hello, World! sin(%g)=%g\n", r, s);
  return 0;           /* success execution of the program */
}
```

Hello World

Dissection I

The compiler must see a declaration of a function before you can call it (the compiler checks the argument and return types). The declaration of library functions appears in so-called “header files” that must be included in the program, e.g.,

```
#include <stdlib.h> /* atof function */
```

We call three functions (atof, sin, printf) and these are declared in three different header files. The main program is a function called main with a return value set to an integer, int (0 if success). The operating system stores the return value, and other programs/utilities can check whether the execution was successful or not. The command-line arguments are transferred to the main function through

```
int main (int argc, char* argv[])
```

Hello World

Dissection II

The command-line arguments are transferred to the main function through

```
int main (int argc, char* argv[])
```

The integer *argc* is the no of command-line arguments, set to one in our case, while *argv* is a vector of strings containing the command-line arguments with *argv*[0] containing the name of the program and *argv*[1], *argv*[2], ... are the command-line args, i.e., the number of lines of input to the program. Here we define floating points, see also below, through the keywords *float* for single precision real numbers and *double* for double precision. The function *atof* transforms a text (*argv*[1]) to a float. The sine function is declared in math.h, a library which is not automatically included and needs to be linked when computing an executable file.

With the command *printf* we obtain a formatted printout. The *printf* syntax is used for formatting output in many C-inspired languages (Perl, Python, awk, partly C++).

Hello World

Now in C++

Here we present first the C++ version.

```
// A comment line begins like this in C++ programs
// Standard ANSI-C++ include files
using namespace std
#include <iostream>  // input and output
int main (int argc, char* argv[])
{
//  convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << '\n';
// success
  return 0;
}
```

C++ Hello World

Dissection I

We have replaced the call to *printf* with the standard C++ function *cout*. The header file $<$ *iostream.h* $>$ is then needed. In addition, we don't need to declare variables like *r* and *s* at the beginning of the program. I personally prefer however to declare all variables at the beginning of a function, as this gives **me** a feeling of greater readability.

Brief summary

C/C++ program

- A C/C++ program begins with include statements of header files (libraries,intrinsic functions etc)
- Functions which are used are normally defined at top (details next week)
- The main program is set up as an integer, it returns 0 (everything correct) or 1 (something went wrong)
- Standard **if**, **while** and **for** statements as in Java, Fortran, Python...
- Integers have a very limited range.

Brief summary

Arrays

- A C/C++ array begins by indexing at 0!
- Array allocations are done by size, not by the final index value.If you allocate an array with 10 elements, you should index them from $0, 1, \ldots, 9$.
- Initialize always an array before a computation.

Serious problems and representation of numbers

Integer and Real Numbers

- **Overflow**
- **Underflow**
- **Roundoff errors**
- **Loss of precision**

Limits, you must declare variables

C++ and Fortran declarations

type in C/C++ and Fortran 90/95	bits	range
int/INTEGER (2)	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
int/long int/INTEGER(4)	32	-2147483648 to 2147483647
signed long int	32	-2147483648 to 2147483647
float/REAL(4)	32	3.4×10^{-44} to $3.4 \times 10^{+38}$
double/REAL(8)	64	1.7×10^{-322} to $1.7 \times 10^{+308}$
long double	64	1.7×10^{-322} to $1.7 \times 10^{+308}$

From decimal to binary representation

How to do it

$$a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \cdots + a_0 2^0.$$

In binary notation we have thus $(417)_{10} = (110110001)_2$ since we have

$$(110100001)_2 = 1\times2^8+1\times2^7+0\times2^6+1\times2^5+0\times2^4+0\times2^3+0\times2^2+0\times2^2+0\times2^1+1\times2^0.$$

To see this, we have performed the following divisions by 2

417/2=208	remainder 1	coefficient of 2^0 is 1
208/2=104	remainder 0	coefficient of 2^1 is 0
104/2=52	remainder 0	coefficient of 2^2 is 0
52/2=26	remainder 0	coefficient of 2^3 is 0
26/2=13	remainder 1	coefficient of 2^4 is 0
13/2= 6	remainder 1	coefficient of 2^5 is 1
6/2= 3	remainder 0	coefficient of 2^6 is 0
3/2= 1	remainder 1	coefficient of 2^7 is 1
1/2= 0	remainder 1	coefficient of 2^8 is 1

From decimal to binary representation

Integer numbers

```
using namespace std;
#include <iostream>
int main (int argc, char* argv[])
{
  int i;
  int terms[32]; // storage of a0, a1, etc, up to 32 bits
  int number = atoi(argv[1]);
  // initialise the term a0, a1 etc
  for (i=0; i < 32 ; i++){ terms[i] = 0;}
  for (i=0; i < 32 ; i++){
    terms[i] = number%2;
    number /= 2;
  }
  // write out results
  cout << "Number of bytes used= " << sizeof(number) << endl;
  for (i=0; i < 32 ; i++){
    cout << " Term nr: " << i << "Value= " << terms[i];
    cout << endl;
  }
  return 0;
}
```

From decimal to binary representation

Integer numbers, Fortran

```
PROGRAM binary_integer
IMPLICIT NONE
  INTEGER  i, number, terms(0:31) ! storage of a0, a1, etc, up to 32 b

  WRITE(*,*) 'Give a number to transform to binary notation'
  READ(*,*) number
! Initialise the terms a0, a1 etc
  terms = 0
! Fortran takes only integer loop variables
  DO i=0, 31
     terms(i) = MOD(number,2)
     number = number/2
  ENDDO
! write out results
  WRITE(*,*) 'Binary representation '
  DO i=0, 31
    WRITE(*,*)' Term nr and value', i, terms(i)
  ENDDO

END PROGRAM binary_integer
```

Integer Numbers

Possible Overflow for Integers

```
// A comment line begins like this in C++ programs
// Program to calculate 2**n
// Standard ANSI-C++ include files */
using namespace std
#include <iostream>
#include <cmath>
int main()
{
   int  int1, int2, int3;
// print to screen
   cout << "Read in the exponential N for 2^N =\n";
// read from screen
   cin >> int2;
   int1 = (int) pow(2., (double) int2);
   cout << " 2^N * 2^N = " << int1*int1 << "\n";
   int3 = int1 - 1;
   cout << " 2^N*(2^N - 1) = " << int1 * int3  << "\n";
   cout << " 2^N- 1 = " << int3  << "\n";
   return 0;
}
// End: program main()
```

Loss of Precision

Machine Numbers

In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation.

$$9.90625 = 0.990625 \times 10^1,$$

and a real non-zero number could be generalized as

$$x = \pm r \times 10^{\mathrm{n}}, \tag{1}$$

with r a number in the range $1/10 \le r < 1$. In a similar way we can use represent a binary number in scientific notation as

$$x = \pm q \times 2^{\mathrm{m}}, \tag{2}$$

with q a number in the range $1/2 \le q < 1$. This means that the mantissa of a binary number would be represented by the general formula

$$(0.a_{-1}a_{-2}\dots a_{-n})_2 = a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \dots + a_{-n} \times 2^{-n}. \tag{3}$$

Loss of Precision

Machine Numbers

In a typical computer, floating-point numbers are represented in the way described above, but with certain restrictions on q and m imposed by the available word length. In the machine, our number x is represented as

$$x = (-1)^s \times \mathrm{mantissa} \times 2^{\mathrm{exponent}}, \tag{4}$$

where s is the sign bit, and the exponent gives the available range. With a single-precision word, 32 bits, 8 bits would typically be reserved for the exponent, 1 bit for the sign and 23 for the mantissa.

Loss of Precision

Machine Numbers

A modification of the scientific notation for binary numbers is to require that the leading binary digit 1 appears to the left of the binary point. In this case the representation of the mantissa q would be $(1.f)_2$ and $1 \leq q < 2$. This form is rather useful when storing binary numbers in a computer word, since we can always assume that the leading bit 1 is there. One bit of space can then be saved meaning that a 23 bits mantissa has actually 24 bits. This means explicitely that a binary number with 23 bits for the mantissa reads

$$(1.a_{-1}a_{-2}\dots a_{-23})_2 = 1\times 2^0 + a_{-1}\times 2^{-1} + +a_{-2}\times 2^{-2} + \dots + a_{-23}\times 2^{-23}. \quad (5)$$

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

where the first bit is reserved for the sign, 1 in this case yielding a negative sign. The exponent m is given by the next 8 binary numbers 01111101 resulting in 125 in the decimal system.

Loss of Precision

Machine Numbers

However, since the exponent has eight bits, this means it has $2^8 - 1 = 255$ possible numbers in the interval $-128 \leq m \leq 127$, our final exponent is $125 - 127 = -2$ resulting in 2^{-2}. Inserting the sign and the mantissa yields the final number in the decimal representation as

$$-2^{-2}\left(1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}\right) =$$

$$(-0.4765625)_{10}.$$

In this case we have an exact machine representation with 32 bits (actually, we need less than 23 bits for the mantissa).

If our number x can be exactly represented in the machine, we call x a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a computation, an inevitable error will arise in representing it as accurately as possible by a machine number.

Loss of Precision

Machine Numbers

A floating number x, labelled $fl(x)$ will therefore always be represented as

$$fl(x) = x(1 \pm \epsilon_x), \tag{6}$$

with x the exact number and the error $|\epsilon_x| \leq |\epsilon_M|$, where ϵ_M is the precision assigned. A number like $1/10$ has no exact binary representation with single or double precision. Since the mantissa

$$(1.a_{-1}a_{-2}\dots a_{-n})_2$$

is always truncated at some stage n due to its limited number of bits, there is only a limited number of real binary numbers. The spacing between every real binary number is given by the chosen machine precision. For a 32 bit words this number is approximately $\epsilon_M \sim 10^{-7}$ and for double precision (64 bits) we have $\epsilon_M \sim 10^{-16}$, or in terms of a binary base as 2^{-23} and 2^{-52} for single and double precision, respectively.

Loss of Precision

Machine Numbers

In the machine a number is represented as

$$fl(x) = x(1 + \epsilon) \tag{7}$$

where $|\epsilon| \leq \epsilon_M$ and ϵ is given by the specified precision, 10^{-7} for single and 10^{-16} for double precision, respectively. ϵ_M is the given precision. In case of a subtraction $a = b - c$, we have

$$fl(a) = fl(b) - fl(c) = a(1 + \epsilon_a), \tag{8}$$

or

$$fl(a) = b(1 + \epsilon_b) - c(1 + \epsilon_c), \tag{9}$$

meaning that

$$fl(a)/a = 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a}, \tag{10}$$

and if $b \approx c$ we see that there is a potential for an increased error in $fl(a)$.

Loss of Precision

Machine Numbers

Define the absolute error as

$$|fl(a) - a|, \tag{11}$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a. \tag{12}$$

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - fl(c) - (b - c)|}{a}, \tag{13}$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}. \tag{14}$$

The relative error is the quantity of interest in scientific work. Information about the absolute error is normally of little use in the absence of the magnitude of the quantity being measured.

Loss of numerical precision

Suppose we wish to evaluate the function

$$f(x) = \frac{1 - \cos(x)}{\sin(x)},$$

for small values of x. Five leading digits. If we multiply the denominator and numerator with $1 + \cos(x)$ we obtain the equivalent expression

$$f(x) = \frac{\sin(x)}{1 + \cos(x)}.$$

If we now choose $x = 0.007$ (in radians) our choice of precision results in

$$\sin(0.007) \approx 0.69999 \times 10^{-2},$$

and

$$\cos(0.007) \approx 0.99998.$$

Loss of numerical precision

The first expression for $f(x)$ results in

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2},$$

while the second expression results in

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2},$$

which is also the exact result. In the first expression, due to our choice of precision, we have only one relevant digit in the numerator, after the subtraction. This leads to a loss of precision and a wrong result due to a cancellation of two nearly equal numbers. If we had chosen a precision of six leading digits, both expressions yield the same answer.

Loss of numerical precision

If we were to evaluate $x \sim \pi$, then the second expression for $f(x)$ can lead to potential losses of precision due to cancellations of nearly equal numbers.

This simple example demonstrates the loss of numerical precision due to roundoff errors, where the number of leading digits is lost in a subtraction of two near equal numbers. The lesson to be drawn is that we cannot blindly compute a function. We will always need to carefully analyze our algorithm in the search for potential pitfalls. There is no magic recipe however, the only guideline is an understanding of the fact that a machine cannot represent correctly **all** numbers.

Loss of Precision, bad thing

Real Numbers

- **Overflow :** When the positive exponent exceeds the max value, e.g., 308 for DOUBLE PRECISION (64 bits). Under such circumstances the program will terminate and some compilers may give you the warning 'OVERFLOW'.
- **Underflow :** When the negative exponent becomes smaller than the min value, e.g., -308 for DOUBLE PRECISION. Normally, the variable is then set to zero and the program continues. Other compilers (or compiler options) may warn you with the 'UNDERFLOW' message and the program terminates.

Loss of precision, real numbers

- **Roundoff errors** A floating point number like

$$x = 1.234567891112131468 = 0.1234567891112131468 \times 10^1 \tag{15}$$

may be stored in the following way. The exponent is small and is stored in full precision. However, the mantissa is not stored fully. In double precision (64 bits), digits beyond the 15th are lost since the mantissa is normally stored in two words, one which is the most significant one representing 123456 and the least significant one containing 789111213. The digits beyond 3 are lost. Clearly, if we are summing alternating series with large numbers, subtractions between two large numbers may lead to roundoff errors, since not all relevant digits are kept. This leads eventually to the next problem, namely

More on Loss of Precision

Real Numbers

- **Loss of precision** When one has to e.g., multiply two large numbers where one suspects that the outcome may be beyond the bonds imposed by the variable declaration, one could represent the numbers by logarithms, or rewrite the equations to be solved in terms of dimensionless variables. When dealing with problems in e.g., particle physics or nuclear physics where distance is measured in fm (10^{-15}m), it can be quite convenient to redefine the variables for distance in terms of a dimensionless variable of the order of unity. To give an example, suppose you work with single precision and wish to perform the addition $1 + 10^{-8}$. In this case, the information containing in 10^{-8} is simply lost in the addition. Typically, when performing the addition, the computer equates first the exponents of the two numbers to be added. For 10^{-8} this has however catastrophic consequences since in order to obtain an exponent equal to 10^{0}, bits in the mantissa are shifted to the right. At the end, all bits in the mantissa are zeros.

A problematic Case

Three ways of computing e^{-x}

1. Brute force

$$\exp{(-x)} = \sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!}$$

2. recursion relation for

$$\exp{(-x)} = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!}$$

$$s_n = -s_{n-1}\frac{x}{n},$$

3.

$$\exp{(x)} = \sum_{n=0}^{\infty} s_n$$

$$\exp{(-x)} = \frac{1}{\exp{(x)}}$$

Program to compute $\exp(-x)$

Brute Force

```
// Program to calculate function exp(-x)
// using straightforward summation with differing  precision
using namespace std
#include <iostream>
#include <cmath>
// type float:  32 bits precision
// type double: 64 bits precision
#define   TYPE          double
#define   PHASE(a)      (1 - 2 * (abs(a) % 2))
#define   TRUNCATION    1.0E-10
// function declaration
TYPE factorial(int);
```

Program to compute $\exp(-x)$

Still Brute Force

```
int main()
{
   int   n;
   TYPE  x, term, sum;
   for(x = 0.0; x < 100.0; x += 10.0)  {
     sum  = 0.0;                //initialization
     n    = 0;
     term = 1;
     while(fabs(term) > TRUNCATION)  {
         term =  PHASE(n) * (TYPE) pow((TYPE) x,(TYPE) n)
                / factorial(n);
         sum += term;
         n++;
     }  // end of while() loop
```

Program to compute $\exp(-x)$

Oh, it never ends!

```
      printf("\nx = %4.1f   exp = %12.5E  series = %12.5E
              number of terms = %d",
              x, exp(-x), sum, n);
   } // end of for() loop

   printf("\n");           // a final line shift on output
   return 0;
} // End: function main()
//     The function factorial()
//     calculates and returns n!
TYPE factorial(int n)
{
   int  loop;
   TYPE fac;
   for(loop = 1, fac = 1.0; loop <= n; loop++)  {
      fac *= loop;
   }
   return fac;
} // End: function factorial()
```

Results $\exp(-x)$

What is going on?

x	$\exp(-x)$	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.453999E-04	44
20.0	0.206115E-08	0.487460E-08	72
30.0	0.935762E-13	-0.342134E-04	100
40.0	0.424835E-17	-0.221033E+01	127
50.0	0.192875E-21	-0.833851E+05	155
60.0	0.875651E-26	-0.850381E+09	171
70.0	0.397545E-30	NaN	171
80.0	0.180485E-34	NaN	171
90.0	0.819401E-39	NaN	171
100.0	0.372008E-43	NaN	171

Program to compute $\exp(-x)$

```
// program to compute exp(-x) without exponentials
using namespace std
#include <iostream>
#include <cmath>
#define  TRUNCATION     1.0E-10

int main()
{
   int       loop, n;
   double    x, term, sum;
   for(loop = 0; loop <= 100; loop += 10)
   {
     x    = (double) loop;          // initialization
     sum  = 1.0;
     term = 1;
     n    = 1;
```

Program to compute $\exp(-x)$

Last statements

```
     while(fabs(term) > TRUNCATION)
       {
 term *= -x/((double) n);
 sum  += term;
 n++;
       } // end while loop
     cout << "x = " << x   << " exp = " << exp(-x) <<"series = "
          << sum  << " number of terms =" << n << "\n";
   } // end of for() loop

   cout << "\n";           // a final line shift on output

}  /*    End: function main() */
```

Results $\exp(-x)$

More Problems

x	$\exp(-x)$	Series	Number of terms in series
0.000000	0.10000000E+01	0.10000000E+01	1
10.000000	0.45399900E-04	0.45399900E-04	44
20.000000	0.20611536E-08	0.56385075E-08	72
30.000000	0.93576230E-13	-0.30668111E-04	100
40.000000	0.42483543E-17	-0.31657319E+01	127
50.000000	0.19287498E-21	0.11072933E+05	155
60.000000	0.87565108E-26	-0.33516811E+09	182
70.000000	0.39754497E-30	-0.32979605E+14	209
80.000000	0.18048514E-34	0.91805682E+17	237
90.000000	0.81940126E-39	-0.50516254E+22	264
100.000000	0.37200760E-43	-0.29137556E+26	291

Most used formula for derivatives

3 point formulae

First derivative ($f_0 = f(x_0)$, $f_{-h} = f(x_0 - h)$ and $f_{+h} = f(x_0 + h)$)

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}.$$

Second derivative

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}.$$

Error Analysis

$$\epsilon = log_{10}\left(\left|\frac{f''_{\rm computed} - f''_{\rm exact}}{f''_{\rm exact}}\right|\right),$$

$$\epsilon_{\rm tot} = \epsilon_{\rm approx} + \epsilon_{\rm ro}.$$

For the computed second derivative we have

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} - 2\sum_{j=1}^{\infty}\frac{f_0^{(2j+2)}}{(2j+2)!}h^{2j},$$

and the truncation or approximation error goes like

$$\epsilon_{\rm approx} \approx \frac{f_0^{(4)}}{12}h^2.$$

Error Analysis

If we were not to worry about loss of precision, we could in principle make h as small as possible. However, due to the computed expression in the above program example

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} = \frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2},$$

we reach fairly quickly a limit for where loss of precision due to the subtraction of two nearly equal numbers becomes crucial.
If $(f_{\pm h} - f_0)$ are very close, we have $(f_{\pm h} - f_0) \approx \epsilon_M$, where $|\epsilon_M| \leq 10^{-7}$ for single and $|\epsilon_M| \leq 10^{-15}$ for double precision, respectively.
We have then

$$|f_0''| = \left|\frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2}\right| \leq \frac{2\epsilon_M}{h^2}.$$

Error Analysis

Our total error becomes

$$|\epsilon_{\text{tot}}| \leq \frac{2\epsilon_M}{h^2} + \frac{f_0^{(4)}}{12} h^2.$$

It is then natural to ask which value of h yields the smallest total error. Taking the derivative of $|\epsilon_{\text{tot}}|$ with respect to h results in

$$h = \left(\frac{24\epsilon_M}{f_0^{(4)}} \right)^{1/4}.$$

With double precision and $x = 10$ we obtain

$$h \approx 10^{-4}.$$

Beyond this value, it is essentially the loss of numerical precision which takes over.

Error Analysis

Due to the subtractive cancellation in the expression for f'' there is a pronounced detoriation in accuracy as h is made smaller and smaller.
It is instructive in this analysis to rewrite the numerator of the computed derivative as

$$(f_h - f_0) + (f_{-h} - f_0) = (e^{x+h} - e^x) + (e^{x-h} - e^x),$$

as

$$(f_h - f_0) + (f_{-h} - f_0) = e^x(e^h + e^{-h} - 2),$$

since it is the difference $(e^h + e^{-h} - 2)$ which causes the loss of precision.

Error Analysis

x	$h = 0.01$	$h = 0.001$	$h = 0.0001$	$h = 0.0000001$	Exact
0.0	1.000008	1.000000	1.000000	1.010303	1.000000
1.0	2.718304	2.718282	2.718282	2.753353	2.718282
2.0	7.389118	7.389057	7.389056	7.283063	7.389056
3.0	20.085704	20.085539	20.085537	20.250467	20.085537
4.0	54.598605	54.598155	54.598151	54.711789	54.598150
5.0	148.414396	148.413172	148.413161	150.635056	148.413159

Error Analysis

The results, for $x = 10$ are shown in the Table

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
10^{-1}	2.0100083361116070	$1.0008336111607230 \times 10^{-2}$
10^{-2}	2.0001000008333358	$1.0000083333605581 \times 10^{-4}$
10^{-3}	2.0000010000000836	$1.0000000834065048 \times 10^{-6}$
10^{-5}	2.0000000099999999	$1.0000000050247593 \times 10^{-8}$
10^{-5}	2.0000000001000000	$9.9999897251734637 \times 10^{-11}$
10^{-6}	2.0000000000010001	$9.9997787827987850 \times 10^{-13}$
10^{-7}	2.0000000000000098	$9.9920072216264089 \times 10^{-15}$
10^{-8}	2.0000000000000000	$0.0000000000000000 \times 10^{0}$
10^{-9}	2.0000000000000000	$1.1102230246251565 \times 10^{-16}$
10^{-10}	2.0000000000000000	$0.0000000000000000 \times 10^{0}$

Week 35

- Monday: Repetition from last week
- Numerical differentiation
- C/C++ programming details, pointers, read/write to/from file
- Wednesday: Intro to linear Algebra and possible presentation of project 1.
- Matrices in C++ and Fortran90/95
- Gaussian elimination
- Computer-Lab: Exercise 3 and start project 1.

Technical Matter in C/C++: Pointer example I

```
1  using namespace std; // note use of namespace
2  int main()
3 {
4   int var;
5   int *p;
6   p = &var;
7   var  = 421;
8   printf("Address of integer variable var : %p\n",&var);
9   printf("Its value: %d\n", var);
10  printf("Value of integer pointer p : %p\n",p);
11  printf("The value p points at :  %d\n",*p);
12  printf("Address of the pointer p : %p\n",&p);
13  return 0;
14 }
```

Pointer example I

Discussion

Line	Comments
4	• Defines an integer variable var.
5	• Define an integer pointer – reserves space in memory.
6	• The content of the address of pointer is the address of var.
7	• The value of var is 421.
8	• Writes the address of var in hexadecimal notation for pointers %p.
9	• Writes the value of var in decimal notation%d.

Pointer example II

```
        ....
5  int matr[2];
6  int *p;
7  p = &matr[0];
8  matr[0] = 321;
9  matr[1] = 322;
   printf("\nAddress of matrix element matr[1]: %p",&matr[0]);
   printf("\nValue of the  matrix element  matr[1]; %d",matr[0]);
   printf("\nAddress of matrix element matr[2]: %p",&matr[1]);
   printf("\nValue of the matrix element  matr[2]: %d\n", matr[1]);
   printf("\nValue of the pointer p: %p",p);
   printf("\nThe value p points to: %d",*p);
   printf("\nThe value that (p+1) points to  %d\n",*(p+1));
   printf("\nAddress of pointer p : %p\n",&p);
        ...
```

Pointer example II

Discussion

Line	
5	• Declaration of an integer array matr with two elements
6	• Declaration of an integer pointer
7	• The pointer is initialized to point at the first element of the array matr.
8–9	• Values are assigned to the array matr.

Pointer example II

Discussion

The ouput of this example, compiled again with c++, is

```
Address of the matrix element matr[1]: 0xbfffef70
Value of the  matrix element  matr[1]; 321
Address of the matrix element matr[2]: 0xbfffef74
Value of the matrix element  matr[2]: 322
Value of the pointer: 0xbfffef70
The value pointer points at: 321
The value that (pointer+1) points at:  322
Address of the pointer variable : 0xbfffef6c
```

File handling, C-way

```
using namespace std;
#include <iostream>
int main(int argc, char *argv[])
{
  FILE *in_file, *out_file;
  if( argc < 3)  {
    printf("The programs has the following structure :\n");
    printf("write in the name of the input and output files \n");
    exit(0);
  }
  in_file = fopen( argv[1], "r");// returns pointer to the  input file
  if( in_file == NULL )  { // NULL means that the file is missing
    printf("Can't find the input file %s\n", argv[1]);
    exit(0);
}
```

File handling, C way contn

```
 out_file = fopen( argv[2], "w"); // returns a pointer to the output f
 if( out_file == NULL )  {       // can't find the file
    printf("Can't find the output file%s\n", argv[2]);
    exit(0);
  }
  fclose(in_file);
  fclose(out_file);
  return 0;
}
```

File handling, C++-way

You must first declare input and output files

```
#include <fstream>


// input and output file as global variable
ofstream ofile;
ifstream ifile;
```

File handling, C++-way

```
int main(int argc, char* argv[])
{
  char *outfilename;
  //Read in output file, abort if there are too
  //few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  .....
  ofile.close();  // close output file
```

File handling, C++-way

```
void output(double r_min , double r_max, int max_step,
            double *d)
{
int i;
ofile << "RESULTS:" << endl;
ofile << setiosflags(ios::showpoint | ios::uppercase);
ofile <<"R_min = " << setw(15) << setprecision(8) <<r_min <<endl;
ofile <<"R_max = " << setw(15) << setprecision(8) <<r_max <<endl;
ofile <<"Number of steps = " << setw(15) << max_step << endl;
ofile << "Five lowest eigenvalues:" << endl;
for(i = 0; i < 5; i++) {
    ofile << setw(15) << setprecision(8) << d[i] << endl;
}
}  // end of function output
```

File handling, C++-way

```
int main(int argc, char* argv[])
{
  char *infilename;
  // Read in input file, abort if there are too
  // few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also input file on same line" << endl;
    exit(1);
  }
  else{
    infilename=argv[1];
  }
  ifile.open(infilename);
  ....
  ifile.close();  // close input file
```

File handling, C++-way

```
const char* filename1 = "myfile";
ifstream ifile(filename1);
string filename2 = filename1 + ".out"
ofstream ofile(filename2);  // new output file
ofstream ofile(filename2, ios_base::app);  // append
```

Read something from the file:

```
double a; int b; char c[200];
ifile >> a >> b >> c;  // skips white space in between
```

Can test on success of reading:

```
if (!(ifile >> a >> b >> c)) ok = 0;
```

Call by value and reference

```
int main(int argc, char *argv[])
{
    int a:                                  // line 1
    int *b;                                 // line 2

    a = 10;                                 // line 3
    b = new int[10];                        // line 4
    for(i = 0; i < 10; i++) {
       b[i] = i;                            // line 5
    }
    func( a,b);                             // line 6
  return 0;
}  // End: function main()
```

Call by value and reference

```
void func( int x, int *y)                // line 7
{
    x += 7;                              // line 8
    *y += 10;                            // line 9
    y[6] += 10;                          // line 10
    return;                              // line 11
}   // End: function func()
```

Call by value and reference

- Lines 1,2: Declaration of two variables a and b. The compiler reserves two locations in memory. The size of the location depends on the type of variable. Two properties are important for these locations – the address in memory and the content in the location.

  The value of a: a. The address of a: &a
  The value of b: *b. The address of b: &b.

- Line 3: The value of a is now 10.

- Line 4: Memory to store 10 integers is reserved. The address to the first location is stored in b. Address to element number 6 is given by the expression (b + 6).

- Line 5: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;

Call by value and reference

- Line 6: The main() function calls the function func() and the program counter transfers to the first statement in func(). With respect to data the following happens. The content of a (= 10) and the content of b (a memory address) are copied to a stack (new memory location) associated with the function func()
- Line 7: The variable x and y are local variables in func(). They have the values – x = 10, y = address of the first element in b in the main().
- Line 8: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().

Call by value and reference

- Line 9: The value of y is an address and the symbol *y means the position in memory which has this address. The value in this location is now increased by 10. This means that the value of b[0] in the main program is equal to 10. Thus func() has modified a value in main().
- Line 10: This statement has the same effect as line 9 except that it modifies the element b[6] in main() by adding a value of 10 to what was there originally, namely 5.
- Line 11: The program counter returns to main(), the next expression after *func(a,b);*. All data on the stack associated with func() are destroyed.

Call by value and reference

- The value of a is transferred to func() and stored in a new memory location called x. Any modification of x in func() does not affect in any way the value of a in main(). This is called **transfer of data by value**. On the other hand the next argument in func() is an address which is transferred to func(). This address can be used to modify the corresponding value in main(). In the C language it is expressed as a modification of the value which y points to, namely the first element of b. This is called **transfer of data by reference** and is a method to transfer data back to the calling function, in this case main().

Call by value and reference

C++ allows however the programmer to use solely call by reference (note that call by reference is implemented as pointers). To see the difference between C and C++, consider the following simple examples. In C we would write

```
int n; n =8;
func(&n); /* &n is a pointer to n */
....
void func(int *i)
{
  *i = 10; /* n is changed to 10 */
  ....
}
```

whereas in C++ we would write

```
int n; n =8;
func(n); // just transfer n itself
....
void func(int& i)
{
  i = 10; // n is changed to 10
  ....
}
```

The reason why we emphasize the difference between call by value and call by reference is that it allows the programmer to avoid pitfalls like unwanted changes of variables. However, many people feel that this reduces the readability of the code.

Call by value and reference, F90/95

In Fortran we can use INTENT(IN), INTENT(OUT), INTENT(INOUT) to let the program know which values should or should not be changed.

```
SUBROUTINE coulomb_integral(np,lp,n,l,coulomb)
  USE effective_interaction_declar
  USE energy_variables
  USE wave_functions
  IMPLICIT NONE
  INTEGER, INTENT(IN)  :: n, l, np, lp
  INTEGER :: i
  REAL(KIND=8), INTENT(INOUT) :: coulomb
  REAL(KIND=8) :: z_rel, oscl_r, sum_coulomb
  ...
```

This hinders unwanted changes and increases readability.

Important Matrix and vector handling packages

The Numerical Recipes codes have been rewritten in Fortran 90/95 and C/C++ by us. The original source codes are taken from the widely used software package LAPACK, which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK.

- LINPACK: package for linear equations and least square problems.
- LAPACK:package for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website `http://www.netlib.org` it is possible to download for free all source codes from this library. Both C/C++ and Fortran versions are available.
- BLAS (I, II and III): (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Blas I is vector operations, II vector-matrix operations and III matrix-matrix operations. Highly parallelized and efficient codes, all available for download from `http://www.netlib.org`.

Basic Matrix Features

Matrix Properties Reminder

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The inverse of a matrix is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = I$$

Basic Matrix Features

Matrix Properties Reminder

Relations	Name	matrix elements
$\mathbf{A} = \mathbf{A}^T$	symmetric	$a_{ij} = a_{ji}$
$\mathbf{A} = \left(\mathbf{A}^T\right)^{-1}$	real orthogonal	$\sum_k a_{ik} a_{jk} = \sum_k a_{ki} a_{kj} = \delta_{ij}$
$\mathbf{A} = \mathbf{A}^*$	real matrix	$a_{ij} = a_{ij}^*$
$\mathbf{A} = \mathbf{A}^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$\mathbf{A} = \left(\mathbf{A}^\dagger\right)^{-1}$	unitary	$\sum_k a_{ik} a_{jk}^* = \sum_k a_{ki}^* a_{kj} = \delta_{ij}$

Some famous Matrices

1. Diagonal if $a_{ij} = 0$ for $i \neq j$
2. Upper triangular if $a_{ij} = 0$ for $i > j$
3. Lower triangular if $a_{ij} = 0$ for $i < j$
4. Upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$
5. Lower Hessenberg if $a_{ij} = 0$ for $i < j + 1$
6. Tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$
7. Lower banded with bandwidth p $a_{ij} = 0$ for $i > j + p$
8. Upper banded with bandwidth p $a_{ij} = 0$ for $i < j + p$
9. Banded, block upper triangular, block lower triangular....

Basic Matrix Features

Some Equivalent Statements

For an $N \times N$ matrix $\mathbf{A}$ the following properties are all equivalent

1. If the inverse of $\mathbf{A}$ exists, $\mathbf{A}$ is nonsingular.
2. The equation $\mathbf{A}\mathbf{x} = 0$ implies $\mathbf{x} = 0$.
3. The rows of $\mathbf{A}$ form a basis of R^N.
4. The columns of $\mathbf{A}$ form a basis of R^N.
5. $\mathbf{A}$ is a product of elementary matrices.
6. 0 is not eigenvalue of $\mathbf{A}$.

Important Mathematical Operations

The basic matrix operations that we will deal with are addition and subtraction

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij}, \tag{16}$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma \mathbf{B} \Longrightarrow a_{ij} = \gamma b_{ij}, \tag{17}$$

vector-matrix multiplication

$$\mathbf{y} = \mathbf{A}\mathbf{x} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij} x_j, \tag{18}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{B}\mathbf{C} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \tag{19}$$

and transposition

$$\mathbf{A} = \mathbf{B}^T \Longrightarrow a_{ij} = b_{ji} \tag{20}$$

Important Mathematical Operations

Similarly, important vector operations that we will deal with are addition and subtraction

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \Longrightarrow x_i = y_i \pm z_i, \tag{21}$$

scalar-vector multiplication

$$\mathbf{x} = \gamma \mathbf{y} \Longrightarrow x_i = \gamma y_i, \tag{22}$$

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{y}\mathbf{z} \Longrightarrow x_i = y_i z_i, \tag{23}$$

the inner or so-called dot product resulting in a constant

$$x = \mathbf{y}^T \mathbf{z} \Longrightarrow x = \sum_{j=1}^{n} y_j z_j, \tag{24}$$

and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \Longrightarrow a_{ij} = y_i z_j, \tag{25}$$

Matrix Handling in C/C++, 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

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
- Use the functions provided in the library package lib.cpp
- Use Blitz++

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;
```

Week 36

Linear Algebra

- Monday: Repetition from last week
- Discussion of Project 1, deadline 14 september (midnight).
- Gaussian elimination, LU decomposition and linear equations
- Dynamic memory allocation in C/C++ and Fortran90/95 , use of the library package Blitz++ for C++ users. How to use the C/C++ and Fortran 90/95 libraries.
- Wednesday: Further discussion of linear algebra methods, numerical stability
- Inverse of a matrix
- Computer-Lab: Project 1.

Gaussian Elimination

We start with the linear set of equations

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

We assume also that the matrix **A** is non-singular and that the matrix elements along the diagonal satisfy $a_{ii} \neq 0$. Simple 4×4 example

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}.$$

or

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4. \end{aligned}$$

Gaussian Elimination

The basic idea of Gaussian elimination is to use the first equation to eliminate the first unknown x_1 from the remaining $n-1$ equations. Then we use the new second equation to eliminate the second unknown x_2 from the remaining $n-2$ equations. With $n-1$ such eliminations we obtain a so-called upper triangular set of equations of the form

$$\begin{aligned}
b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= y_1 \\
b_{22}x_2 + b_{23}x_3 + b_{24}x_4 &= y_2 \\
b_{33}x_3 + b_{34}x_4 &= y_3 \\
b_{44}x_4 &= y_4.
\end{aligned}$$

We can solve this system of equations recursively starting from x_n (in our case x_4) and proceed with what is called a backward substitution. This process can be expressed mathematically as

$$x_m = \frac{1}{b_{mm}}\left(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\right) \qquad m = n-1, n-2, \dots, 1. \tag{26}$$

To arrive at such an upper triangular system of equations, we start by eliminating the unknown x_1 for $j = 2, n$. We achieve this by multiplying the first equation by a_{j1}/a_{11} and then subtract the result from the jth equation. We assume obviously that $a_{11} \neq 0$ and that $\mathbf{A}$ is not singular.

Gaussian Elimination

Our actual 4×4 example reads after the first operation

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ w_2^{(2)} \\ w_3^{(2)} \\ w_4^{(2)} \end{pmatrix},$$

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= y_1 \\ a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= w_2^{(2)} \\ a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 &= w_3^{(2)} \\ a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 &= w_4^{(2)}, \end{aligned} \tag{27}$$

Gaussian Elimination

The new coefficients are

$$b_{1k} = a_{1k}^{(1)} \; k = 1, \ldots, n, \tag{28}$$

where each $a_{1k}^{(1)}$ is equal to the original a_{1k} element. The other coefficients are

$$a_{jk}^{(2)} = a_{jk}^{(1)} - \frac{a_{j1}^{(1)} a_{1k}^{(1)}}{a_{11}^{(1)}} \; j, k = 2, \ldots, n, \tag{29}$$

with a new right-hand side given by

$$y_1 = w_1^{(1)}, \; w_j^{(2)} = w_j^{(1)} - \frac{a_{j1}^{(1)} w_1^{(1)}}{a_{11}^{(1)}} \; j = 2, \ldots, n. \tag{30}$$

We have also set $w_1^{(1)} = w_1$, the original vector element. We see that the system of unknowns $x_1, \ldots, x_n$ is transformed into an $(n-1) \times (n-1)$ problem.

Gaussian Elimination

This step is called forward substitution. Proceeding with these substitutions, we obtain the general expressions for the new coefficients

$$a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)} a_{mk}^{(m)}}{a_{mm}^{(m)}} \; j,k = m+1,\dots,n, \tag{31}$$

with $m = 1,\dots,n-1$ and a right-hand side given by

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)} w_m^{(m)}}{a_{mm}^{(m)}} \; j = m+1,\dots,n. \tag{32}$$

This set of $n-1$ elimations leads us to an equations which is solved by back substitution. If the arithmetics is exact and the matrix **A** is not singular, then the computed answer will be exact.

Even though the matrix elements along the diagonal are not zero, numerically small numbers may appear and subsequent divisions may lead to large numbers, which, if added to a small number may yield losses of precision. Suppose for example that our first division in $(a_{22} - a_{21}a_{12}/a_{11})$ results in -10^{-7} and that a_{22} is one. one. We are then adding $10^7 + 1$. With single precision this results in 10^7.

Gaussian Elimination and Tridiagonal matrices, project 1

Suppose we want to solve the following boundary value equation

$$-\frac{d^2u(x)}{dx^2} = f(x, u(x)),$$

with $x \in (a, b)$ and with boundary conditions $u(a) = u(b) = 0$. We assume that f is a continuous function in the domain $x \in (a, b)$. Since, except the few cases where it is possible to find analytic solutions, we will seek after approximate solutions, we choose to represent the approximation to the second derivative from the previous chapter

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2).$$

We subdivide our interval $x \in (a, b)$ into n subintervals by setting $x_i = ih$, with $i = 0, 1, \dots, n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1, 2, \dots n$ we replace the differential operator with the above formula resulting in

$$u''(x_i) \approx \frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2},$$

which we rewrite as

$$u_i'' \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

Gaussian Elimination and Tridiagonal matrices, project 1

We can rewrite our original differential equation in terms of a discretized equation with approximations to the derivatives as

$$-\frac{u_{i+1}-2u_i+u_{i-i}}{h^2}=f(x_i,u(x_i)),$$

with $i=1,2,\dots,n$. We need to add to this system the two boundary conditions $u(a)=u_0$ and $u(b)=u_{n+1}$. If we define a matrix

$$\mathbf{A}=\frac{1}{h^2}\left(\begin{array}{cccccc} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{array}\right)$$

and the corresponding vectors $\mathbf{u}=(u_1,u_2,\dots,u_n)^T$ and $\mathbf{f}(\mathbf{u})=f(x_1,x_2,\dots,x_n,u_1,u_2,\dots,u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system of linear equations with a large number of unknowns

$$\mathbf{A}\mathbf{u}=\mathbf{f}(\mathbf{u}).$$

Gaussian Elimination and Tridiagonal matrices, project 1

We start with the linear set of equations

$$\mathbf{A}\mathbf{u} = \mathbf{f},$$

where $\mathbf{A}$ is a tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix}$$

where a, b, c are one-dimensional arrays of length $1:n$. In project 1 the arrays a and c are equal, namely $a_i = c_i = -1/h^2$. The matrix is also positive definite.

Gaussian Elimination and Tridiagonal matrices, project 1

We can rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}.$$

Gaussian Elimination and Tridiagonal matrices, project 1

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i,$$

for $i = 1, 2, \ldots, n$. We see that u_{-1} and u_{n+1} are not required and we can set $a_1 = c_n = 0$. In many applications the matrix is symmetric and we have $a_i = c_i$. The algorithm for solving this set of equations is rather simple and requires two steps only, a forward substitution and a backward substitution. These steps are also common to the algorithms based on Gaussian elimination that we discussed previously. However, due to its simplicity, the number of floating point operations is in this case proportional with $O(n)$ while Gaussian elimination requires $2n^3/3 + O(n^2)$ floating point operations.

Gaussian Elimination and Tridiagonal matrices, project 1

In case your system of equations leads to a tridiagonal matrix, it is clearly an overkill to employ Gaussian elimination or the standard LU decomposition. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter 10.
Our algorithm starts with forward substitution with a loop over of the elements i and can be expressed via the following piece of code

```
btemp = b[1];
u[1] = f[1]/btemp;
for(i=2 ; i <= n ; i++) {
   temp[i] = c[i-1]/btemp;
   btemp = b[i]-a[i]*temp[i];
   u[i] = (f[i] - a[i]*u[i-1])/btemp;
}
```

Gaussian Elimination and Tridiagonal matrices, project 1

Note that you should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. Finally we perform the backsubstitution leading to the following code

```
for(i=n-1 ; i >= 1 ; i--) {
   u[i] -= temp[i+1]*u[i+1];
}
```

Gaussian Elimination and Tridiagonal matrices, project 1

Note that our sums start with $i = 1$ and that one should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n - 1$ with u_2 eliminated. However, a tridiagonal matrix problem is not a guarantee that we can find a solution. The matrix $\mathbf{A}$ which rephrases a second derivative in a discretized form

$$
\mathbf{A} = \begin{pmatrix}
2 & -1 & 0 & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & 0 & 0 \\
0 & -1 & 2 & -1 & 0 & 0 \\
0 & \dots & \dots & \dots & \dots & \dots \\
0 & 0 & 0 & -1 & 2 & -1 \\
0 & 0 & 0 & 0 & -1 & 2
\end{pmatrix},
$$

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \geq |a_k| + |c_k|$ for $k = 2, 3, \dots, n-1$. This is a relevant but not sufficient condition to guarantee that the matrix $\mathbf{A}$ yields a solution to a linear equation problem. The matrix needs also to be irreducible. A tridiagonal irreducible matrix means that all the elements a_i and c_i are non-zero. If these two conditions are present, then $\mathbf{A}$ is nonsingular and has a unique LU decomposition.

Project 1, hints

When setting up the algo it is useful to note that the different operations on the matrix (here as a 4×4 case with diagonals d_i and off-diagonals e_i

$$
\begin{pmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix} \rightarrow \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix} \rightarrow \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix}
$$

and finally

$$
\begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & 0 & \tilde{d}_4 \end{pmatrix}
$$

Project 1, hints

We notice the sub-blocks which get repeated

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & 0 & \tilde{d}_4 \end{pmatrix}$$

The matrices we often end up with in rewriting for for example partial differential equations, have the feature that all leading principal submatrices are non-singular. If the matrix is symmetric as well it can be rewritten as $A = LDL^T$ with D the diagonal and we have the following relations $a_{11} = d_1$, $a_{k,k-1} = e_{k-1}d_{k-1}$ for $k = 2, \dots, n$ and finally

$$a_{kk} = d_k + e_{k-1}^2 d_{k-1} = d_k + e_{k-1} a_{k,k-1}$$

for $k = 2, \dots, n$.

LU Decomposition

The LU decomposition method means that we can rewrite this matrix as the product of two matrices **L** and **U** where

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}.$$

LU decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4. \end{aligned}$$

The above set of equations is conveniently solved by using LU decomposition as an intermediate step.

The matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and **A** is non-singular, then the LU factorization is unique and the determinant is given by

$$det\{\mathbf{A}\} = det\{\mathbf{LU}\} = det\{\mathbf{L}\}det\{\mathbf{U}\} = u_{11}u_{22}\dots u_{nn}.$$

Linear Algebra Methods

- Gaussian elimination, $O(2/3n^3)$ flops, general matrix
- LU decomposition, upper triangular and lower tridiagonal matrices, $O(2/3n^3)$ flops, general matrix. Get easily the inverse, determinant and can solve linear equations with back-substitution only, $O(n^2)$ flops
- Cholesky decomposition $A = LL^T$. Real symmetric or hermitian positive definite matrix, $O(1/3n^3)$ flops.
- Tridiagonal linear systems, important for differential equations. Normally positive definite and non-singular. $O(8n)$ flops for symmetric. $A = LDL^T$ with D the diagonal. Special case of banded matrices.
- Singular value decomposition
- the QR method will be discussed in chapter 7 in connection with eigenvalue systems. $O(4/3n^3)$ flops.

How to use the Library functions

Standard C/C++: fetch the files lib.cpp and lib.h. You can make a directory where you store these files, and eventually its compiled version lib.o. The example here is program1.cpp from chapter 6 and performs the matrix inversion.

```
/  Simple matrix inversion example
#include <iostream>
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include    "lib.h"

using namespace std;

/* function declarations */

void inverse(double **, int);
```

How to use the Library functions

```
void inverse(double **a, int n)
{
  int          i,j, *indx;
  double       d, *col, **y;
  // allocate space in memory
  indx = new int[n];
  col  = new double[n];
  y    = (double **) matrix(n, n, sizeof(double));
  ludcmp(a, n, indx, &d);   // LU decompose  a[][]
  printf("\n\nLU form of matrix of a[][]:\n");
  for(i = 0; i < n; i++) {
    printf("\n");
    for(j = 0; j < n; j++) {
      printf(" a[%2d][%2d] = %12.4E",i, j, a[i][j]);
    }
  }
```

How to use the Library functions

```
    // find inverse of a[][] by columns
    for(j = 0; j < n; j++) {
      // initialize right-side of linear equations
      for(i = 0; i < n; i++) col[i] = 0.0;
      col[j] = 1.0;
      lubksb(a, n, indx, col);
      // save result in y[][]
      for(i = 0; i < n; i++) y[i][j] = col[i];
    }   //j-loop over columns
    // return the inverse matrix in a[][]
    for(i = 0; i < n; i++) {
      for(j = 0; j < n; j++) a[i][j] = y[i][j];
    }
    free_matrix((void **) y);     // release local memory
    delete [] col;
    delete []indx;
} // End: function inverse()
```

How to use the Library functions

For Fortran users:

```
PROGRAM matrix
  USE constants
  USE F90library
  IMPLICIT NONE
  !      The definition of the matrix, using dynamic allocation
  REAL(DP), ALLOCATABLE, DIMENSION(:,:) :: a, ainv, unity
  !      the determinant
  REAL(DP) :: d
  !      The size of the matrix
  INTEGER :: n
  ....
  !      Allocate now place in heap for a
  ALLOCATE ( a(n,n), ainv(n,n), unity(n,n) )
```

How to use the Library functions

For Fortran users:

```
  WRITE(6,*) ' The matrix before inversion'
  WRITE(6,'(3F12.6)') a
  ainv=a
  CALL matinv (ainv, n, d)
  ....
  !      get the unity matrix
  unity=MATMUL(ainv,a)
  WRITE(6,*) ' The unity matrix'
  WRITE(6,'(3F12.6)') unity
  !      deallocate all arrays
  DEALLOCATE (a, ainv, unity)
END PROGRAM matrix
```

What is Blitz++ http://www.oonumerics.org/blitz/?

Blitz++ http://www.oonumerics.org/blitz/ is a C++ library whose two main goals are to improve the numerical efficiency of C++ and to extend the conventional dense array model to incorporate new and useful features. Some examples of such extensions are flexible storage formats, tensor notation and index placeholders. It allows you also to write several operations involving vectors and matrices in a simple and clear (from a mathematical point of view) way. The way you would code the addition of two matrices looks very similar to the way it is done in Fortran90/95. The C++ programming language offers many features useful for tackling complex scientific computing problems: inheritance, polymorphism, generic programming, and operator overloading are some of the most important. Unfortunately, these advanced features came with a hefty performance pricetag: until recently, C++ lagged behind Fortran's performance by anywhere from 20% to a factor of ten. It was not uncommon to read in textbooks on high-performance computing that if performance matters, then one should resort to Fortran, preferentiall even Fortran 77.

What is Blitz++ `http://www.oonumerics.org/blitz/`?

As a result, untill very recently, the adoption of C++ for scientific computing has been slow. This has changed quite a lot in the last years and modern C++ compilers with numerical libraries have improved the situation considerably. Recent benchmarks show C++ encroaching steadily on Fortran's high-performance monopoly, and for some benchmarks, C++ is even faster than Fortran! These results are being obtained not through better optimizing compilers, preprocessors, or language extensions, but through the use of template techniques. By using templates cleverly, optimizations such as loop fusion, unrolling, tiling, and algorithm specialization can be performed automatically at compile time.
The features of Blitz++ which are useful for our studies are the dynamical allocation of vectors and matrices and algebraic operations on these objects. In particular, if you access the Blitz++ webpage at `http://www.oonumerics.org/blitz/`, we recommend that you study chapters two and three.

A simple makefile

```
# Path where Blitz is installed
BZDIR = /site/Blitz++-0.9_64
CXX = c++
# Flags for optimizing executables
# CXXFLAGS = -O2 -I$(BZDIR) -ftemplate-depth-30
# Flags for debugging
CXXFLAGS = -ftemplate-depth-30 -g -DBZ_DEBUG -I$(BZDIR)/include
LDFLAGS =
LIBS = -L$(BZDIR)/lib -lblitz -lm
TARGETS = blitz_test
.SUFFIXES: .o.cpp
.cpp.o:
                $(CXX) $(CXXFLAGS) -c $*.cpp
$(TARGETS):
                $(CXX)  $(LDFLAGS)  $@.o -o $@ $(LIBS)
all:
                $(TARGETS)

blitz_test:             blitz_test.o


clean:
                rm -f *.o
```

Blitz++ example

```
//     Simple test case of matrix operations
//     using Blitz++
#include <blitz/array.h>
#include <iostream>
using namespace std;
using namespace blitz;

int main()
{
  // Create two 4x4 arrays.  We want them to look like matrices, so
  // we'll make the valid index range 1..4 (rather than 0..3 which is
  // the default).

  Range r(1,4);
  Array<float,2> A(r,r), B(r,r);
  //  initialize a matrix
  A = 0.;  B = 0.;
```

Blitz++ example

```
// The first will be a Hilbert matrix:
//
// a   =   1
//  ij   -----
//       i+j-1
//
// Blitz++ provides a set of types { firstIndex, secondIndex, ... }
// which act as placeholders for indices.  These can be used directl
// in expressions.  For example, we can fill out the A matrix like t

firstIndex i;    // Placeholder for the first index
secondIndex j;   // Placeholder for the second index

A = 1.0 / (i+j-1);
cout << "A = " << A << endl;
// Now the A matrix has each element equal to a_ij = 1/(i+j-1).
```

Blitz++ example

```
// The matrix B will be the permutation matrix
//
// [ 0 0 0 1 ]
// [ 0 0 1 0 ]
// [ 0 1 0 0 ]
// [ 1 0 0 0 ]
//
// Here are two ways of filling out B:

B = (i == (5-j));          // Using an equation -- a bit cryptic

cout << "B = " << B << endl;

B = 0, 0, 0, 1,            // Using an initializer list
  0, 0, 1, 0,
  0, 1, 0, 0,
  1, 0, 0, 0;

cout << "B = " << B << endl;
```

Blitz++ example

```
// Now some examples of tensor-like notation.

Array<float,3> C(r,r,r);  // A three-dimensional array: 1..4, 1..4,

thirdIndex k;             // Placeholder for the third index

// This expression will set
//
// c    = a   * b
//  ijk    ik    kj

C = A(i,k) * B(k,j);
```

Blitz++ example

```
// In real tensor notation, the repeated k index would imply a
  // contraction (or summation) along k.  In Blitz++, you must explici
  // indicate contractions using the sum(expr, index) function:

  Array<float,2> D(r,r);

  D = sum(A(i,k) * B(k,j), k);

  // The above expression computes the matrix product of A and B.

  cout << "D = " << D << endl;
```

Blitz++ example

```
  // Now let's fill out a two-dimensional array with a radially symmet
  // decaying sinusoid.

  int N = 64;                        // Size of array: N x N
  Array<float,2> F(N,N);
  float midpoint = (N-1)/2.;
  int cycles = 3;
  float omega = 2.0 * M_PI * cycles / double(N);
  float tau = - 10.0 / N;

  F = cos(omega * sqrt(pow2(i-midpoint) + pow2(j-midpoint)))
    * exp(tau * sqrt(pow2(i-midpoint) + pow2(j-midpoint)));

  return 0;
}
```

More about classes in C++ from INF-VERK3830 slides

Blitz++ Library

See at webpage under blitz programs and examples. Here we look at the code linEq.

```
     /*
     ** The template function
     **      solveEq()
     ** solves the set of linear equations,
     ** check the results and print it to
     ** standard output
     */

template <typename T>
void solveEq(INPUTDATA data)
{
  int            row,col;
  T              val;
  Array<T,2>     A(data.dim, data.dim),  A1(data.dim, data.dim);
  Array<T,1>     B(data.dim), X(data.dim);
  Array<int,1>   Index(data.dim);

      // fill the matrix A( , ) and vector B() with data

  matrixVal(data, A, B);
```

Blitz++ Library

```
      /*
       ** In order to check the solution we save the original
       ** matrix A and vector B and use A1 and X
       ** in the calculation since ludcmp() and lubksb() destroy
       ** the original elements
       */

  A1 = A;               // Blitz asignements
  X  = B;

       // necessary parameter if ludcmp() is used to invert a matrix

  T permutation = static_cast<T>(1);
// LU decomposition of A1()
 ludcmp(A1, data.dim, Index, permutation);
```

Blitz++ Library

```
cout << endl << "Time used:  " << ex_time.hour << " hour    "
                                << ex_time.min << " min      "
                                << ex_time.sec << " sec      "
      << endl;
 cout << endl << "The coefficient matrix A1 after
                 ludcmp(A1 = " << A1  << endl;
 lubksb(A1, data.dim, Index, X);  // solve the equations
 cout << endl << endl
      << "Solution to " << data.dim <<" linear equations"
      << endl <<endl;
 cout << endl << "The coefficient matrix A = " << A  << endl;
 cout << endl << "The right-hand matrix  B = " << B  << endl;
 cout << endl << "The solution           X = " << X  << endl;
 checkEq(A, X, B);
```

Blitz++ Library

```
cout << endl << endl
       << "Check the solution B - (A x X) = " << B
       << endl <<endl;

  A.free();    // release memory   -- Blitz methods
  A1.free();
  B.free();
  X.free();
```

Blitz++ Library, output for random matrix

```
Solution to 3 linear equations

The coefficient matrix A = 3 x 3
[  -2.57343  -1.71153   4.39192
   -4.98846  -1.04559  -3.17281
    4.54754  0.539849   3.24798 ]

The right-hand matrix  B = 3
 [  -1.22136   2.67661  -4.17957  ]
The solution            X = 3
 [  -1.79775   4.60816  0.464313  ]
Check the solution B - (A x X) = 3
 [ -1.11022e-15         0          0   ]
```

Week 37

Linear Algebra and differential equations

- Monday: Repetition from last week
- Iterative methods, Gauss-Seidel. Cubic spline and interpolation (chapter 6).
- Start discussion of eigenvalue problems (chapter 7)
- Wednesday: Eigenvalue problems
- We start discussing Jacobi's algorithm, chapter 7.
- Discussion of classes, chapters 3 and 6.
- Presentation of project 2.
- Computer-Lab: Project 2

Iterative methods, Chapter 6

- Direct solvers such as Gauss elimination and LU decomposition discussed last week.
- Iterative solvers such as Basic iterative solvers, Jacobi, Gauss-Seidel, Successive over-relaxation. These methods are easy to parallelize, as we will se later. Much used in solutions of partial differential equations.
- Other iterative methods such as Krylov subspace methods with Generalized minimum residual (GMRES) and Conjugate gradient etc will not be discussed.

Iterative methods, Jacobi's method

It is a simple method for solving

$$\hat{A}\mathbf{x} = \mathbf{b},$$

where $\hat{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors. The vector $\mathbf{x}$ is the unknown.
It is an iterative scheme where we start with a guess for the unknown, and after $k+1$ iterations we have

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L} + \hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.
If the matrix $\hat{A}$ is positive definite or diagonally dominant, one can show that this method will always converge to the exact solution.

Iterative methods, Jacobi's method

We can demonstrate Jacobi's method by this 4×4 matrix problem. We assume a guess for the vector elements $x_i^{(0)}$, a guess which represents our first iteration. The new values are obtained by substitution

$$
\begin{aligned}
x_1^{(1)} &= (b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)})/a_{11} \\
x_2^{(1)} &= (b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)})/a_{22} \\
x_3^{(1)} &= (b_3 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)})/a_{33} \\
x_4^{(1)} &= (b_4 - a_{41}x_1^{(0)} - a_{42}x_2^{(0)} - a_{43}x_3^{(0)})/a_{44},
\end{aligned}
$$

which after $k+1$ iterations reads

$$
\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}
$$

Iterative methods, Jacobi's method

We can generalize the above equations to

$$x_i^{(k+1)} = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)})/a_{ii}$$

or in an even more compact form as

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L} + \hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

Iterative methods, Gauss-Seidel's method

Our 4×4 matrix problem

$$
\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}
$$

can be rewritten as

$$
\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44},
\end{aligned}
$$

which allows us to utilize the preceding solution (forward substitution). This improves normally the convergence behavior and leads to the Gauss-Seidel method!

Iterative methods, Gauss-Seidel's method

We can generalize

$$\begin{aligned}
x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} &= (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44},
\end{aligned}$$

to the following form

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1, 2, \ldots, n.$$

The procedure is generally continued until the changes made by an iteration are below some tolerance.

The convergence properties of the Jacobi method and the Gauss-Seidel method are dependent on the matrix $\hat{A}$. These methods converge when the matrix is symmetric positive-definite, or is strictly or irreducibly diagonally dominant. Both methods sometimes converge even if these conditions are not satisfied.

Iterative methods, Successive over-relaxation

Given a square system of n linear equations with unknown **x**:

$$\hat{A}\mathbf{x} = \mathbf{b}$$

where:

$$\hat{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Iterative methods, Successive over-relaxation

Then A can be decomposed into a diagonal component D, and strictly lower and upper triangular components L and U:

$$\hat{A} = \hat{D} + \hat{L} + \hat{U},$$

where

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

The system of linear equations may be rewritten as:

$$(D + \omega L)\mathbf{x} = \omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}$$

for a constant $\omega > 1$.

Iterative methods, Successive over-relaxation

The method of successive over-relaxation is an iterative technique that solves the left hand side of this expression for x, using previous value for x on the right hand side. Analytically, this may be written as:

$$\mathbf{x}^{(k+1)} = (D+\omega L)^{-1}\big(\omega \mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}^{(k)}\big).$$

However, by taking advantage of the triangular form of $(D+\omega L)$, the elements of $x^{(k+1)}$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1,2,\ldots,n.$$

The choice of relaxation factor is not necessarily easy, and depends upon the properties of the coefficient matrix. For symmetric, positive-definite matrices it can be proven that $0 < \omega < 2$ will lead to convergence, but we are generally interested in faster convergence rather than just convergence.

Cubic Splines, Chapter 6

Cubic spline interpolation is among one of the mostly used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called cubic spline method to be described below.
A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.
Assume we have at our disposal $n+1$ points $x_0, x_1, \dots x_n$ arranged so that $x_0 < x_1 < x_2 < \dots x_{n-1} < x_n$ (such points are called knots). A spline function s of degree k with $n+1$ knots is defined as follows

- On every subinterval $[x_{i-1}, x_i)$ s is a polynomial of degree $\leq k$.
- s has $k-1$ continuous derivatives in the whole interval $[x_0, x_n]$.

Splines

As an example, consider a spline function of degree $k = 1$ defined as follows

$$s(x) = \begin{cases} s_0(x) = a_0 x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1 x + b_1 & x \in [x_1, x_2) \\ \dots & \dots \\ s_{n-1}(x) = a_{n-1} x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases}$$

In this case the polynomial consists of series of straight lines connected to each other at every endpoint. The number of continuous derivatives is then $k - 1 = 0$, as expected when we deal with straight lines. Such a polynomial is quite easy to construct given $n + 1$ points $x_0, x_1, \dots x_n$ and their corresponding function values.

Splines

The most commonly used spline function is the one with $k = 3$, the so-called cubic spline function. Assume that we have in adddition to the $n+1$ knots a series of functions values $y_0 = f(x_0), y_1 = f(x_1), \dots y_n = f(x_n)$. By definition, the polynomials s_{i-1} and s_i are thence supposed to interpolate the same point i, i.e.,

$$s_{i-1}(x_i) = y_i = s_i(x_i),$$

with $1 \leq i \leq n-1$. In total we have n polynomials of the type

$$s_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i2}x^3,$$

yielding $4n$ coefficients to determine.

Splines

Every subinterval provides in addition the $2n$ conditions

$$y_i = s(x_i),$$

and

$$s(x_{i+1}) = y_{i+1},$$

to be fulfilled. If we also assume that s' and s'' are continuous, then

$$s'_{i-1}(x_i) = s'_i(x_i),$$

yields $n-1$ conditions. Similarly,

$$s''_{i-1}(x_i) = s''_i(x_i),$$

results in additional $n-1$ conditions. In total we have $4n$ coefficients and $4n-2$ equations to determine them, leaving us with 2 degrees of freedom to be determined.

Splines

Using the last equation we define two values for the second derivative, namely

$$s_i''(x_i) = f_i,$$

and

$$s_i''(x_{i+1}) = f_{i+1},$$

and setting up a straight line between f_i and f_{i+1} we have

$$s_i''(x) = \frac{f_i}{x_{i+1} - x_i}(x_{i+1} - x) + \frac{f_{i+1}}{x_{i+1} - x_i}(x - x_i),$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1} - x_i)}(x_{i+1} - x)^3 + \frac{f_{i+1}}{6(x_{i+1} - x_i)}(x - x_i)^3 + c(x - x_i) + d(x_{i+1} - x).$$

Splines

Using the conditions $s_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$ we can in turn determine the constants c and d resulting in

$$\begin{aligned} s_i(x) = & \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1} - x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x - x_i)^3 \\ + & (\frac{y_{i+1}}{x_{i+1}-x_i} - \frac{f_{i+1}(x_{i+1}-x_i)}{6})(x - x_i) + (\frac{y_i}{x_{i+1}-x_i} - \frac{f_i(x_{i+1}-x_i)}{6})(x_{i+1} - x). \quad (33) \end{aligned}$$

Splines

How to determine the values of the second derivatives f_i and f_{i+1}? We use the continuity assumption of the first derivatives

$$s'_{i-1}(x_i) = s'_i(x_i),$$

and set $x = x_i$. Defining $h_i = x_{i+1} - x_i$ we obtain finally the following expression

$$h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_i f_{i+1} = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}),$$

and introducing the shorthands $u_i = 2(h_i + h_{i-1})$, $v_i = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1})$, we can reformulate the problem as a set of linear equations to be solved through e.g., Gaussian elemination

Splines

Gaussian elimination

$$\begin{bmatrix} u_1 & h_1 & 0 & \dots & & & & \\ h_1 & u_2 & h_2 & 0 & \dots & & & \\ 0 & h_2 & u_3 & h_3 & 0 & \dots & & \\ \dots & & \dots & \dots & \dots & \dots & \dots & \\ & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & & 0 & h_{n-2} & u_{n-1} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}.$$

Note that this is a set of tridiagonal equations and can be solved through only $O(n)$ operations.

Splines

The functions supplied in the program library are *spline* and *splint*. In order to use cubic spline interpolation you need first to call

```
spline(double x[], double y[], int n, double yp1,
          double yp2, double y2[])
```

This function takes as input $x[0,..,n-1]$ and $y[0,..,n-1]$ containing a tabulation $y_i = f(x_i)$ with $x_0 < x_1 < .. < x_{n-1}$ together with the first derivatives of $f(x)$ at x_0 and x_{n-1}, respectively. Then the function returns $y2[0,..,n-1]$ which contains the second derivatives of $f(x_i)$ at each point x_i. n is the number of points. This function provides the cubic spline interpolation for all subintervals and is called only once.

Splines

Thereafter, if you wish to make various interpolations, you need to call the function

```
splint(double x[], double y[], double y2a[], int n,
                 double x, double *y)
```

which takes as input the tabulated values $x[0, .., n-1]$ and $y[0, .., n-1]$ and the output y2a[0,..,n - 1] from *spline*. It returns the value y corresponding to the point x.

Eigenvalue Solvers

Let us consider the matrix **A** of dimension n. The eigenvalues of **A** is defined through the matrix equation

$$\mathbf{A}\mathbf{x}^{(\nu)} = \lambda^{(\nu)}\mathbf{x}^{(\nu)},$$

where $\lambda^{(\nu)}$ are the eigenvalues and $\mathbf{x}^{(\nu)}$ the corresponding eigenvectors. Unless otherwise stated, when we use the wording eigenvector we mean the right eigenvector. The left eigenvector is defined as

$$\mathbf{x}^{(\nu)}{}_L\mathbf{A} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}{}_L$$

The above right eigenvector problem is equivalent to a set of n equations with n unknowns x_i.

Eigenvalue Solvers

The eigenvalue problem can be rewritten as

$$\left(\mathbf{A} - \lambda^{(\nu)} I\right) \mathbf{x}^{(\nu)} = 0,$$

with I being the unity matrix. This equation provides a solution to the problem if and only if the determinant is zero, namely

$$\left|\mathbf{A} - \lambda^{(\nu)}\mathbf{I}\right| = 0,$$

which in turn means that the determinant is a polynomial of degree n in λ and in general we will have n distinct zeros.

Eigenvalue Solvers

The eigenvalues of a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ are thus the n roots of its characteristic polynomial

$$P(\lambda) = det(\lambda\mathbf{I} - \mathbf{A}),$$

or

$$P(\lambda) = \prod_{i=1}^{n} (\lambda_i - \lambda)\,.$$

The set of these roots is called the spectrum and is denoted as $\lambda(\mathbf{A})$. If $\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ then we have

$$det(\mathbf{A}) = \lambda_1\lambda_2\ldots\lambda_n,$$

and if we define the trace of **A** as

$$Tr(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

then $Tr(\mathbf{A}) = \lambda_1 + \lambda_2 + \cdots + \lambda_n$.

Abel-Ruffini Impossibility Theorem

The Abel-Ruffini theorem (also known as Abel's impossibility theorem) states that there is no general solution in radicals to polynomial equations of degree five or higher. The content of this theorem is frequently misunderstood. It does not assert that higher-degree polynomial equations are unsolvable. In fact, if the polynomial has real or complex coefficients, and we allow complex solutions, then every polynomial equation has solutions; this is the fundamental theorem of algebra. Although these solutions cannot always be computed exactly with radicals, they can be computed to any desired degree of accuracy using numerical methods such as the Newton-Raphson method or Laguerre method, and in this way they are no different from solutions to polynomial equations of the second, third, or fourth degrees.

The theorem only concerns the form that such a solution must take. The content of the theorem is that the solution of a higher-degree equation cannot in all cases be expressed in terms of the polynomial coefficients with a finite number of operations of addition, subtraction, multiplication, division and root extraction. Some polynomials of arbitrary degree, of which the simplest nontrivial example is the monomial equation $ax^n = b$, are always solvable with a radical.

Abel-Ruffini Impossibility Theorem

The Abel-Ruffini theorem says that there are some fifth-degree equations whose solution cannot be so expressed. The equation $x^5 - x + 1 = 0$ is an example. Some other fifth degree equations can be solved by radicals, for example $x^5 - x^4 - x + 1 = 0$. The precise criterion that distinguishes between those equations that can be solved by radicals and those that cannot was given by Galois and is now part of Galois theory: a polynomial equation can be solved by radicals if and only if its Galois group is a solvable group.

Today, in the modern algebraic context, we say that second, third and fourth degree polynomial equations can always be solved by radicals because the symmetric groups S_2, S_3 and S_4 are solvable groups, whereas S_n is not solvable for $n \geq 5$.

Eigenvalue Solvers

In the present discussion we assume that our matrix is real and symmetric, that is $\mathbf{A} \in \mathbb{R}^{n\times n}$. The matrix $\mathbf{A}$ has n eigenvalues $\lambda_1 \dots \lambda_n$ (distinct or not). Let $\mathbf{D}$ be the diagonal matrix with the eigenvalues on the diagonal

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n \end{pmatrix}.$$

If $\mathbf{A}$ is real and symmetric then there exists a real orthogonal matrix $\mathbf{S}$ such that

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n),$$

and for $j = 1 : n$ we have $\mathbf{AS}(:, j) = \lambda_j \mathbf{S}(:, j)$.

Eigenvalue Solvers

To obtain the eigenvalues of $\mathbf{A} \in \mathbb{R}^{n \times n}$, the strategy is to perform a series of similarity transformations on the original matrix $\mathbf{A}$, in order to reduce it either into a diagonal form as above or into a tridiagonal form.
We say that a matrix $\mathbf{B}$ is a similarity transform of $\mathbf{A}$ if

$$\mathbf{B} = \mathbf{S}^T \mathbf{A} \mathbf{S}, \qquad \text{where} \qquad \mathbf{S}^T \mathbf{S} = \mathbf{S}^{-1} \mathbf{S} = \mathbf{I}.$$

The importance of a similarity transformation lies in the fact that the resulting matrix has the same eigenvalues, but the eigenvectors are in general different.

Eigenvalue Solvers

To prove this we start with the eigenvalue problem and a similarity transformed matrix $\mathbf{B}$.

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \qquad \text{and} \qquad \mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}.$$

We multiply the first equation on the left by $\mathbf{S}^T$ and insert $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between $\mathbf{A}$ and $\mathbf{x}$. Then we get

$$(\mathbf{S}^\mathbf{T}\mathbf{A}\mathbf{S})(\mathbf{S}^\mathbf{T}\mathbf{x}) = \lambda\mathbf{S}^\mathbf{T}\mathbf{x}, \tag{34}$$

which is the same as

$$\mathbf{B}\left(\mathbf{S}^\mathbf{T}\mathbf{x}\right) = \lambda\left(\mathbf{S}^\mathbf{T}\mathbf{x}\right).$$

The variable λ is an eigenvalue of $\mathbf{B}$ as well, but with eigenvector $\mathbf{S}^\mathbf{T}\mathbf{x}$.

Eigenvalue Solvers

The basic philosophy is to

- either apply subsequent similarity transformations (direct method) so that

$$\mathbf{S}_N^T \dots \mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 \dots \mathbf{S}_N = \mathbf{D}, \tag{35}$$

- or apply subsequent similarity transformations so that **A** becomes tridiagonal (Householder) or upper/lower triangular (**QR** method). Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.
- or use so-called power methods
- or use iterative methods (Krylov, Lanczos, Arnoldi). These methods are popular for huge matrix problems.

Gaussian Elimination and Tridiagonal matrices, project 1

In project 1 we rewrote our original differential equation in terms of a discretized equation with approximations to the derivatives as

$$-\frac{u_{i+1}-2u_i+u_{i-i}}{h^2}=f(x_i,u(x_i)),$$

with $i=1,2,\dots,n$. We need to add to this system the two boundary conditions $u(a)=u_0$ and $u(b)=u_{n+1}$. If we define a matrix

$$\mathbf{A}=\frac{1}{h^2}\left(\begin{array}{cccccc} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{array}\right)$$

and the corresponding vectors $\mathbf{u}=(u_1,u_2,\dots,u_n)^T$ and $\mathbf{f}(\mathbf{u})=f(x_1,x_2,\dots,x_n,u_1,u_2,\dots,u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system of linear equations with a large number of unknowns

$$\mathbf{Au}=\mathbf{f}(\mathbf{u}).$$

Project 2, part a-b)

We are first interested in the solution of the radial part of Schrödinger's equation for one electron. This equation reads

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r).$$

In our case $V(r)$ is the harmonic oscillator potential $(1/2)kr^2$ with $k=m\omega^2$ and E is the energy of the harmonic oscillator in three dimensions. The oscillator frequency is ω and the energies are

$$E_{nl}=\hbar\omega\left(2n+l+\frac{3}{2}\right),$$

with $n=0,1,2,\dots$ and $l=0,1,2,\dots$.
Since we have made a transformation to spherical coordinates it means that $r\in[0,\infty)$. The quantum number l is the orbital momentum of the electron. Then we substitute $R(r)=(1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\left(V(r)+\frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r)=Eu(r).$$

The boundary conditions are $u(0)=0$ and $u(\infty)=0$.

Project 2, part a-b)

We introduce a dimensionless variable $\rho = (1/\alpha) r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho) + \left(V(\rho) + \frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho) = Eu(\rho).$$

In project 2 we set $l = 0$. Inserting $V(\rho) = (1/2)k\alpha^2\rho^2$ we end up with

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho) + \frac{k}{2}\alpha^2\rho^2 u(\rho) = Eu(\rho).$$

We multiply thereafter with $2m\alpha^2/\hbar^2$ on both sides and obtain

$$-\frac{d^2}{d\rho^2}u(\rho) + \frac{mk}{\hbar^2}\alpha^4\rho^2 u(\rho) = \frac{2m\alpha^2}{\hbar^2}Eu(\rho).$$

Project 2, part a-b)

We have from the previous slide

$$-\frac{d^2}{d\rho^2}u(\rho)+\frac{mk}{\hbar^2}\alpha^4\rho^2u(\rho)=\frac{2m\alpha^2}{\hbar^2}Eu(\rho).$$

The constant α can now be fixed so that

$$\frac{mk}{\hbar^2}\alpha^4=1,$$

or

$$\alpha=\left(\frac{\hbar^2}{mk}\right)^{1/4}.$$

Defining

$$\lambda=\frac{2m\alpha^2}{\hbar^2}E,$$

we can rewrite Schrödinger's equation as

$$-\frac{d^2}{d\rho^2}u(\rho)+\rho^2u(\rho)=\lambda u(\rho).$$

This is the first equation to solve numerically. In three dimensions the eigenvalues for $l=0$ are $\lambda_0=3,\lambda_1=7,\lambda_2=11,\dots.$

Project 2, part a-b)

We use the by now standard expression for the second derivative of a function u

$$u'' = \frac{u(\rho + h) - 2u(\rho) + u(\rho - h)}{h^2} + O(h^2), \tag{36}$$

where h is our step. Next we define minimum and maximum values for the variable ρ, $\rho_{\mathrm{min}} = 0$ and ρ_{max}, respectively. You need to check your results for the energies against different values ρ_{max}, since we cannot set $\rho_{\mathrm{max}} = \infty$.

Project 2, part a-b)

With a given number of steps, n_{step}, we then define the step h as

$$h = \frac{\rho_{\mathrm{max}} - \rho_{\mathrm{min}}}{n_{\mathrm{step}}}.$$

Define an arbitrary value of ρ as

$$\rho_i = \rho_{\mathrm{min}} + ih \qquad i = 0, 1, 2, \dots, n_{\mathrm{step}}$$

we can rewrite the Schrödinger equation for ρ_i as

$$-\frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} + \rho_i^2 u(\rho_i) = \lambda u(\rho_i),$$

or in a more compact way

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \rho_i^2 u_i = -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + V_i u_i = \lambda u_i,$$

where $V_i = \rho_i^2$ is the harmonic oscillator potential.

Project 2, part a-b)

Define first the diagonal matrix element

$$d_i = \frac{2}{h^2} + V_i,$$

and the non-diagonal matrix element

$$e_i = -\frac{1}{h^2}.$$

In this case the non-diagonal matrix elements are given by a mere constant. *All non-diagonal matrix elements are equal*. With these definitions the Schrödinger equation takes the following form

$$d_i u_i + e_{i-1} u_{i-1} + e_{i+1} u_{i+1} = \lambda u_i,$$

where u_i is unknown. We can write the latter equation as a matrix eigenvalue problem

$$\left(\begin{array}{ccccccc} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{n_{\mathrm{step}}-2} & e_{n_{\mathrm{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n_{\mathrm{step}}-1} & d_{n_{\mathrm{step}}} \end{array} \right) \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\mathrm{step}}-1} \end{array} \right) = \lambda \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\mathrm{step}}-1} \end{array} \right) \tag{37}$$

Project 2, part a-b)

or if we wish to be more detailed, we can write the tridiagonal matrix as

$$\left(\begin{array}{ccccccc} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{n_{\mathrm{step}}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{n_{\mathrm{step}}-1} \end{array}\right) \quad (38)$$

Recall that the solutions are known via the boundary conditions at $i = n_{\mathrm{step}}$ and at the other end point, that is for ρ_0. The solution is zero in both cases.

Project 2, part c)

We are going to study two electrons in a harmonic oscillator well which also interact via a repulsive Coulomb interaction. Let us start with the single-electron equation written as

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\frac{1}{2}kr^2u(r)=E^{(1)}u(r),$$

where $E^{(1)}$ stands for the energy with one electron only. For two electrons with no repulsive Coulomb interaction, we have the following Schrödinger equation

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dr_1^2}-\frac{\hbar^2}{2m}\frac{d^2}{dr_2^2}+\frac{1}{2}kr_1^2+\frac{1}{2}kr_2^2\right)u(r_1,r_2)=E^{(2)}u(r_1,r_2).$$

Project 2, part c)

Note that we deal with a two-electron wave function $u(r_1, r_2)$ and two-electron energy $E^{(2)}$.

With no interaction this can be written out as the product of two single-electron wave functions, that is we have a solution on closed form.

We introduce the relative coordinate $\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$ and the center-of-mass coordinate $\mathbf{R} = 1/2(\mathbf{r}_1 + \mathbf{r}_2)$. With these new coordinates, the radial Schrödinger equation reads

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} - \frac{\hbar^2}{4m}\frac{d^2}{dR^2} + \frac{1}{4}kr^2 + kR^2\right) u(r,R) = E^{(2)} u(r,R).$$

Project 2, part c)

The equations for r and R can be separated via the ansatz for the wave function $u(r,R)=\psi(r)\phi(R)$ and the energy is given by the sum of the relative energy E_r and the center-of-mass energy E_R, that is

$$E^{(2)} = E_r + E_R.$$

We add then the repulsive Coulomb interaction between two electrons, namely a term

$$V(r_1,r_2) = \frac{\beta e^2}{|\mathbf{r}_1-\mathbf{r}_2|} = \frac{\beta e^2}{r},$$

with $\beta e^2 = 1.44$ eVnm.

Project 2, part c)

Adding this term, the r-dependent Schrödinger equation becomes

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2}+\frac{1}{4}kr^2+\frac{\beta e^2}{r}\right)\psi(r)=E_r\psi(r).$$

This equation is similar to the one we had previously in parts (a) and (b) and we introduce again a dimensionless variable $\rho=r/\alpha$. Repeating the same steps, we arrive at

$$-\frac{d^2}{d\rho^2}\psi(\rho)+\frac{mk}{4\hbar^2}\alpha^4\rho^2\psi(\rho)+\frac{m\alpha\beta e^2}{\rho\hbar^2}\psi(\rho)=\frac{m\alpha^2}{\hbar^2}E_r\psi(\rho).$$

Project 2, part c)

We want to manipulate this equation further to make it as similar to that in (a) as possible. We define a 'frequency'

$$\omega_r^2 = \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4,$$

and fix the constant α by requiring

$$\frac{m\alpha\beta e^2}{\hbar^2} = 1$$

or

$$\alpha = \frac{\hbar^2}{m\beta e^2}.$$

Project 2, part c)

Defining

$$\lambda = \frac{m\alpha^2}{\hbar^2} E,$$

we can rewrite Schrödinger's equation as

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \omega_r^2\rho^2\psi(\rho) + \frac{1}{\rho}\psi(\rho) = \lambda\psi(\rho).$$

Project 2, part c)

We treat ω_r as a parameter which reflects the strength of the oscillator potential.
Here we will study the cases $\omega_r = 0.01$, $\omega_r = 0.5$, $\omega_r = 1$, and $\omega_r = 5$ for the ground state only, that is the lowest-lying state.
With no repulsive Coulomb interaction you should get a result which corresponds to the relative energy of a non-interacting system. Make sure your results are stable as functions of $\rho_{\max}$ and the number of steps.
We are only interested in the ground state with $l = 0$. We omit the center-of-mass energy.

For specific oscillator frequencies, the above equation has analytic answers, see the article by M. Taut, Phys. Rev. A 48, 3561 - 3566 (1993). The article can be retrieved from the following web address

`http://prola.aps.org/abstract/PRA/v48/i5/p3561_1`.

Eigenvalue Solvers, Jacobi

Consider an $(n \times n)$ orthogonal transformation matrix

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & \cos\theta & 0 & \dots & 0 & \sin\theta \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & -\sin\theta & 0 & \dots & 1 & \cos\theta \\ 0 & 0 & \dots & 0 & \dots & \dots & 0 & 1 \end{pmatrix}$$

with property $\mathbf{S}^{\mathbf{T}} = \mathbf{S}^{-1}$. It performs a plane rotation around an angle θ in the Euclidean $n-$dimensional space.

Eigenvalue Solvers, Jacobi

It means that its matrix elements that differ from zero are given by

$$s_{kk} = s_{ll} = \cos\theta, s_{kl} = -s_{lk} = -\sin\theta, s_{ii} = -s_{ii} = 1 \qquad i \neq k \qquad i \neq l,$$

A similarity transformation

$$\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S},$$

results in

$$\begin{aligned}
b_{ik} &= a_{ik}\cos\theta - a_{il}\sin\theta, i \neq k, i \neq l \\
b_{il} &= a_{il}\cos\theta + a_{ik}\sin\theta, i \neq k, i \neq l \\
b_{kk} &= a_{kk}\cos^2\theta - 2a_{kl}\cos\theta\sin\theta + a_{ll}\sin^2\theta \\
b_{ll} &= a_{ll}\cos^2\theta + 2a_{kl}\cos\theta sin\theta + a_{kk}\sin^2\theta \\
b_{kl} &= (a_{kk} - a_{ll})\cos\theta\sin\theta + a_{kl}(\cos^2\theta - \sin^2\theta)
\end{aligned}$$

The angle θ is arbitrary. The recipe is to choose θ so that all non-diagonal matrix elements b_{kl} become zero.

Eigenvalue Solvers, Jacobi

The main idea is thus to reduce systematically the norm of the off-diagonal matrix elements of a matrix $\mathbf{A}$

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2}.$$

To demonstrate the algorithm, we consider the simple 2×2 similarity transformation of the full matrix. The matrix is symmetric, we single out $1 \leq k < l \leq n$ and use the abbreviations $c = \cos\theta$ and $s = \sin\theta$ to obtain

$$\begin{pmatrix} b_{kk} & 0 \\ 0 & b_{ll} \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_{kk} & a_{kl} \\ a_{lk} & a_{ll} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}.$$

Eigenvalue Solvers, Jacobi

We require that the non-diagonal matrix elements $b_{kl} = b_{lk} = 0$, implying that

$$a_{kl}(c^2 - s^2) + (a_{kk} - a_{ll})cs = b_{kl} = 0.$$

If $a_{kl} = 0$ one sees immediately that $\cos\theta = 1$ and $\sin\theta = 0$.
The Frobenius norm of an orthogonal transformation is always preserved. The Frobenius norm is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2}.$$

This means that for our 2×2 case we have

$$2a_{kl}^2 + a_{kk}^2 + a_{ll}^2 = b_{kk}^2 + b_{ll}^2,$$

which leads to

$$\mathrm{off}(\mathbf{B})^2 = ||\mathbf{B}||_F^2 - \sum_{i=1}^{n} b_{ii}^2 = \mathrm{off}(\mathbf{A})^2 - 2a_{kl}^2,$$

since

$$||\mathbf{B}||_F^2 - \sum_{i=1}^{n} b_{ii}^2 = ||\mathbf{A}||_F^2 - \sum_{i=1}^{n} a_{ii}^2 + (a_{kk}^2 + a_{ll}^2 - b_{kk}^2 - b_{ll}^2).$$

This results means that the matrix **A** moves closer to diagonal form for each transformation.

Eigenvalue Solvers, Jacobi

Defining the quantities $\tan\theta = t = s/c$ and

$$\cot 2\theta = \tau = \frac{a_{ll} - a_{kk}}{2a_{kl}},$$

we obtain the quadratic equation (using $\cot 2\theta = 1/2(\cot\theta - \tan\theta)$)

$$t^2 + 2\tau t - 1 = 0,$$

resulting in

$$t = -\tau \pm \sqrt{1+\tau^2},$$

and c and s are easily obtained via

$$c = \frac{1}{\sqrt{1+t^2}},$$

and $s = tc$. Choosing t to be the smaller of the roots ensures that $|\theta| \leq \pi/4$ and has the effect of minimizing the difference between the matrices **B** and **A** since

$$||\mathbf{B} - \mathbf{A}||_F^2 = 4(1-c)\sum_{i=1, i\neq k,l}^{n} (a_{ik}^2 + a_{il}^2) + \frac{2a_{kl}^2}{c^2}.$$

Eigenvalue Solvers, Jacobi algo

- Choose a tolerance ϵ, making it a small number, typically 10^{-8} or smaller.
- Setup a **while**-test where one compares the norm of the newly computed off-diagonal matrix elements

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} a_{ij}^2} > \epsilon.$$

- Now choose the matrix elements a_{kl} so that we have those with largest value, that is $|a_{kl}| = \max_{i\neq j}|a_{ij}|$.
- Compute thereafter $\tau = (a_{ll} - a_{kk})/2a_{kl}$, $\tan\theta$, $\cos\theta$ and $\sin\theta$.
- Compute thereafter the similarity transformation for this set of values (k, l), obtaining the new matrix $\mathbf{B} = \mathbf{S}(k, l, \theta)^T\mathbf{A}\mathbf{S}(k, l, \theta)$.
- Compute the new norm of the off-diagonal matrix elements and continue till you have satisfied $\mathrm{off}(\mathbf{B}) \leq \epsilon$

The convergence rate of the Jacobi method is however poor, one needs typically $3n^2 - 5n^2$ rotations and each rotation requires $4n$ operations, resulting in a total of $12n^3 - 20n^3$ operations in order to zero out non-diagonal matrix elements.

Jacobi's method, an example to convice you about the algorithm

We specialize to a symmetric 3×3 matrix $\mathbf{A}$. We start the process as follows (assuming that $a_{23} = a_{32}$ is the largest non-diagonal) with $c = \cos\theta$ and $s = \sin\theta$

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{pmatrix}.$$

We will choose the angle θ in order to have $a_{23} = a_{32} = 0$. We get (symmetric matrix)

$$\mathbf{B} = \begin{pmatrix} a_{11} & a_{12}c - a_{13}s & a_{12}s + a_{13}c \\ a_{12}c - a_{13}s & a_{22}c^2 + a_{33}s^2 - 2a_{23}sc & (a_{22} - a_{33})sc + a_{23}(c^2 - s^2) \\ a_{12}s + a_{13}c & (a_{22} - a_{33})sc + a_{23}(c^2 - s^2) & a_{22}s^2 + a_{33}c^2 + 2a_{23}sc \end{pmatrix}.$$

Note that a_{11} is unchanged! As it should.

Jacobi's method, an example to convice you about the algorithm

We have

$$\mathbf{B} = \begin{pmatrix} a_{11} & a_{12}c - a_{13}s & a_{12}s + a_{13}c \\ a_{12}c - a_{13}s & a_{22}c^2 + a_{33}s^2 - 2a_{23}sc & (a_{22} - a_{33})sc + a_{23}(c^2 - s^2) \\ a_{12}s + a_{13}c & (a_{22} - a_{33})sc + a_{23}(c^2 - s^2) & a_{22}s^2 + a_{33}c^2 + 2a_{23}sc \end{pmatrix}.$$

or

$$\begin{aligned} b_{11} &= a_{11} \\ b_{12} &= a_{12}\cos\theta - a_{13}\sin\theta, 1 \neq 2, 1 \neq 3 \\ b_{13} &= a_{13}\cos\theta + a_{12}\sin\theta, 1 \neq 2, 1 \neq 3 \\ b_{22} &= a_{22}\cos^2\theta - 2a_{23}\cos\theta\sin\theta + a_{33}\sin^2\theta \\ b_{33} &= a_{33}\cos^2\theta + 2a_{23}\cos\theta\sin\theta + a_{22}\sin^2\theta \\ b_{23} &= (a_{22} - a_{33})\cos\theta\sin\theta + a_{23}(\cos^2\theta - \sin^2\theta) \end{aligned}$$

We will fix the angle θ so that $b_{23} = 0$.

Jacobi's method, an example to convice you about the algorithm

We get then a new matrix

$$\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & 0 \\ b_{13} & 0 & a_{33} \end{pmatrix}.$$

We repeat then assuming that b_{12} is the largest non-diagonal matrix element and get a new matrix

$$\mathbf{C} = \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & 0 \\ b_{13} & 0 & b_{33} \end{pmatrix} \begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

We continue this process till all non-diagonal matrix elements are zero (ideally). You will notice that performing the above operations that the matrix element b_{23} which was previous zero becomes different from zero. This is one of the problems which slows down the jacobi procedure.

Jacobi's method, an example to convice you about the algorithm

The more general expression for the new matrix elements are

$$
\begin{aligned}
b_{ii} &= a_{ii}, i \neq k, i \neq l \\
b_{ik} &= a_{ik}\cos\theta - a_{il}\sin\theta, i \neq k, i \neq l \\
b_{il} &= a_{il}\cos\theta + a_{ik}\sin\theta, i \neq k, i \neq l \\
b_{kk} &= a_{kk}\cos^2\theta - 2a_{kl}\cos\theta\sin\theta + a_{ll}\sin^2\theta \\
b_{ll} &= a_{ll}\cos^2\theta + 2a_{kl}\cos\theta\sin\theta + a_{kk}\sin^2\theta \\
b_{kl} &= (a_{kk} - a_{ll})\cos\theta\sin\theta + a_{kl}(\cos^2\theta - \sin^2\theta)
\end{aligned}
$$

This is what we will need to code.

Jacobi code example

Main part

```
//  we have defined a matrix A and a matrix R for
    the eigenvector, both of dim n x n
//  The final matrix R has the eigenvectors in its
    row elements, it is set to one
//  for the diagonal elements in the beginning,
    zero else.
....
double tolerance = 1.0E-10;
int iterations = 0;
while ( maxnondiag > tolerance && iterations <=
    maxiter)
{
   int p, q;
   maxnondiag  = offdiag(A, p, q, n);
   Jacobi_rotate(A, R, p, q, n);
   iterations++;
}
...
```

Jacobi code example

Finding the max nondiagonal element

```
//  the offdiag function
double offdiag(double **A, int p, int q, int n);
{
   double max;
   for (int i = 0; i < n; ++i)
   {
       for ( int j = i+1; j < n; ++j)
       {
           double aij = fabs(A[i][j]);
           if ( aij > max)
           {
              max = aij;  p = i; q = j;
           }
       }
   }
   return max;
}
...
```

Jacobi code example

Finding the new matrix elements

```
void Jacobi_rotate ( double ** A, double ** R, int
    k, int l, int n )
{
  double s, c;
  if ( A[k][l] != 0.0 ) {
    double t, tau;
    tau = (A[l][l] - A[k][k])/(2*A[k][l]);

    if ( tau >= 0 ) {
      t = 1.0/(tau + sqrt(1.0 + tau*tau));
    } else {
      t = -1.0/(-tau +sqrt(1.0 + tau*tau));
    }

    c = 1/sqrt(1+t*t);
    s = c*t;
  } else {
    c = 1.0;
    s = 0.0;
  }
```

Jacobi code example

```
double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
a_kk = A[k][k];
a_ll = A[l][l];
A[k][k] = c*c*a_kk - 2.0*c*s*A[k][l] + s*s*a_ll;
A[l][l] = s*s*a_kk + 2.0*c*s*A[k][l] + c*c*a_ll;
A[k][l] = 0.0;  // hard-coding non-diagonal
    elements by hand
A[l][k] = 0.0;  // same here
for ( int i = 0; i < n; i++ ) {
  if ( i != k && i != l ) {
    a_ik = A[i][k];
    a_il = A[i][l];
    A[i][k] = c*a_ik - s*a_il;
    A[k][i] = A[i][k];
    A[i][l] = c*a_il + s*a_ik;
    A[l][i] = A[i][l];
  }
```

Jacobi code example

and finally the new eigenvectors

```
      r_ik = R[i][k];
      r_il = R[i][l];

      R[i][k] = c*r_ik - s*r_il;
      R[i][l] = c*r_il + s*r_ik;
    }
    return;
} // end of function jacobi_rotate
```

Week 38

Eigenvalue solvers and classes

- Monday: Repetition from last week and discussion of project 2
- Jacobi's and Householder's algorithms
- Wednesday:
- Householder's algorithm and Francis' algorithm
- How to construct a vector-matrix class

Eigenvalue Solvers, Householder

The first step consists in finding an orthogonal matrix $\mathbf{S}$ which is the product of $(n-2)$ orthogonal matrices

$$\mathbf{S} = \mathbf{S}_1\mathbf{S}_2 \dots \mathbf{S}_{n-2},$$

each of which successively transforms one row and one column of $\mathbf{A}$ into the required tridiagonal form. Only $n-2$ transformations are required, since the last two elements are already in tridiagonal form. In order to determine each $\mathbf{S_i}$ let us see what happens after the first multiplication, namely,

$$\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & a'_{23} & \dots & \dots & \dots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & \dots & \dots & \dots & a'_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & a'_{n2} & a'_{n3} & \dots & \dots & \dots & a'_{nn} \end{pmatrix}$$

where the primed quantities represent a matrix $\mathbf{A}'$ of dimension $n-1$ which will subsequentely be transformed by $\mathbf{S_2}$.

Eigenvalue Solvers, Householder

The factor e_1 is a possibly non-vanishing element. The next transformation produced by $\mathbf{S_2}$ has the same effect as $\mathbf{S_1}$ but now on the submatirx $\mathbf{A}^{'}$ only

$$
(\mathbf{S}_1\mathbf{S}_2)^T\,\mathbf{A}\mathbf{S}_1\mathbf{S}_2 = \begin{pmatrix}
a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\
e_1 & a'_{22} & e_2 & 0 & \dots & \dots & 0 \\
0 & e_2 & a''_{33} & \dots & \dots & \dots & a''_{3n} \\
0 & \dots & \dots & \dots & \dots & \dots & \\
0 & 0 & a''_{n3} & \dots & \dots & \dots & a''_{nn}
\end{pmatrix}
$$

Note that the effective size of the matrix on which we apply the transformation reduces for every new step. In the previous Jacobi method each similarity transformation is in principle performed on the full size of the original matrix.

Eigenvalue Solvers, Householder

After a series of such transformations, we end with a set of diagonal matrix elements

$$a_{11}, a'_{22}, a''_{33} \dots a_{nn}^{n-1},$$

and off-diagonal matrix elements

$$e_1, e_2, e_3, \dots, e_{n-1}.$$

The resulting matrix reads

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \left(\begin{array}{ccccccc} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & a''_{33} & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & a_{n-2}^{(n-1)} & e_{n-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n-1} & a_{n-1}^{(n-1)} \end{array} \right).$$

Eigenvalue Solvers, Householder

It remains to find a recipe for determining the transformation $\mathbf{S}_n$. We illustrate the method for $\mathbf{S}_1$ which we assume takes the form

$$\mathbf{S_1} = \begin{pmatrix} 1 & \mathbf{0^T} \\ \mathbf{0} & \mathbf{P} \end{pmatrix},$$

with $\mathbf{0^T}$ being a zero row vector, $\mathbf{0^T} = \{0, 0, \cdots\}$ of dimension $(n-1)$. The matrix $\mathbf{P}$ is symmetric with dimension $((n-1) \times (n-1))$ satisfying $\mathbf{P}^2 = \mathbf{I}$ and $\mathbf{P}^T = \mathbf{P}$. A possible choice which fullfils the latter two requirements is

$$\mathbf{P} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^T,$$

where $\mathbf{I}$ is the $(n-1)$ unity matrix and $\mathbf{u}$ is an $n-1$ column vector with norm $\mathbf{u}^T\mathbf{u}$ (inner product).

Eigenvalue Solvers, Householder

Note that $\mathbf{u}\mathbf{u}^T$ is an outer product giving a matrix of dimension $((n-1)\times(n-1))$. Each matrix element of $\mathbf{P}$ then reads

$$P_{ij} = \delta_{ij} - 2u_iu_j,$$

where i and j range from 1 to $n-1$. Applying the transformation $\mathbf{S}_1$ results in

$$\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 = \begin{pmatrix} a_{11} & (\mathbf{P}\mathbf{v})^T \\ \mathbf{P}\mathbf{v} & \mathbf{A}' \end{pmatrix},$$

where $\mathbf{v}^{\mathbf{T}} = \{a_{21}, a_{31}, \cdots, a_{n1}\}$ and $\mathbf{P}$ must satisfy $(\mathbf{P}\mathbf{v})^T = \{k, 0, 0, \cdots\}$. Then

$$\mathbf{P}\mathbf{v} = \mathbf{v} - 2\mathbf{u}(\mathbf{u}^T\mathbf{v}) = k\mathbf{e}, \tag{39}$$

with $\mathbf{e}^{\mathbf{T}} = \{1, 0, 0, \dots 0\}$.

Eigenvalue Solvers, Householder

Solving the latter equation gives us **u** and thus the needed transformation **P**. We do first however need to compute the scalar k by taking the scalar product of the last equation with its transpose and using the fact that $\mathbf{P}^2 = \mathbf{I}$. We get then

$$(\mathbf{P}\mathbf{v})^T\mathbf{P}\mathbf{v} = k^2 = \mathbf{v}^T\mathbf{v} = |v|^2 = \sum_{i=2}^{n} a_{i1}^2,$$

which determines the constant $k = \pm v$.

Eigenvalue Solvers, Householder

Now we can rewrite Eq. (39) as

$$\mathbf{v} - k\mathbf{e} = 2\mathbf{u}(\mathbf{u}^T\mathbf{v}),$$

and taking the scalar product of this equation with itself and obtain

$$2(\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), \tag{40}$$

which finally determines

$$\mathbf{u} = \frac{\mathbf{v} - k\mathbf{e}}{2(\mathbf{u}^T\mathbf{v})}.$$

In solving Eq. (40) great care has to be exercised so as to choose those values which make the right-hand largest in order to avoid loss of numerical precision. The above steps are then repeated for every transformations till we have a tridiagonal matrix suitable for obtaining the eigenvalues.

Eigenvalue Solvers, Householder, brute force

Our Householder transformation has given us a tridiagonal matrix. We discuss here how one can use Jacobi's iterative procedure to obtain the eigenvalues. Letus specialize to a 4×4 matrix. The tridiagonal matrix takes the form

$$\mathbf{A} = \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix}.$$

As a first observation, if any of the elements e_i are zero the matrix can be separated into smaller pieces before diagonalization. Specifically, if $e_1 = 0$ then d_1 is an eigenvalue.

Eigenvalue Solvers, Householder

Thus, let us introduce a transformation $\mathbf{S_1}$ which operates like

$$\mathbf{S_1} = \begin{pmatrix} \cos\theta & 0 & 0 & \sin\theta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \cos\theta & 0 & 0 & \cos\theta \end{pmatrix}$$

Eigenvalue Solvers, Householder

Then the similarity transformation

$$\mathbf{S_1^T A S_1} = \mathbf{A}' = \left(\begin{array}{cccc} d_1' & e_1' & 0 & 0 \\ e_1' & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e'3 \\ 0 & 0 & e_3' & d_4' \end{array} \right)$$

produces a matrix where the primed elements in $\mathbf{A}'$ have been changed by the transformation whereas the unprimed elements are unchanged. If we now choose θ to give the element $a_{21}^{'} = e^{'} = 0$ then we have the first eigenvalue $= a_{11}^{'} = d_1^{'}$. (This is actually what you are doing in project 2!!)

Eigenvalue Solvers, Householder's and Francis' algorithm

This procedure can be continued on the remaining three-dimensional submatrix for the next eigenvalue. Thus after few transformations we have the wanted diagonal form. What we see here is just a special case of the more general procedure developed by Francis in two articles in 1961 and 1962. Using Jacobi's method is not very efficient ether.

The algorithm is based on the so-called **QR** method (or just **QR**-algorithm). It follows from a theorem by Schur which states that any square matrix can be written out in terms of an orthogonal matrix $\hat{Q}$ and an upper triangular matrix $\hat{U}$. Historically R was used instead of U since the wording right triangular matrix was first used.

Eigenvalue Solvers, Householder's and Francis' algorithm

The method is based on an iterative procedure similar to Jacobi's method, by a succession of planar rotations. For a tridiagonal matrix it is simple to carry out in principle, but complicated in detail!
Schur's theorem

$$\hat{A} = \hat{Q}\hat{U},$$

is used to rewrite any square matrix into a unitary matrix times an upper triangular matrix. We say that a square matrix is similar to a triangular matrix.
Householder's algorithm which we have derived is just a special case of the general Householder algorithm. For a symmetric square matrix we obtain a tridiagonal matrix.

There is a corollary to Schur's theorem which states that every Hermitian matrix is

unitarily similar to a diagonal matrix.

Eigenvalue Solvers, Householder's and Francis' algorithm

It follows that we can define a new matrix

$$\hat{A}\hat{Q} = \hat{Q}\hat{U}\hat{Q},$$

and multiply from the left with $\hat{Q}^{-1}$ we get

$$\hat{Q}^{-1}\hat{A}\hat{Q} = \hat{B} = \hat{U}\hat{Q},$$

where the matrix $\hat{B}$ is a similarity transformation of $\hat{A}$ and has the same eigenvalues as $\hat{B}$.

Eigenvalue Solvers, Householder's and Francis' algorithm

Suppose $\hat{A}$ is the triangular matrix we obtained after the Householder transformation,

$$\hat{A} = \hat{Q}\hat{U},$$

and multiply from the left with $\hat{Q}^{-1}$ resulting in

$$\hat{Q}^{-1}\hat{A} = \hat{U}.$$

Suppose that $\hat{Q}$ consists of a series of planar Jacobi like rotations acting on sub blocks of $\hat{A}$ so that all elements below the diagonal are zeroed out

$$\hat{Q} = \hat{R}_{12}\hat{R}_{23}\dots\hat{R}_{n-1,n}.$$

Eigenvalue Solvers, Householder's and Francis' algorithm

A transformation of the type $\hat{R}_{12}$ looks like

$$\hat{R}_{12} = \begin{pmatrix} c & s & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ -s & c & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & & & \\ 0 & 0 & 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix}$$

Eigenvalue Solvers, Householder's and Francis' algorithm

The matrix $\hat{U}$ takes then the form

$$\hat{U} = \begin{pmatrix} x & x & x & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & x & x & x & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & x & x & x & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & & & \\ 0 & 0 & 0 & 0 & 0 & \dots & x & x & x \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & x \end{pmatrix}$$

which has a second superdiagonal.

Eigenvalue Solvers, Householder's and Francis' algorithm

We have now found $\hat{Q}$ and $\hat{U}$ and this allows us to find the matrix $\hat{B}$ which is, due to Schur's theorem, unitarily similar to a triangular matrix (upper in our case) since we have that

$$\hat{Q}^{-1}\hat{A}\hat{Q} = \hat{B},$$

from Schur's theorem the matrix $\hat{B}$ is triangular and the eigenvalues the same as those of $\hat{A}$ and are given by the diagonal matrix elements of $\hat{B}$. Why?
Our matrix $\hat{B} = \hat{U}\hat{Q}$.

Another iterative procedure

The matrix $\hat{A}$ is transformed into a tridiagonal form and the last step is to transform it into a diagonal matrix giving the eigenvalues on the diagonal.
The eigenvalues of a matrix can be obtained using the characteristic polynomial

$$P(\lambda) = det(\lambda\mathbf{I} - \mathbf{A}) = \prod_{i=1}^{n} (\lambda_i - \lambda),$$

which rewritten in matrix form reads

$$P(\lambda) = \begin{pmatrix} d_1 - \lambda & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 - \lambda & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 - \lambda & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{N_{\mathrm{step}}-2} - \lambda & e_{N_{\mathrm{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{N_{\mathrm{step}}-1} & d_{N_{\mathrm{step}}-1} - \lambda \end{pmatrix}$$

Eigenvalue Solvers, Householder

We can solve this equation in an iterative manner. We let $P_k(\lambda)$ be the value of k subdeterminant of the above matrix of dimension $n \times n$. The polynomial $P_k(\lambda)$ is clearly a polynomial of degree k. Starting with $P_1(\lambda)$ we have $P_1(\lambda) = d_1 - \lambda$. The next polynomial reads $P_2(\lambda) = (d_2 - \lambda)P_1(\lambda) - e_1^2$. By expanding the determinant for $P_k(\lambda)$ in terms of the minors of the nth column we arrive at the recursion relation

$$P_k(\lambda) = (d_k - \lambda)P_{k-1}(\lambda) - e_{k-1}^2 P_{k-2}(\lambda).$$

Together with the starting values $P_1(\lambda)$ and $P_2(\lambda)$ and good root searching methods we arrive at an efficient computational scheme for finding the roots of $P_n(\lambda)$. However, for large matrices this algorithm is rather inefficient and time-consuming.

Eigenvalue Solvers, Householder functions

The programs which performs these transformations are
matrix $\mathbf{A} \longrightarrow$ tridiagonal matrix $\longrightarrow$ diagonal matrix

C++:	void trd2(double $**$a, int n, double d[], double e[])
	void tqli(double d[], double[], int n, double $**$z)
Fortran:	CALL tred2(a, n, d, e)
	CALL tqli(d, e, n, z)

Eigenvalue Solvers, Power methods

We assume $\hat{A}$ can be diagonalized. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n eigenvalues (counted with multiplicity) of $\hat{A}$ and let v_1, $v2, \ldots, v_n$ be the corresponding eigenvectors. We assume that λ_1 is the dominant eigenvalue, so that $|\lambda_1| > |\lambda_j|$ for $j > 1$.
The initial vector b_0 can be written:

$$b_0 = c_1 v_1 + c_2 v_2 + \cdots + c_m v_m.$$

If b_0 is chosen randomly (with uniform probability), then c_1 ÃÂÂ 0 with probability 1.
Now,

$$\begin{aligned} A^k b_0 &= c_1 A^k v_1 + c_2 A^k v_2 + \cdots + c_m A^k v_m \\ &= c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + \cdots + c_m \lambda_m^k v_m \\ &= c_1 \lambda_1^k \left(v_1 + \frac{c_2}{c_1} \left(\frac{\lambda_2}{\lambda_1} \right)^k v_2 + \cdots + \frac{c_m}{c_1} \left(\frac{\lambda_m}{\lambda_1} \right)^k v_m \right). \end{aligned}$$

The expression within parentheses converges to v_1 because $|\lambda_j/\lambda_1| < 1$ for $j > 1$. On the other hand, we have

$$b_k = \frac{A^k b_0}{\|A^k b_0\|}.$$

Therefore, b_k converges to (a multiple of) the eigenvector v_1. The convergence is geometric, with ratio

$$\left| \frac{\lambda_2}{\lambda_1} \right|,$$

where λ_2 denotes the second dominant eigenvalue. Thus, the method converges slowly if there is an eigenvalue close in magnitude to the dominant eigenvalue.

Eigenvalue Solvers, Power methods

Under the assumptions:

- A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues
- The starting vector b_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue.

then:

- A subsequence of (b_k) converges to an eigenvector associated with the dominant eigenvalue

Note that the sequence (b_k) does not necessarily converge. It can be shown that $b_k = e^{i\phi_k} v_1 + r_k$ where: v_1 is an eigenvector associated with the dominant eigenvalue, and $\|r_k\| \to 0$. The presence of the term $e^{i\phi_k}$ implies that (b_k) does not converge unless $e^{i\phi_k} = 1$. Under the two assumptions listed above, the sequence (μ_k) defined by $\mu_k = \frac{b_k^* A b_k}{b_k^* b_k}$ converges to the dominant eigenvalue.

Eigenvalue Solvers, Power methods

Power iteration is not used very much because it can find only the dominant eigenvalue. The algorithm is however very useful in some specific case. For instance, Google uses it to calculate the page rank of documents in their search engine. For matrices that are well-conditioned and as sparse as the web matrix, the power iteration method can be more efficient than other methods of finding the dominant eigenvector.

Some of the more advanced eigenvalue algorithms can be understood as variations of the power iteration. For instance, the inverse iteration method applies power iteration to the matrix $\hat{A}^{-1}$. Other algorithms look at the whole subspace generated by the vectors b_k. This subspace is known as the Krylov subspace. It can be computed by Arnoldi iteration or Lanczos iteration. The latter is method of choice for diagonalizing symmetric matrices with huge dimensionalities.

Using Lapack to solve linear algebra and eigenvalue problems

Suppose you wanted to solve a general system of linear equations $\hat{A}\mathbf{x} = \mathbf{b}$, where $\hat{A}$is an $n \times n$ square matrix and x and b are n-element column vectors. You opt to use the routine **dgesv**. The man page abstract obtained with

```
man dgesv
```

if Lapack is properly installed. The C++ declaration is

```
void dgesv(int n, int  nrhs,  double  *da,  int  lda,  int *ipivot, d
```

Lapack example

```
#include <iostream>
#define MAX 10
using namespace std;
int main(){
    // Values needed for dgesv
    int n;
    int nrhs = 1;
    double a[MAX][MAX];
    double b[1][MAX];
    int lda = MAX;
    int ldb = MAX;
    int ipiv[MAX];
    int info;
    // Other values
    int i,j;
}
```

Lapack example

```
    // Read the values of the matrix
    cout << "Enter n \n";
    cin >> n;
    cout << "On each line type a row of the matrix A
        followed by one element of b:\n";
    for(i = 0; i < n; i++){
      cout << "row " << i << " ";
      for(j = 0; j < n; j++)std::cin >> a[j][i];
      cin >> b[0][i];
    }
}
```

Lapack example

```
    // Solve the linear system
    dgesv(n, nrhs, &a[0][0], lda, ipiv, &b[0][0],
        ldb, &info);
    // Check for success
    if(info == 0)
    {
        // Write the answer
        cout << "The answer is\n";
        for(i = 0; i < n; i++)
          cout << "b[" << i << "]\t" << b[0][i] << "\
              n";
    }
    else
    {
        // Write an error message
        cerr << "dgesv returned error " << info << "\
            n";
    }
    return info;
}
```

Programming classes

In Fortran a vector or matrix start with 1, but it is easy to change a vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at -10 and ends at 10, we could declare it as REAL(KIND=8):: vector(-10:10). Similarly, if we want to start at zero and end at 10 we could write REAL(KIND=8):: vector(0:10). We have also seen that Fortran allows us to write a matrix addition $\mathbf{A} = \mathbf{B} + \mathbf{C}$ as A = B + C. This means that we have overloaded the addition operator so that it translates this operation into two loops and an addition of two matrix elements $a_{ij} = b_{ij} + c_{ij}$.

Programming classes

The way the matrix addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.

In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two matrices we need to explicitely write out the two loops as

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

Programming classes

However, the strength of C++ over programming languages like C and Fortran 77 is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ matrix elements a_{ij}, from $a[i][j]$ to say $a(i,j)$, as we would do in Fortran. Similarly, we could also change the default range from $0 : n-1$ to $1 : n$.

To achieve this we need to introduce two important entities in C++ programming,

classes and templates.

Programming classes

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Programming classes

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.
A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

Programming classes

C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

```
// Program to calculate addition and multiplication
     of two complex numbers
using namespace std;
#include <iostream>
#include <cmath>
#include <complex>
int main()
{
  complex<double> x(6.1,8.2), y(0.5,1.3);
  // write out x+y
  cout << x + y << x*y  << endl;
  return 0;
}
```

where we add and multiply two complex numbers $x = 6.1 + \imath 8.2$ and $y = 0.5 + \imath 1.3$ with the obvious results $z = x + y = 6.6 + \imath 9.5$ and $z = x \cdot y = -7.61 + \imath 12.03$.

Programming classes

We proceed by splitting our task in three files.
We define first a header file complex.h which contains the declarations of the class.
The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

```
#ifndef Complex_H
#define Complex_H
//   various include statements and definitions
#include <iostream>            // Standard ANSI-C++
   include files
#include <new>
#include ....

class Complex
{...
definition of variables and their character
};
//   declarations of various functions used by the
   class
...
#endif
```

Programming classes

Next we provide a file complex.cpp where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files complex.h and complex.cpp are normally placed in a directory with other classes and libraries we have defined.

Finally,we discuss here an example of a main program which uses this particular class.

An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

Programming classes

```
#include "Complex.h"
... other include and declarations
int main ()
{
  Complex a(0.1,1.3);     // we declare a complex
      variable a
  Complex b(3.0), c(5.0,-2.3);  // we declare
      complex variables b and c
  Complex d = b;          //  we declare  a new
      complex variable d
  cout << "d=" << d << ", a=" << a << ", b=" << b
      << endl;
  d = a*c + b/a;  //   we add, multiply and divide
      two complex numbers
  cout << "Re(d)=" << d.Re() << ", Im(d)=" << d.Im
      () << endl;  // write out of the real and
      imaginary parts
}
```

Programming classes

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + \imath 1.3$, $b = 3.0 + \imath 0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - \imath 2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

Programming classes

```
class Complex
{
private:
   double re, im; // real and imaginary part
public:
   Complex (); //
      Complex c;
   Complex (double re, double im = 0.0); //
      Definition of a complex variable;
   Complex (const Complex& c); //
      Usage: Complex c(a);   // equate two complex
       variables
   Complex& operator= (const Complex& c); // c = a;
        //  equate two complex variables, same as
       previous
....
```

Programming classes

```
    ~Complex () {}                              //
       destructor
    double   Re () const;            // double real_part
         = a.Re();
    double   Im () const;            // double imag_part
         = a.Im();
    double   abs () const;           // double m = a.abs
        (); // modulus
    friend Complex operator+ (const Complex&  a,
        const Complex& b);
    friend Complex operator- (const Complex&  a,
        const Complex& b);
    friend Complex operator* (const Complex&  a,
        const Complex& b);
    friend Complex operator/ (const Complex&  a,
        const Complex& b);
};
```

Programming classes

The class is defined via the statement **class** Complex. We must first use the key word **class**, which in turn is followed by the user-defined variable name Complex. The body of the class, data and functions, is encapsulated within the parentheses {...};.

Programming classes

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use **protected** instead of **private**, then data and functions can be inherited outside the class.

Programming classes

The key word **public** means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration **friend** means that stand-alone functions can work on privately declared variables of the type `(re, im)`. Data members of a class should be declared as private variables.

Programming classes

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type Complex and how this variable is initialized. We have chose three possibilities in the example above:

- A declaration like Complex c; calls the member function Complex() which can have the following implementation

  ```
  Complex::Complex() { re = im = 0.0; }
  ```

  meaning that it sets the real and imaginary parts to zero. Note the way a member function is defined. The constructor is the first function that is called when an object is instantiated.

Programming classes

- Another possibility is

```
Complex::Complex()    {}
```

  which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.
- A call like Complex a(0.1,1.3); means that we could call the member function Complex(**double**, **double**)as

```
Complex::Complex(double re_a, double im_a)
{ re = re_a; im = im_a; }
```

Programming classes

The simplest member function are those we defined to extract the real and imaginary part of a variable. Here you have to recall that these are private data, that is they invisible for users of the class. We obtain a copy of these variables by defining the functions

```
double Complex:: Re () const { return re; }} // getting the real part
double Complex:: Im () const { return im; }  // and the imaginary part
```

Note that we have introduced the declaration **const.** What does it mean? This declaration means that a variabale cannot be changed within a called function.

Programming classes

If we define a variable as **const double** p = 3; and then try to change its value, we will get an error when we compile our program. This means that constant arguments in functions cannot be changed.

```
// const arguments (in functions) cannot be changed
   :
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */ }
```

If we declare the function and try to change the value to 0.2, the compiler will complain by sending an error message.

Programming classes

If we define a function to compute the absolute value of complex variable like

```
double Complex:: abs ()  { return sqrt(re*re + im*
    im);}
```

without the constant declaration and define thereafter a function myabs as

```
double myabs (const Complex& c)
{ return c.abs(); }   // Not ok because c.abs() is
    not a const func.
```

the compiler would not allow the c.abs() call in myabs since Complex::abs is not a constant member function.

Programming classes

Constant functions cannot change the object's state. To avoid this we declare the function abs as

```
double Complex::abs() const { return sqrt(re*re +
    im*im); }
```

Programming classes

C++ (and Fortran) allow for overloading of operators. That means we can define algebraic operations on for example vectors or any arbitrary object. As an example, a vector addition of the type $\mathbf{c} = \mathbf{a} + \mathbf{b}$ means that we need to write a small part of code with a for-loop over the dimension of the array. We would rather like to write this statement as `c = a+b;` as this makes the code much more readable and close to eventual equations we want to code. To achieve this we need to extend the definition of operators.

Programming classes

Let us study the declarations in our complex class. In our main function we have a statement like d = b;, which means that we call d.**operator**= (b) and we have defined a so-called assignment operator as a part of the class defined as

```
Complex& Complex:: operator= (const Complex& c)
{
   re = c.re;
   im = c.im;
   return *this;
}
```

Programming classes

With this function, statements like Complex d = b; or Complex d(b); make a new object *d*, which becomes a copy of *b*. We can make simple implementations in terms of the assignment

```
Complex :: Complex (const Complex& c)
{ *this = c; }
```

which is a pointer to "this object", ∗**this** is the present object, so ∗**this** = c; means setting the present object equal to *c*, that is **this**−>**operator**= (c);.

Programming classes

The meaning of the addition operator $+$ for Complex objects is defined in the function

Complex **operator**+ (**const** Complex& a, **const** Complex& b); *//a+b*

The compiler translates `c = a + b;` into `c = operator+ (a, b);`.

Since this implies the call to function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop.

Programming classes

The solution to this is to inline functions. We discussed inlining in chapter 2 of the lecture notes. Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

```
inline Complex operator+ (const Complex& a, const
    Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Inline functions, with complete bodies must be written in the header file complex.h.

Programming classes

Consider the case c = a + b; that is,
c.**operator**= (**operator**+ (a,b)); If **operator**+, **operator**= and the constructor Complex(r,i) all are inline functions, this transforms to

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

by the compiler, i.e., no function calls

Programming classes

The stand-alone function **operator+** is a friend of the Complex class

```
class Complex
{
    ...
    friend Complex operator+ (const Complex& a,
        const Complex& b);
    ...
};
```

so it can read (and manipulate) the private data parts *re* and *im* via

```
inline Complex operator+ (const Complex& a, const
    Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Programming classes

Since we do not need to alter the re and im variables, we can get the values by Re() and Im(), and there is no need to be a friend function

```
inline Complex operator+ (const Complex& a, const
    Complex& b)
{ return Complex (a.Re() + b.Re(), a.Im() + b.Im())
    ; }
```

Programming classes

The multiplication functionality can now be extended to imaginary numbers by the following code

```
inline Complex operator* (const Complex& a, const
    Complex& b)
{
  return Complex(a.re*b.re - a.im*b.im, a.im*b.re +
      a.re*b.im);
}
```

It will be convenient to inline all functions used by this operator.

Programming classes

To inline the complete expression a∗b;, the constructors and **operator**= must also be inlined. This can be achieved via the following piece of code

```
inline Complex::Complex() { re = im = 0.0; }
inline Complex::Complex(double re_, double im_)
{ ... }
inline Complex::Complex(const Complex& c)
{ ... }
inline Complex::operator=(const Complex& c)
{ ... }
```

Programming classes

```
// e, c, d are complex
e = c*d;
// first compiler translation:
e.operator= (operator* (c,d));
// result of nested inline functions
// operator=, operator*, Complex(double,double=0):
e.re = c.re*d.re - c.im*d.im;
e.im = c.im*d.re + c.re*d.im;
```

The definitions **operator**− and **operator**/ follow the same set up.

Programming classes

Finally, if we wish to write to file or another device a complex number using the simple syntax cout << c;, we obtain this by defining the effect of << for a Complex object as

```
ostream& operator<< (ostream& o, const Complex& c)
{ o << "(" << c.Re() << "," << c.Im() << ") ";
   return o;}
```

Programming classes, templates

What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace **double** with for example **int**.
C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword **template** followed by a list of template arguments enclosed in brackets.

Programming classes

We can therefore make a more general class by rewriting our original example as

```
template<class T>
class Complex
{
private:
    T re, im; // real and imaginary part
public:
    Complex (); //
        Complex c;
    Complex (T re, T im = 0); // Definition of a
        complex variable;
    Complex (const Complex& c); //
        Usage: Complex c(a);    // equate two complex
         variables
    Complex& operator= (const Complex& c); // c = a;
          //  equate two complex variables, same as
         previous
```

Programming classes

We can therefore make a more general class by rewriting our original example as

```
    ~Complex () {}                              //
       destructor
     T    Re () const;            // T real_part = a.Re();
     T    Im () const;            // T imag_part = a.Im();
     T    abs () const;           // T m = a.abs(); //
        modulus
     friend Complex operator+ (const Complex&  a,
        const Complex& b);
     friend Complex operator- (const Complex&  a,
        const Complex& b);
     friend Complex operator* (const Complex&  a,
        const Complex& b);
     friend Complex operator/ (const Complex&  a,
        const Complex& b);
};
```

Programming classes

What it says is that Complex is a parameterized type with T as a parameter and T has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

```
Complex<double> a(10.0,5.1);
Complex<int> b(1,0);
```

Programming classes

Member functions of our class are defined by preceding the name of the function with the **template** keyword. Consider the function we defined as
Complex:: Complex (**double** re_a, **double** im_a). We would rewrite this function as

```
template<class T>
Complex<T>:: Complex (T re_a, T im_a)
{ re = re_a; im = im_a; }
```

The member functions are otherwise defined following ordinary member function definitions.

Programming classes

Here follows a very simple first class

```
// Class to compute the square of a number
class Squared{
  public:
    // Default constructor, not used here
    Squared(){}

    // Overload the function operator()
    double operator()(double x){
      return x*x;
    }
};
```

Programming classes

and we would use it as

```
#include <iostream>
using namespace std;

int main(){
  Squared s;
  cout << s(3) << endl;
}
```

What do we have then?

- Several functions under the program link, see the file OOcodes.tar.gz
- Matrix and array manipulations (similar to Blitz++)
- Random numbers and numerical integration
- Functions to convert arrays from C++ to Numpy and viceversa
- Numerical derivatives and differential equation solvers.

These files can serve as a help when you want to start to write your own classes. We will discuss some of these classes during the course.

But what should I do else????? Some hints.

If your needs (common in most problems) include handling of large arrays and linear algebra problem, I would not recommend to write your own vector-matrix or more general array handling class. It is easyto make error.

- Old-fashioned allocation of arrays and explicit handling of all loops in for example matrix-matrix multiplication.
- You can use what we have developed for Blitz++ or the Array class (but more limited than Blitz++)
- or (recommended) you can use armadillo, a great C++ library for handling arrays and doing linear algebra.
- Armadillo provides a user friendly interface to lapack and blas functions. Here an example of using the Blas function **DGEMM** for matrix-matrix multiplication.
- After having installed armadillo, compile with **c++ -O3 -o test.x test.cpp -lblas**.

Matrix-matrix multiplication

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

/*Because fortran files don't have any header files
    ,
 *we need to declare the functions ourself.*/
extern "C"
{
    void dgemm_(char*, char*, int*, int*, int*,
        double*,
            double*, int*, double*, int*, double*,
                double*, int*);
}
```

Matrix-matrix multiplication

```
int main(int argc, char** argv)
{
    //Dimensions
    int n = atoi(argv[1]);
    int m = n;
    int p = m;

    /*Create random matrices
     * (note that older versions of armadillo uses
         "rand" instead of "randu") */
    srand(time(NULL));
    mat A(n, p);
    A.randu();
```

Matrix-matrix multiplication

```
// Pretty print, and pretty save, are as easy
   as the two following lines.
//     cout << A << endl;
//     A.save("A.mat", raw_ascii);
mat A_trans = trans(A);
mat B(p, m);
B.randu();
mat C(n, m);
//     cout << B << endl;
//     B.save("B.mat", raw_ascii);
```

Matrix-matrix multiplication

```
//    ARMADILLO   TEST
cout << "Starting armadillo multiplication\n";
//Simple wall_clock timer is a part of
    armadillo.
wall_clock timer;
timer.tic();
C = A*B;
double num_sec = timer.toc();
cout << "-- Finished in " << num_sec << "
    seconds.\n\n";
```

Matrix-matrix multiplication

```
C = zeros<mat> (n, m);
cout << "Starting blas multiplication.\n";
{
    char trans = 'N';
    double alpha = 1.0;
    double beta = 0.0;
    int _numRowA = A.n_rows;
    int _numColA = A.n_cols;
    int _numRowB = B.n_rows;
    int _numColB = B.n_cols;
    int _numRowC = C.n_rows;
    int _numColC = C.n_cols;
    int lda = (A.n_rows >= A.n_cols) ? A.n_rows
        : A.n_cols;
    int ldb = (B.n_rows >= B.n_cols) ? B.n_rows
        : B.n_cols;
    int ldc = (C.n_rows >= C.n_cols) ? C.n_rows
        : C.n_cols;
```

Matrix-matrix multiplication, calling DGEMM

```
        dgemm_(&trans, &trans, &_numRowA, &_numColB
            , &_numColA, &alpha,
                A.memptr(), &lda, B.memptr(), &ldb,
                    &beta, C.memptr(), &ldc);
}
```

Week 39

Numerical integration

- Monday: Brief repetition from last week
- More on classes and templates
- Numerical integration, from the trapezoidal rule to Gaussian quadrature (chapter 4)
- Adaptive methods
- Wednesday:
- Gaussian quadrature
- Examples of multidimensional integrals
- and perhaps start discussion of parallelization (chapter 4.7)

Equal Step Methods

Generalities

- Choose a step size
$$h = \frac{b-a}{N}$$
where N is the number of steps and a and b the lower and upper limits of integration.
- Choose then to stop the Taylor expansion of the function $f(x)$ at a certain derivative.
- With these approximations to $f(x)$ perform the integration.

$$\int_a^b f(x)dx = \int_a^{a+2h} f(x)dx + \int_{a+2h}^{a+4h} f(x)dx + \dots \int_{b-2h}^{b} f(x)dx.$$

The strategy then is to find a reliable Taylor expansion for $f(x)$ in the smaller sub intervals. Consider e.g., evaluating $\int_{-h}^{+h} f(x)dx$

Equal Step Methods

Trapezoidal Rule

Taylor expansion

$$f(x) = f_0 + \frac{f_h - f_0}{h} x + O(x^2),$$

for $x = x_0$ to $x = x_0 + h$ and

$$f(x) = f_0 + \frac{f_0 - f_{-h}}{h} x + O(x^2),$$

for $x = x_0 - h$ to $x = x_0$. The error goes like $O(x^2)$. If we then evaluate the integral we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{2}\left(f_h + 2f_0 + f_{-h}\right) + O(h^3),$$

which is the well-known trapezoidal rule. Local error $O(h^3) = O((b-a)^3/N^3)$, and the *global error* goes like $\approx O(h^2)$.

Equal Step Methods

Trapezoidal Rule

Easy to implement numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$ and multiply with $h/2$
- Perform a loop over $n = 1$ to $n-1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $f(a+h) + f(a+2h) + f(a+3h) + \cdots + f(b-h)$. Each step in the loop corresponds to a given value $a+nh$.
- Multiply the final result by h and add $hf(a)/2$ and $hf(b)/2$.

Trapezoidal Rule

```
double trapezoidal_rule(double a, double b, int n,
                        double (*func)(double))
{
      double trapez_sum;
      double fa, fb, x, step;
      int    j;
      step=(b-a)/((double) n);
      fa=(*func)(a)/2. ;
      fb=(*func)(b)/2. ;
      trapez_sum=0.;
      for (j=1; j <= n-1; j++){
         x=j*step+a;
         trapez_sum+=(*func)(x);
      }
      trapez_sum=(trapez_sum+fb+fa)*step;
      return trapez_sum;
}  // end function for trapezoidal rule
```

Trapezoidal Rule

Pay attention to the way we transfer the name of a function. This gives us the possibility to define a general trapezoidal method, where we give as input the name of the function.

```
double trapezoidal_rule(double a, double b, int n,
                        double (*func)(double))
```

We call this function simply as something like this

```
integral = trapezoidal_rule(a, b, n,
   mysuperduperfunction);
```

Equal Step Methods

Simpson

The first and second derivatives are given by

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j},$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j},$$

results in $f(x) = f_0 + \frac{f_h - f_{-h}}{2h} x + \frac{f_h - 2f_0 + f_{-h}}{h^2} x^2 + O(x^3)$. Inserting this formula in the integral

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}\left(f_h + 4f_0 + f_{-h}\right) + O(h^5),$$

which is Simpson's rule.

Equal Step Methods

Simpson's rule

Note that the improved accuracy in the evaluation of the derivatives gives a better error approximation, $O(h^5)$ vs. $O(h^3)$. But this is just the *local error approximation*. Using Simpson's rule we arrive at the composite rule

$$I = \int_a^b f(x)dx = \frac{h}{3}\left(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f_b\right),$$

with a global error which goes like $O(h^4)$. Algo

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$
- Perform a loop over $n = 1$ to $n-1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $4f(a+h) + 2f(a+2h) + 4f(a+3h) + \cdots + 4f(b-h)$. Odd values of n give 4 as factor while even values yield 2 as factor.
- Multiply the final result by $\frac{h}{3}$.

Equal Step Methods

The basic idea behind all integration methods is to approximate the integral

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i),$$

where ω and x are the weights and the chosen mesh points, respectively. Simpson's rule gives

$$\omega : \{h/3, 4h/3, 2h/3, 4h/3, \dots, 4h/3, h/3\},$$

for the weights, while the trapezoidal rule resulted in

$$\omega : \{h/2, h, h, \dots, h, h/2\}.$$

In general, an integration formula which is based on a Taylor series using N points, will integrate exactly a polynomial P of degree $N-1$. That is, the N weights ω_n can be chosen to satisfy N linear equations

Equal Step Methods, Polynomials and Newton-Cotes

Given $n+1$ distinct points $x_0, \ldots, x_n \in [a, b]$ and $n+1$ values $y_0, \ldots, y_n$ there exists a unique polynomial p_n with the property

$$p_n(x_j) = y_j \quad j = 0, \ldots, n$$

In the Lagrange representation this interpolation polynomial is given by

$$p_n = \sum_{k=0}^{n} l_k y_k,$$

with the Lagrange factors

$$l_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} \quad k = 0, \ldots, n$$

Example: $n = 1$

$$p_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0} x - \frac{y_1 x_0 + y_0 x_1}{x_1 - x_0},$$

which we recognize as the equation for a straight line.

Equal Step Methods, Polynomials and Newton-Cotes

The polynomial interpolatory quadrature of order n with equidistant quadrature points $x_k = a + kh$ and step $h = (b-a)/n$ is called the Newton-Cotes quadrature formula of order n. The integral is

$$\int_a^b f(x)dx \approx \int_a^b p_n(x)dx = \sum_{k=0}^{n} w_k f(x_k)$$

with

$$w_k = h\frac{(-1)^{n-k}}{k!(n-k)!}\int_0^n \prod_{\substack{j=0\\ j\neq k}}^{n} (z-j)dz,$$

for $k = 0, \ldots, n$.

Equal Step Methods, Polynomials and Newton-Cotes

The local error for the trapezoidal rule is

$$\int_a^b f(x)dx - \frac{b-a}{2}\left[f(a) + f(b)\right] = -\frac{h^3}{12}f^{(2)}(\xi),$$

and the global error (composite formula)

$$\int_a^b f(x)dx - T_h(f) = -\frac{b-a}{12}h^2 f^{(2)}(\xi).$$

For Simpson's rule we have

$$\int_a^b f(x)dx - \frac{b-a}{6}\left[f(a) + 4f((a+b)/2) + f(b)\right] = -\frac{h^5}{90}f^{(4)}(\xi),$$

and the global error

$$\int_a^b f(x)dx - S_h(f) = -\frac{b-a}{180}h^4 f^{(4)}(\xi).$$

with $\xi \in [a, b]$.

Gaussian Quadrature

- Methods based on Taylor series using $n+1$ points will integrate exactly a polynomial P of degree n. If a function $f(x)$ can be approximated with a polynomial of degree n

$$f(x) \approx P_n(x),$$

with $n+1$ mesh points we should be able to integrate exactly the polynomial P_n.

- Gaussian quadrature methods promise more than this. We can get a better polynomial approximation with order greater than $n+1$ to $f(x)$ and still get away with only $n+1$ mesh points. More precisely, we approximate

$$f(x) \approx P_{2n+1}(x),$$

and with only $n+1$ mesh points these methods promise that

$$\int f(x)dx \approx \int P_{2n+1}(x)dx = \sum_{i=0}^{n} P_{2n+1}(x_i)\omega_i,$$

Gaussian Quadrature

What we have done till now is called Newton-Cotes quadrature. The numerical approximation goes like $O(h^n)$, where n is method-dependent.
A greater precision for a given amount of numerical work can be achieved if we are willing to give up the requirement of equally spaced integration points. In Gaussian quadrature (hereafter GQ), both the mesh points and the weights are to be determined. The points will not be equally spaced The theory behind GQ is to obtain an arbitrary weight ω through the use of so-called orthogonal polynomials. These polynomials are orthogonal in some interval say e.g., [-1,1]. Our points x_i are chosen in some optimal sense subject only to the constraint that they should lie in this interval. Together with the weights we have then $2(n+1)$ ($n+1$ the number of points) parameters at our disposal.

Gaussian Quadrature

Even though the integrand is not smooth, we could render it smooth by extracting from it the weight function of an orthogonal polynomial, i.e., we are rewriting

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=0}^{n} \omega_i g(x_i),$$

where g is smooth and W is the weight function, which is to be associated with a given orthogonal polynomial.

Gaussian Quadrature

Weight Functions

The weight function W is non-negative in the integration interval $x \in [a, b]$ such that for any $n \geq 0$ $\int_a^b |x|^n W(x)dx$ is integrable. The naming weight function arises from the fact that it may be used to give more emphasis to one part of the interval than another.

Weight function	Interval	Polynomial
$W(x) = 1$	$x \in [a, b]$	Legendre
$W(x) = e^{-x^2}$	$-\infty \leq x \leq \infty$	Hermite
$W(x) = e^{-x}$	$0 \leq x \leq \infty$	Laguerre
$W(x) = 1/(\sqrt{1 - x^2})$	$-1 \leq x \leq 1$	Chebyshev

Legendre

$$I = \int_{-1}^{1} f(x)dx$$

$$C(1-x^2)P - m_l^2 P + (1-x^2)\frac{d}{dx}\left((1-x^2)\frac{dP}{dx}\right) = 0.$$

C is a constant. For $m_l = 0$ we obtain the Legendre polynomials as solutions, whereas $m_l \neq 0$ yields the so-called associated Legendre polynomials. The corresponding polynomials P are

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1)^k \qquad k = 0, 1, 2, \ldots,$$

which, up to a factor, are the Legendre polynomials L_k. The latter fulfil the orthorgonality relation

$$\int_{-1}^{1} L_i(x)L_j(x)dx = \frac{2}{2i+1}\delta_{ij},$$

and the recursion relation

$$(j+1)L_{j+1}(x) + jL_{j-1}(x) - (2j+1)xL_j(x) = 0.$$

Laguerre

$$I = \int_0^{\infty} f(x)dx = \int_0^{\infty} x^{\alpha} e^{-x} g(x)dx.$$

These polynomials arise from the solution of the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right)\mathcal{L}(x) = 0,$$

where l is an integer $l \geq 0$ and λ a constant. They fulfil the orthorgonality relation

$$\int_{-\infty}^{\infty} e^{-x}\mathcal{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathcal{L}_{n+1}(x) = (2n+1-x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x).$$

Hermite

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{\infty} e^{-x^2} g(x)dx.$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2 H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0.$$

They fulfil the orthorgonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

Gaussian Quadrature, general Properties

A quadrature formula

$$\int_a^b W(x)f(x)dx \approx \sum_{i=0}^{n} \omega_i f(x_i),$$

with $n+1$ distinct quadrature points (mesh points) is a called a Gaussian quadrature formula if it integrates all polynomials $p \in P_{2n+1}$ exactly, that is

$$\int_a^b W(x)p(x)dx = \sum_{i=0}^{n} \omega_i p(x_i),$$

It is assumed that $W(x)$ is continuous and positive and that the integral

$$\int_a^b W(x)dx,$$

exists. Note that the replacement of $f \rightarrow Wg$ is normally a better approximation due to the fact that we may isolate possible singularities of W and its derivatives at the endpoints of the interval.

Week 40

Parallelization and Monte Carlo methods

- Monday: Repetition from last week
- Examples of multidimensional integrals and other integration tricks
- Parallelization (chapter 4.7)
- Wednesday:
- Programming: Cython, Python and C++ by Jørgen Høgberget
- Basics of Monte Carlo methods
- Monte Carlo integration (project 3 will be presented Monday of week 41)

Final lecture notes (electronic version) on webpage of course Friday October 7. Hard-copy version toward end of next week.

A simple example

We want to compute

$$I = \int_0^\infty x \exp(-x) \sin x = \frac{1}{2},$$

using brute force Trapezoidal rule, Simpson's rule, Gauss-Legendre, Gauss-Laguerre and Gauss-Legendre again but with a smarter mapping.

- Before we start it can be useful to study the integrand.
- How should we pick the integration limits?

Simple integral

Approximate

$$\int_0^\infty f(x)dx \approx \int_0^\Lambda f(x)dx$$

```
int n;
double a, b, alf, xx;
cout << "Read in the number of integration
   points" << endl;
cin >> n;
cout << "Read in integration limits" << endl;
cin >> a >> b;
```

Simple integral

```
//    reserve space in memory for vectors containing
      the mesh points
//    weights and function values for the use of the
      gauss-legendre
//    method
      double *x = new double [n];
      double *w = new double [n];
      // Gauss-Laguerre is old-fashioned translation
           of F77 --> C++
      // arrays start at 1 and end at n
      double *xgl = new double [n+1];
      double *wgl = new double [n+1];
```

Simple integral

Note the parameter *alf* in $x^{alpha} \exp -x$

```
//    set up the mesh points and weights
      gauleg(a, b,x,w, n);
//    set up the mesh points and weights
      alf = 1.0;  // <--- Note alf
      gauss_laguerre(xgl,wgl, n, alf);
//    evaluate the integral with the Gauss-Legendre
    method
//    Note that we initialize the sum. Here brute
    force gauleg
      double int_gauss = 0.;
      for ( int i = 0;  i < n; i++){
         int_gauss+=w[i]*int_function(x[i]);
      }
```

Simple integral, importance integration/sampling

```
//    evaluate the integral with the Gauss-Laguerre
  method
//    Note that we initialize the sum
      double int_gausslag = 0.;
      for ( int i = 1;  i <= n; i++){
        int_gausslag+=wgl[i]*sin(xgl[i]);      }
```

Simple integral

```
//    evaluate the integral with the Gauss-Laguerre
   method
//    Here we change the mesh points with a mapping
//    Need to call gauleg from -1 to + 1
      gauleg(-1.0, 1.0,x,w, n);
      double pi_4 = acos(-1.0)*0.25;
      for ( int i = 0;  i < n; i++){
        xx=pi_4*(x[i]+1.0);
        r[i]= tan(xx);
        s[i]=pi_4/(cos(xx)*cos(xx))*w[i];
      }
      double int_gausslegimproved = 0.;
      for ( int i = 0;  i < n; i++){
        int_gausslegimproved += s[i]*int_function(r[
            i]);
      }
```

A six-dimensional integral, project 3 2010

The ansatz for the wave function for two electrons is given by the product of two 1*s* wave functions as

$$\Psi(\mathbf{r}_1,\mathbf{r}_2) = e^{-\alpha(r_1+r_2)}.$$

Note that it is not possible to find a closed form solution to Schrödinger's equation for two interacting electrons in the helium atom.
The integral we need to solve is the quantum mechanical expectation value of the correlation energy between two electrons, namely

$$\langle \frac{1}{|\mathbf{r}_1-\mathbf{r}_2|}\rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1+r_2)}\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|} = \frac{5\pi^2}{16^2} = 0.192765711. \quad (41)$$

Note that our wave function is not normalized. There is a normalization factor missing.

Brute force, six-dimensional integral, Gauss-Legendre

```
      double *x = new double [N];
      double *w = new double [N];
//    set up the mesh points and weights
      gauleg(a,b,x,w, N);

//    evaluate the integral with the Gauss-Legendre
    method
//    Note that we initialize the sum
      double int_gauss = 0.;
      for (int i=0;i<N;i++){
              for (int j = 0;j<N;j++){
              for (int k = 0;k<N;k++){
              for (int l = 0;l<N;l++){
              for (int m = 0;m<N;m++){
              for (int n = 0;n<N;n++){
        int_gauss+=w[i]*w[j]*w[k]*w[l]*w[m]*w[n]
       *int_function(x[i],x[j],x[k],x[l],x[m],x[n])
          ;
                 }}}}}
        }
```

The six-dimensional integral with Gauss-Legendre

```
//  this function defines the function to integrate
double int_function(double x1, double y1, double z1
    , double x2, double y2, double z2)
{
   double alpha = 2.;
// evaluate the different terms of the exponential
   double exp1=-2*alpha*sqrt(x1*x1+y1*y1+z1*z1);
   double exp2=-2*alpha*sqrt(x2*x2+y2*y2+z2*z2);
   double deno=sqrt(pow((x1-x2),2)+pow((y1-y2),2)+
       pow((z1-z2),2));
  if(deno <pow(10.,-6.)) { return 0;}
  else return exp(exp1+exp2)/deno;
} // end of function to evaluate
```

Switch to spherical coordinates

Useful to change to spherical coordinates

$$d\mathbf{r}_1 d\mathbf{r}_2 = r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2,$$

and

$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 cos(\beta)}}$$

with

$$\cos(\beta) = \cos(\theta_1)\cos(\theta_2) + \sin(\theta_1)\sin(\theta_2)\cos(\phi_1 - \phi_2))$$

Switch to spherical coordinates

This means that our integral becomes

$$\int_0^{\infty} r_1^2 dr_1 \int_0^{\infty} r_2^2 dr_2 \int_0^{\pi} dcos(\theta_1) \int_0^{\pi} dcos(\theta_2) \int_0^{2\pi} d\phi_1 \int_0^{2\pi} d\phi_2 \times$$

$$\frac{e^{-2\alpha(r_1+r_2)}}{\sqrt{r_1^2+r_2^2-2r_1r_2\cos(\theta_1)\cos(\theta_2)+\sin(\theta_1)\sin(\theta_2)\cos(\phi_1-\phi_2))}}$$

since

$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2+r_2^2-2r_1r_2cos(\beta)}}$$

with

$$\cos(\beta) = \cos(\theta_1)\cos(\theta_2)+\sin(\theta_1)\sin(\theta_2)\cos(\phi_1-\phi_2))$$

Adaptive methods

Before we abondon totally methods like the trapezoidal rule, we mention breefly how an adaptive integration method can be implemented.

The above methods are all based on a defined step length, normally provided by the user, dividing the integration domain with a fixed number of subintervals. This is rather simple to implement may be inefficient, in particular if the integrand varies considerably in certain areas of the integration domain. In these areas the number of fixed integration points may not be adequate. In other regions, the integrand may vary slowly and fewer integration points may be needed.

Adaptive methods

In order to account for such features, it may be convenient to first study the properties of integrand, via for example a plot of the function to integrate. If this function oscillates largely in some specific domain we may then opt for adding more integration points to that particular domain. However, this procedure needs to be repeated for every new integrand and lacks obviously the advantages of a more generic code.

Adaptive methods

Assume that we want to compute an integral using say the trapezoidal rule. We limit ourselves to a one-dimensional integral. Our integration domain is defined by $x \in [a, b]$. The algorithm goes as follows

- We compute our first approximation by computing the integral for the full domain. We label this as $I^{(0)}$. It is obtained by calling our previously discussed function **trapezoidal_rule** as

```
I0 = trapezoidal_rule(a, b, n, function);
```

- In the next step we split the integration in two, with $c = (a + b)/2$. We compute then the two integrals $I^{(1L)}$ and $I^{(1R)}$

```
I1L = trapezoidal_rule(a, c, n, function);
```

  and

```
I1R = trapezoidal_rule(c, b, n, function);
```

  With a given defined tolerance, being a small number provided by us, we estimate the difference $|I^{(1L)} + I^{(1R)} - I^{(0)}| < \mathrm{tolerance}$. If this test is satisfied, our first approximation is satisfactory.

- If not, we can set up a recursive procedure where the integral is split into subsequent subintervals until our tolerance is satisfied.

Adaptive methods

```
//     Simple recursive function that implements
    the
//     adaptive integration using the trapezoidal
    rule
const int maxrecursions = 50;
const double tolerance = 1.0E-10;
//  Takes as input the integration  limits, number
    of points, function to integrate
//  and the number of steps
void adaptive_integration(double a, double b,
    double *Integral, int n, int steps, double (*
    func)(double))
      if ( steps > maxrecursions){
        cout << 'Too many recursive steps, the
            function varies too much' << endl;
        break;
      }
```

Adaptive methods

```
double c = (a+b)*0.5;
// the whole integral
double I0 = trapezoidal_rule(a, b,n, func);
//  the left half
double I1L = trapezoidal_rule(a, c,n, func);
//  the right half
double I1R = trapezoidal_rule(c, b,n, func);
if (fabs(I1L+I1R-I0) < tolerance )  integral =
    I0;
else
{
   adaptive_integration(a, c, integral, int n,
       ++steps, func)
   adaptive_integration(c, b, integral, int n,
       ++steps, func)
}
}
// end function Adaptive integration
```

The variables **Integral** and **steps** should be initialized to zero by the function that calls the adaptive procedure.

Going Parallel with MPI

In projects 3 and 4 you will need to parallelize the codes you develop.

Task parallelism: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation or integrations are examples of this. It is almost embarrassingly trivial to parallelize Monte Carlo codes.

MPI is a message-passing library where all the routines have corresponding C/C++-binding

MPI_Command_name

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

MPI_COMMAND_NAME

What is Message Passing Interface (MPI)? Yet another library!

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library.

MPI is a specification, not a particular implementation. MPI programs should be able to run on all possible machines and run all MPI implementetations without change.

An MPI computation is a collection of processes communicating with messages.

See chapter 4.7 of lecture notes for more details.

MPI

MPI is a library specification for the message passing interface, proposed as a standard.

- independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use.
Designed for high performance.
Insert communication and synchronization functions where necessary.

Demands from the HPC community

In the field of scientific computing, there is an ever-lasting wish to do larger simulations using shorter computer time.
Development of the capacity for single-processor computers can hardly keep up with the pace of scientific computing:

- processor speed
- memory size/speed

Solution: parallel computing!

The basic ideas of parallel computing

- Pursuit of shorter computation time and larger simulation size gives rise to parallel computing.
- Multiple processors are involved to solve a global problem.
- The essence is to divide the entire computation evenly among collaborative processors. Divide and conquer.

A rough classification of hardware models

- Conventional single-processor computers can be called SISD (single-instruction-single-data) machines.
- SIMD (single-instruction-multiple-data) machines incorporate the idea of parallel processing, which use a large number of process- ing units to execute the same instruction on different data.
- Modern parallel computers are so-called MIMD (multiple-instruction- multiple-data) machines and can execute different instruction streams in parallel on different data.

Shared memory and distributed memory

- One way of categorizing modern parallel computers is to look at the memory configuration.
- In shared memory systems the CPUs share the same address space. Any CPU can access any data in the global memory.
- In distributed memory systems each CPU has its own memory. The CPUs are connected by some network and may exchange messages.

Different parallel programming paradigms

- **Task parallelism:** the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation is one example. Integration is another. However this paradigm is of limited use.
- **Data parallelism:** use of multiple threads (e.g. one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between processors are often hidden, thus easy to program. However, the user surrenders much control to a specialized compiler. Examples of data parallelism are compiler-based parallelization and OpenMP directives.

Today's situation of parallel computing

- Distributed memory is the dominant hardware configuration. There is a large diversity in these machines, from MPP (massively parallel processing) systems to clusters of off-the-shelf PCs, which are very cost-effective.
- Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware. It is primarily used for the distributed memory systems, but can also be used on shared memory systems.

In these lectures we consider only message-passing for writing parallel programs.

Overhead present in parallel computing

- **Uneven load balance**: not all the processors can perform useful work at all time.
- **Overhead of synchronization.**
- **Overhead of communication.**
- Extra computation due to parallelization.

Due to the above overhead and that certain part of a sequential algorithm cannot be parallelized we may not achieve an optimal parallelization.

Parallelizing a sequential algorithm

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This is the difficult part,
- Distribute the global work and data among P processors.

Process and processor

- We refer to process as a logical unit which executes its own code, in an MIMD style.
- The processor is a physical device on which one or several processes are executed.
- The MPI standard uses the concept process consistently throughout its documentation.

Bindings to MPI routines

MPI is a message-passing library where all the routines have corresponding C/C++-binding

MPI_Command_name

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

MPI_COMMAND_NAME

The discussion in these slides focuses on the C++ binding.

Communicator

- A group of MPI processes with a name (context).
- Any process is identified by its rank. The rank is only meaningful within a particular communicator.
- By default communicator MPI_COMM_WORLD contains all the MPI processes.
- Mechanism to identify subset of processes.
- Promotes modular design of parallel libraries.

The most important/used MPI routines

- MPI_ Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI communicator?
- MPI_Comm_rank - which one am I? (A number between 0 and size-1.)
- MPI_Reduce(Allreduce) - Collect data from all nodes and either sum them up in one or all (Allreduce). Useful for numerical integration
- MPI_Send - send a message to a particular process within an MPI communicator
- MPI_Recv - receive a message from a particular process within an MPI communicator

The first MPI C/C++ program

Let every process write "Hello world" on the standard output.
This is program2.cpp of chapter 4.

```
using namespace std;
#include <mpi.h>
#include <iostream>
int main (int nargs, char* args[])
{
int numprocs, my_rank;
//   MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
cout << "Hello world, I have  rank " << my_rank <<
    " out of "
     << numprocs << endl;
//  End MPI
MPI_Finalize ();
```

The Fortran program

```fortran
PROGRAM hello
INCLUDE "mpif.h"
INTEGER:: size, my_rank, ierr

CALL  MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
WRITE(*,*)"Hello world, I've rank ",my_rank," out
   of ",size
CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

Note 1

The output to screen is not ordered since all processes are trying to write to screen simultaneously. It is then the operating system which opts for an ordering. If we wish to have an organized output, starting from the first process, we may rewrite our program as in the next example (program3.cpp), see again chapter 4.7 of lecture notes.

Ordered output with MPI_Barrier

```
int main (int nargs, char* args[])
{
 int numprocs, my_rank, i;
 MPI_Init (&nargs, &args);
 MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
 for (i = 0; i < numprocs; i++) {}
 MPI_Barrier (MPI_COMM_WORLD);
 if (i == my_rank) {
 cout << "Hello world, I have  rank " << my_rank <<
        " out of " << numprocs << endl;}
      MPI_Finalize ();
```

Note 2

Here we have used the *MPI_Barrier* function to ensure that that every process has completed its set of instructions in a particular order. A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here *MPI_COMM_WORLD* have called *MPI_Barrier*. The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like *MPI_ALLREDUCE* to be discussed later, have the same property; viz. no process can exit the operation until all processes have started. However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

Strategies

- Develop codes locally, run with some few processes and test your codes. Do benchmarking, timing and so forth on local nodes, for example your laptop. You can install MPICH2 on your laptop (most new laptos come with dual cores). You can test with one node at the lab.
- When you are convinced that your codes run correctly, you start your production runs on available supercomputers, in our case titan.uio.no.

How do I run MPI on the machines at the lab (MPICH2)

The machines at the lab are all quad-cores

- Compile with mpicxx or mpic++
- Set up collaboration between processes and run

```
mpd --ncpus=4 &
#  run code with
mpiexec -n 4 ./nameofprog
```

  Here we declare that we will use 4 processes via the $-ncpus$ option and via $-n4$ when running.
- End with

```
mpdallexit
```

Can I do it on my own PC/laptop?

Of course:

- go to `http://www.mcs.anl.gov/research/projects/mpich2/`
- follow the instructions and install it on your own PC/laptop

I don't have windows as operating system and need dearly your feedback.

Ordered output with MPI_Recv and MPI_Send

```
.....
int numprocs, my_rank, flag;
MPI_Status status;
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
if (my_rank > 0)
MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100,
            MPI_COMM_WORLD, &status);
cout << "Hello world, I have  rank " << my_rank <<
    " out of "
<< numprocs << endl;
if (my_rank < numprocs-1)
MPI_Send (&my_rank, 1, MPI_INT, my_rank+1,
           100, MPI_COMM_WORLD);
MPI_Finalize ();
```

Note 3

The basic sending of messages is given by the function *MPI_SEND*, which in C/C++ is defined as

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)}
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable **buf** is the variable we wish to send while **count** is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

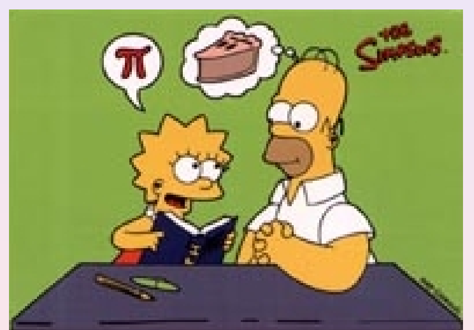
Note 4

Once you have sent a message, you must receive it on another task. The function **MPI_RECV** is similar to the send call.

```
int MPI_Recv( void *buf, int count, MPI_Datatype
    datatype,
              int source,
              int tag, MPI_Comm comm, MPI_Status *
                 status )
```

The arguments that are different from those in *MPI_SEND* are **buf** which is the name of the variable where you will be storing the received data, **source** which replaces the destination in the send command. This is the return ID of the sender.
Finally, we have used **MPI_Status status;** where one can check if the receive was completed.
The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.
Armed with this wisdom, performed all hello world greetings, we are now ready for serious work.

Integrating π



Examples

- Go to the webpage and click on the programs link
- Go to MPI and then chapter 4
- Look at program5.ccp and program6.cpp. (Fortran version also available).
- These codes compute π using the rectangular and trapezoidal rules.

Integration algos

The trapezoidal rule (example6.cpp)

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right).$$

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length h and heigth given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given rectangle. This is used in program5.cpp.

Dissection of example program5.cpp

```
1   //    Reactangle rule and numerical integration
2   using namespace std;
3   #include <mpi.h>
4   #include <iostream>

5   int main (int nargs, char* args[])
6   {
7       int numprocs, my_rank, i, n = 1000;
8       double local_sum, rectangle_sum, x, h;
9       //   MPI initializations
10      MPI_Init (&nargs, &args);
11      MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
12      MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

Dissection of example program5.cpp

```
13      //   Read from screen a possible new vaue of
    n
14      if (my_rank == 0 && nargs > 1) {
15         n = atoi(args[1]);
16      }
17      h = 1.0/n;
18      //  Broadcast n and h to all processes
19      MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD
   );
20      MPI_Bcast (&h, 1, MPI_DOUBLE, 0,
   MPI_COMM_WORLD);
21      //  Every process sets up its contribution
   to the integral
22      local_sum = 0.;
```

Dissection of example program5.cpp

After the standard initializations with MPI such as MPI_Init, MPI_Comm_size and MPI_Comm_rank, MPI_COMM_WORLD contains now the number of processes defined by using for example

```
mpiexec -np 10 ./prog.x
```

In line 4 we check if we have read in from screen the number of mesh points n. Note that in line 7 we fix $n = 1000$, however we have the possibility to run the code with a different number of mesh points as well. If my_rank equals zero, which correponds to the master node, then we read a new value of n if the number of arguments is larger than two. This can be done as follows when we run the code

```
mpiexec -np 10 ./prog.x  10000
```

Dissection of example program5.cpp

```
23        for (i = my_rank; i < n; i += numprocs) {
24          x = (i+0.5)*h;
25          local_sum += 4.0/(1.0+x*x);
26       }
27        local_sum *= h;
```

In line 17 we define also the step length h. In lines 19 and 20 we use the broadcast function MPI_Bcast. We use this particular function because we want data on one processor (our master node) to be shared with all other processors. The broadcast function sends data to a group of processes.

Dissection of example program5.cpp

The MPI routine MPI_Bcast transfers data from one task to a group of others. The format for the call is in C++ given by the parameters of

```
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);.
MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

in a case of a double. The general structure of this function is

```
MPI_Bcast( void *buf, int count, MPI_Datatype
    datatype, int root, MPI_Comm comm).
```

All processes call this function, both the process sending the data (with rank zero) and all the other processes in MPI_COMM_WORLD. Every process has now copies of n and h, the number of mesh points and the step length, respectively.

We transfer the addresses of n and h. The second argument represents the number of data sent. In case of a one-dimensional array, one needs to transfer the number of array elements. If you have an $n \times m$ matrix, you must transfer $n \times m$. We need also to specify whether the variable type we transfer is a non-numerical such as a logical or character variable or numerical of the integer, real or complex type.

Dissection of example program5.cpp

```
28      if (my_rank == 0) {
29        MPI_Status status;
30        rectangle_sum = local_sum;
31        for (i=1; i < numprocs; i++) {
32          MPI_Recv(&local_sum,1,MPI_DOUBLE,
    MPI_ANY_SOURCE,500,
                              MPI_COMM_WORLD,&status);
33          rectangle_sum += local_sum;
34        }
35        cout << "Result: " << rectangle_sum  <<
    endl;
36      } else
37        MPI_Send(&local_sum,1,MPI_DOUBLE,0,500,
    MPI_COMM_WORLD);
38      // End MPI
39      MPI_Finalize ();
40      return 0;
41    }
```

Dissection of example program5.cpp

In lines 23-27, every process sums its own part of the final sum used by the rectangle rule. The receive statement collects the sums from all other processes in case my_rank == 0, else an MPI send is performed. If we are not the master node, we send the results, else they are received and the local results are added to final sum. The above can be rewritten using the MPI_allreduce, as discussed in the next example.

The above function is not very elegant. Furthermore, the MPI instructions can be simplified by using the functions MPI_Reduce or MPI_Allreduce. The first function takes information from all processes and sends the result of the MPI operation to one process only, typically the master node. If we use MPI_Allreduce, the result is sent back to all processes, a feature which is useful when all nodes need the value of a joint operation. We limit ourselves to MPI_Reduce since it is only one process which will print out the final number of our calculation, The arguments to MPI_Allreduce are the same.

MPI_reduce

Call as

```
MPI_reduce( void *senddata, void* resultdata, int
    count,
      MPI_Datatype datatype, MPI_Op, int root,
         MPI_Comm comm)
```

The two variables *senddata* and *resultdata* are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable *count* represents the total dimensionality, 1 in case of just one variable, while MPI_Datatype defines the type of variable which is sent and received.
The new feature is MPI_Op. It defines the type of operation we want to do. In our case, since we are summing the rectangle contributions from every process we define MPI_Op = MPI_SUM. If we have an array or matrix we can search for the largest og smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINOC. If we want the product we write MPI_PROD.
MPI_Allreduce is defined as

```
MPI_Alreduce( void *senddata, void* resultdata, int
      count,
          MPI_Datatype datatype, MPI_Op, MPI_Comm
            comm)}.
```

Dissection of example program6.cpp

```
//    Trapezoidal rule and numerical integration
   usign MPI, example program6.cpp
using namespace std;
#include <mpi.h>
#include <iostream>

//     Here we define various functions called by
   the main program

double int_function(double );
double trapezoidal_rule(double , double , int ,
   double (*)(double));

//   Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum,
      local_sum;
  double  time_start, time_end, total_time;
```

Dissection of example program6.cpp

```
//  MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
time_start = MPI_Wtime();
//  Fixed values for a, b and n
a = 0.0 ; b = 1.0;  n = 1000;
h = (b-a)/n;     // h is the same for all
    processes
local_n = n/numprocs;
// make sure n > numprocs, else integer division
    gives zero
// Length of each process' interval of
// integration = local_n*h.
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
```

Dissection of example program6.cpp

```
total_sum = 0.0;
local_sum = trapezoidal_rule(local_a, local_b,
    local_n,
                                             &int_function);
MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
                 MPI_SUM, 0, MPI_COMM_WORLD);
time_end = MPI_Wtime();
total_time = time_end-time_start;
if ( my_rank == 0) {
  cout << "Trapezoidal rule = " <<  total_sum <<
      endl;
  cout << "Time = " <<  total_time
         << " on number of processors: "  <<
            numprocs  << endl;
}
// End MPI
MPI_Finalize ();
return 0;
}  // end of main program
```

Dissection of example program6.cpp

We use MPI_reduce to collect data from each process. Note also the use of the function MPI_Wtime. The final functions are

```
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate
```

Dissection of example program6.cpp

Implementation of the trapezoidal rule.

```
//   this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n,
                        double (*func)(double))
{
  double trapez_sum;
  double fa, fb, x, step;
  int    j;
  step=(b-a)/((double) n);
  fa=(*func)(a)/2. ;
  fb=(*func)(b)/2. ;
  trapez_sum=0.;
  for (j=1; j <= n-1; j++){
    x=j*step+a;
    trapez_sum+=(*func)(x);
  }
  trapez_sum=(trapez_sum+fb+fa)*step;
  return trapez_sum;
}  // end trapezoidal_rule
```

Plan for Monte Carlo Lectures, chapters 11-14 in Lecture notes

- This week: intro, MC integration and probability distribution functions (PDFs)
- Next week: More on integration, PDFs, MC integration and random walks. Project 3 is presented Monday.
- Third week: random walks and statistical physics, presentation of project 4.
- Fourth week: Statistical physics
- Fifth week: Most likely quantum Monte Carlo

Approximately from Wednesday 5/10 till Wednesday 2/11. The rest of the semester is reserved to a discussion of differential equations and project work.

Monte Carlo Keywords

Consider it is a numerical experiment

- Be able to generate random variables following a given probability distribution function PDF
- Find a probability distribution function (PDF).
- Sampling rule for accepting a move
- Compute standard deviation and other expectation values
- Techniques for improving errors

Enhances algorithmic thinking!

Probability Distribution Functions PDF

	Discrete PDF	continuous PDF
Domain	$\{x_1, x_2, x_3, \ldots, x_N\}$	$[a, b]$
probability	$p(x_i)$	$p(x)dx$
Cumulative	$P_i = \sum_{l=1}^{i} p(x_l)$	$P(x) = \int_a^x p(t)dt$
Positivity	$0 \leq p(x_i) \leq 1$	$p(x) \geq 0$
Positivity	$0 \leq P_i \leq 1$	$0 \leq P(x) \leq 1$
Monotonuous	$P_i \geq P_j$ if $x_i \geq x_j$	$P(x_i) \geq P(x_j)$ if $x_i \geq x_j$
Normalization	$P_N = 1$	$P(b) = 1$

Expectation Values

- Discrete PDF

$$E[x^k] = \langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i),$$

provided that the sums (or integrals) $\sum_{i=1}^{N} p(x_i)$ converge absolutely (viz , $\sum_{i=1}^{N} |p(x_i)|$ converges)

- Continuous PDF

$$E[x^k] = \langle x^k \rangle = \int_a^b x^k p(x) dx,$$

- Function $f(x)$

$$E[f^k] = \langle f^k \rangle = \int_a^b f^k p(x) dx,$$

- Variance

$$\sigma_f^2 = E[f^2] - (E[f])^2 = \langle f^2 \rangle - \langle f \rangle^2$$

Important PDFs

- uniform distribution
$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$
which gives for $a = 0, b = 1$ $p(x) = 1$ for $x \in [0, 1]$ and zero else.
- exponential distribution
$$p(x) = \alpha e^{-\alpha x},$$
with probability different from zero in $[0, \infty]$
- normal distribution (Gaussian)
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$
with probability different from zero in $[-\infty, \infty]$

All random number generators use the uniform distribution for $x \in [0, 1]$.

Why Monte Carlo?

An example from quantum mechanics: most problems of interest in e.g., atomic, molecular, nuclear and solid state physics consist of a large number of interacting electrons and ions or nucleons. The total number of particles N is usually sufficiently large that an exact solution cannot be found. Typically, the expectation value for a chosen hamiltonian for a system of N particles is

$$\langle H \rangle =$$

$$\frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N) H(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)},$$

an in general intractable problem.

This integral is actually the starting point in a Variational Monte Carlo calculation.

Gaussian quadrature: Forget it! given 10 particles and 10 mesh points for each degree of freedom and an ideal 1 petaflops machine (all operations take the same time), how long will it ta ke to compute the above integral? Lifetime of the universe $T \approx 4.7 \times 10^{17}$s.

More on dimensionality

As an example from the nuclear many-body problem, we have Schrödinger's equation as a differential equation

$$\hat{H}\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A) = E\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A)$$

where

$$\mathbf{r}_1, .., \mathbf{r}_A,$$

are the coordinates and

$$\alpha_1, .., \alpha_A,$$

are sets of relevant quantum numbers such as spin and isospin for a system of A nucleons ($A = N + Z$, N being the number of neutrons and Z the number of protons).

Even more on dimensionality

There are

$$2^A \times \left(\begin{array}{c} A \\ Z \end{array} \right)$$

coupled second-order differential equations in $3A$ dimensions.

For a nucleus like ^{10}Be this number is **215040**. This is a truely challenging many-body problem.

But what do we gain by Monte Carlo Integration?

A crude approach consists in setting all weights equal 1, $\omega_i = 1$. With $dx = h = (b-a)/N$ where $b = 1$, $a = 0$ in our case and h is the step size. The integral

$$I = \int_0^1 f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i),$$

can be rewritten using the concept of the average of the function f for a given PDF $p(x)$ as

$$E[f] = \langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i)p(x_i),$$

and identify $p(x)$ with the uniform distribution, viz $p(x) = 1$ when $x \in [0,1]$ and zero for all other values of x. The integral is then the average of f over the interval $x \in [0,1]$

$$I = \int_0^1 f(x)dx \approx E[f] = \langle f \rangle.$$

But what do we gain by Monte Carlo Integration?

In addition to the average value $\langle f \rangle$ the other important quantity in a Monte-Carlo calculation is the variance σ^2 and the standard deviation σ. We define first the variance of the integral with f for a uniform distribution in the interval $x \in [0,1]$ to be

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \langle f \rangle)^2 p(x_i),$$

and inserting the uniform distribution this yields

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2,$$

or

$$\sigma_f^2 = E[f^2] - (E[f])^2 = \left(\langle f^2 \rangle - \langle f \rangle^2\right).$$

which is nothing but a measure of the extent to which f deviates from its average over the region of integration. The standard deviation is defined as the square root of the variance.

But what do we gain by Monte Carlo Integration?

If we consider the above results for a fixed value of N as a measurement, we could however recalculate the above average and variance for a series of different measurements. If each such measuremernt produces a set of averages for the integral I denoted $\langle f\rangle_l$, we have for M measurements that the integral is given by

$$\langle I\rangle_M = \frac{1}{M}\sum_{l=1}^{M}\langle f\rangle_l.$$

If we can consider the probability of correlated events to be zero, we can rewrite the variance of these series of measurements as (equating $M = N$)

$$\sigma_N^2 \approx \frac{1}{N}\left(\langle f^2\rangle - \langle f\rangle^2\right) = \frac{\sigma_f^2}{N}.$$

We note that the standard deviation is proportional with the inverse square root of the number of measurements

$$\sigma_N \sim \frac{1}{\sqrt{N}}.$$

The aim in Monte Carlo calculations is to have σ_N as small as possible after N samples. The results from one sample represents, since we are using concepts from statistics, a 'measurement'.

But what do we gain by Monte Carlo Integration?

- We saw that the trapezoidal rule carries a truncation error $O(h^2)$, with h the step length.
- Quadrature rules such as Newton-Cotes have a truncation error which goes like $\sim O(h^k)$, with $k \geq 1$. Recalling that the step size is defined as $h = (b-a)/N$, we have an error which goes like $\sim N^{-k}$.
- Monte Carlo integration is more efficient in higher dimensions. Assume that our integration volume is a hypercube with side L and dimension d. This cube contains hence $N = (L/h)^d$ points and therefore the error in the result scales as $N^{-k/d}$ for the traditional methods.
- The error in the Monte carlo integration is however independent of d and scales as $\sigma \sim 1/\sqrt{N}$, always!
- Comparing this with traditional methods, shows that Monte Carlo integration is more efficient than an order-k algorithm when $d > 2k$

Week 41

Monte Carlo methods

- Monday: Repetition from last week
- Numerical integration with Monte Carlo methods and project 3
- Wednesday:
- Random number generators and properties of PDFs
- PDFs, covariance and standard deviation and standard error

Monte Carlo Keywords

Consider it is a numerical experiment

- Be able to generate random variables following a given probability distribution function PDF
- Find a probability distribution function (PDF).
- Sampling rule for accepting a move
- Compute standard deviation and other expectation values
- Techniques for improving errors

Enhances algorithmic thinking!

Some simple examples: Example 1: Particles in a Box

Consider a box divided into two equal halves separated by a wall. At the beginning, time $t = 0$, there are N particles on the left side. A small hole in the wall is then opened and one particle can pass through the hole per unit time.

After some time the system reaches its equilibrium state with equally many particles in both halves, $N/2$. Instead of determining complicated initial conditions for a system of N particles, we model the system by a simple statistical model. In order to simulate this system, which may consist of $N \gg 1$ particles, we assume that all particles in the left half have equal probabilities of going to the right half.

Particles in a Box

We introduce the label n_l to denote the number of particles at every time on the left side, and $n_r = N - n_l$ for those on the right side. The probability for a move to the right during a time step Δt is n_l/N. The algorithm for simulating this problem may then look like as follows

- Choose the number of particles N.
- Make a loop over time, where the maximum time should be larger than the number of particles N.
- For every time step Δt there is a probability n_l/N for a move to the right. Compare this probability with a random number x.
- If $x \leq n_l/N$, decrease the number of particles in the left half by one, i.e., $n_l = n_l - 1$. Else, move a particle from the right half to the left, i.e., $n_l = n_l + 1$. **This is our sampling rule**
- Increase the time by one unit (the external loop).

In this case, a Monte Carlo sample corresponds to one time unit Δt.

Particles in a Box

```
// setup of initial conditions
nleft = initial_n_particles;
max_time = 10*initial_n_particles;
idum = -1;
// sampling over number of particles
for( time=0; time <= max_time; time++){
  random_n = ((int) initial_n_particles*ran0(&idum));
  if ( random_n <= nleft){
    nleft -= 1;
  }
  else{
    nleft += 1;
  }
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << setw(15) << time;
  ofile << setw(15) << nleft << endl;
}
```

Example 2: Radioactive Decay

Assume that a the time $t = 0$ we have $N(0)$ nuclei of type X which can decay radioactively. At a time $t > 0$ we are left with $N(t)$ nuclei. With a transition probability ω, which expresses the probability that the system will make a transition to another state during a time step of one second, we have the following first-order differential equation

$$dN(t) = -\omega N(t)dt,$$

whose solution is

$$N(t) = N(0)e^{-\omega t},$$

where we have defined the mean lifetime τ of X as

$$\tau = \frac{1}{\omega}.$$

If a nucleus X decays to a daugther nucleus Y which also can decay, we get the following coupled equations

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t),$$

and

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) + \omega_X N_X(t).$$

Radioactive Decay

Probability for a decay of a particle during a time step Δt is

$$\frac{\Delta N(t)}{N(t)\Delta t} = -\lambda$$

λ is inversely proportional to the lifetime

- Choose the number of particles $N(t=0) = N_0$.
- Make a loop over the number of time steps, with maximum time bigger than the number of particles N_0
- At every time step there is a probability λ for decay. Compare this probability with a random number x.
- If $x \leq \lambda$, reduce the number of particles with one i.e., $N = N - 1$. If not, keep the same number of particles till the next time step. **This is our sampling rule**
- Increase by one the time step (the external loop)

Radioactive Decay

```
    idum=-1;  // initialise random number generator
    // loop over monte carlo cycles
    // One monte carlo loop is one sample
    for (cycles = 1; cycles <= number_cycles; cycles++){
      n_unstable = initial_n_particles;
      //  accumulate the number of particles per time step per trial
      ncumulative[0] += initial_n_particles;
      // loop over each time step
      for (time=1; time <= max_time; time++){
        // for each time step, we check each particle
        particle_limit = n_unstable;
        for ( np = 1; np <=  particle_limit; np++) {
          if( ran0(&idum) <= decay_probability) {
            n_unstable=n_unstable-1;
          }
        }  // end of loop over particles
        ncumulative[time] += n_unstable;
      }  // end of loop over time steps
    }    // end of loop over MC trials
  }   // end mc_sampling function
```

Example 3: Acceptance-Rejection Method

This is a rather simple and appealing method after von Neumann. Assume that we are looking at an interval $x \in [a, b]$, this being the domain of the PDF $p(x)$. Suppose also that the largest value our distribution function takes in this interval is M, that is

$$p(x) \leq M \qquad x \in [a, b].$$

Then we generate a random number x from the uniform distribution for $x \in [a, b]$ and a corresponding number s for the uniform distribution between $[0, M]$. If

$$p(x) \geq s,$$

we accept the new value of x, else we generate again two new random numbers x and s and perform the test in the latter equation again.

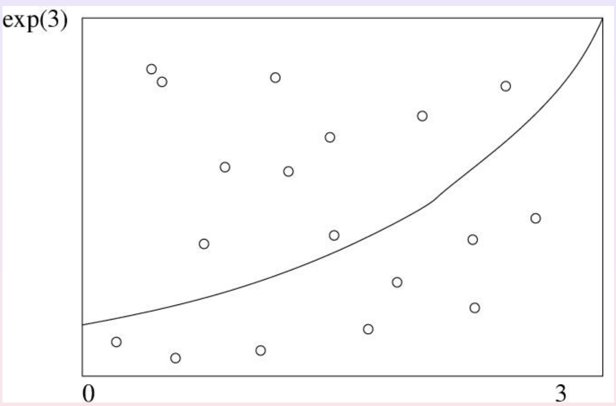
Acceptance-Rejection Method

As an example, consider the evaluation of the integral

$$I = \int_0^3 \exp(x) dx.$$

Obviously to derive it analytically is much easier, however the integrand could pose some more difficult challenges. The aim here is simply to show how to implent the acceptance-rejection algorithm. The integral is the area below the curve $f(x) = \exp(x)$. If we uniformly fill the rectangle spanned by $x \in [0, 3]$ and $y \in [0, \exp(3)]$, the fraction below the curve obatained from a uniform distribution, and multiplied by the area of the rectangle, should approximate the chosen integral. It is rather easy to implement this numerically, as shown in the following code.

Simple Plot of the Accept-Reject Method



Acceptance-Rejection Method

```
//    Loop over Monte Carlo trials n
      integral =0.;
      for ( int i = 1;  i <= n; i++){
//    Finds a random value for x in the interval [0,3]
           x = 3*ran0(&idum);
//    Finds y-value between [0,exp(3)]
           y = exp(3.0)*ran0(&idum);
//    if the value of y at exp(x) is below the curve, we accept
//    THIS IS OUR SAMPLING RULE
           if ( y  < exp(x)) s = s+ 1.0;
//    The integral is area enclosed below the line f(x)=exp(x)
     }
//   Then we multiply with the area of the rectangle and
//   divide by the number of cycles
     Integral = 3.*exp(3.)*s/n
```

Monte Carlo Integration

With uniform distribution $p(x) = 1$ for $x \in [0, 1]$ and zero else

$$I = \int_0^1 f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i),$$

$$I = \int_0^1 f(x)dx \approx E[f] = \langle f \rangle.$$

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2,$$

or

$$\sigma_f^2 = E[f^2] - (E[f])^2 = \left(\langle f^2 \rangle - \langle f \rangle^2\right).$$

Brute Force Algorithm for Monte Carlo Integration

- Choose the number of Monte Carlo samples N.
- Make a loop over N and for every step generate a random number x_i in the interval $x_i \in [0, 1]$ by calling a random number generator.
- Use this number to compute $f(x_i)$.
- Find the contribution to the variance and the mean value for every loop contribution.
- After N samplings, compute the final mean value and the standard deviation

Brute Force Integration

```
// crude mc function to calculate pi
   int i, n;
   long idum;
   double crude_mc, x, sum_sigma, fx, variance;
   cout << "Read in the number of Monte-Carlo
      samples" << endl;
   cin >> n;
   crude_mc = sum_sigma=0. ; idum=-1 ;
//    evaluate the integral with the a crude Monte-
   Carlo method
    for ( i = 1;  i <= n; i++){
          x=ran0(&idum);
          fx=func(x);
          crude_mc += fx;
          sum_sigma += fx*fx;
    }
    crude_mc = crude_mc/((double) n );
    sum_sigma = sum_sigma/((double) n );
    variance=sum_sigma-crude_mc*crude_mc;
```

Or: another Brute Force Integration

```
// crude mc function to calculate pi
int main()
{
  const int n = 1000000;
  double x, fx, pi, invers_period, pi2;
  int i;
  invers_period = 1./RAND_MAX;
  srand(time(NULL));
  pi = pi2 = 0.;
  for (i=0; i<n;i++)
    {
      x = double(rand())*invers_period;
      //   This is our sampling rule, all points
           accepted
      fx = 4./(1+x*x);
      pi += fx;
      pi2 += fx*fx;
    }
  pi /= n;  pi2 = pi2/n - pi*pi;
  cout << "pi=" << pi << " sigma^2=" << pi2 << endl
```

Brute Force Integration

Note the call to a function which generates random numbers according to the uniform distribution

```
long idum;
idum=-1 ;
.....
x=ran0(&idum);
....
```

or

```
...
invers_period = 1./RAND_MAX;
srand(time(NULL));
...
x = double(rand())*invers_period;
```

Algorithm for Monte Carlo Integration, Results

N	I	σ_N
10	3.10263E+00	3.98802E-01
100	3.02933E+00	4.04822E-01
1000	3.13395E+00	4.22881E-01
10000	3.14195E+00	4.11195E-01
100000	3.14003E+00	4.14114E-01
1000000	3.14213E+00	4.13838E-01
10000000	3.14177E+00	4.13523E-01
10^9	3.14162E+00	4.13581E-01

We note that as N increases, the integral itself never reaches more than an agreement to the fourth or fifth digit. The variance also oscillates around its exact value $4.13581E-01$. Note well that the variance need not be zero but can, with appropriate redefinitions of the integral be made smaller. A smaller variance yields also a smaller standard deviation. This is the topic of importance sampling.

Transformation of Variables

The starting point is always the uniform distribution

$$p(x)dx = \begin{cases} dx & 0 \le x \le 1 \\ 0 & else \end{cases}$$

with $p(x) = 1$ and satisfying

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

All random number generators provided in the program library generate numbers in this domain.
When we attempt a transformation to a new variable $x \to y$ we have to conserve the probability

$$p(y)dy = p(x)dx,$$

which for the uniform distribution implies

$$p(y)dy = dx.$$

Transformation of Variables

Let us assume that $p(y)$ is a PDF different from the uniform PDF $p(x) = 1$ with $x \in [0, 1]$. If we integrate the last expression we arrive at

$$x(y) = \int_0^y p(y')dy',$$

which is nothing but the cumulative distribution of $p(y)$, i.e.,

$$x(y) = P(y) = \int_0^y p(y')dy'.$$

This is an important result which has consequences for eventual improvements over the brute force Monte Carlo.

Example 1, a general Uniform Distribution

Suppose we have the general uniform distribution

$$p(y)dy = \begin{cases} \frac{dy}{b-a} & a \le y \le b \\ 0 & else \end{cases}$$

If we wish to relate this distribution to the one in the interval $x \in [0,1]$ we have

$$p(y)dy = \frac{dy}{b-a} = dx,$$

and integrating we obtain the cumulative function

$$x(y) = \int_a^y \frac{dy'}{b-a},$$

yielding

$$y = a + (b-a)x,$$

a well-known result!

Example 2, from Uniform to Exponential

Assume that

$$p(y) = e^{-y},$$

which is the exponential distribution, important for the analysis of e.g., radioactive decay. Again, $p(x)$ is given by the uniform distribution with $x \in [0, 1]$, and with the assumption that the probability is conserved we have

$$p(y)dy = e^{-y}dy = dx,$$

which yields after integration

$$x(y) = P(y) = \int_0^y \exp(-y')dy' = 1 - \exp(-y),$$

or

$$y(x) = -ln(1 - x).$$

This gives us the new random variable y in the domain $y \in [0, \infty)$ determined through the random variable $x \in [0, 1]$ generated by our favorite random generator.

Example 2, from Uniform to Exponential

This means that if we can factor out $\exp(-y)$ from an integrand we may have

$$I = \int_0^\infty F(y)dy = \int_0^\infty \exp(-y)G(y)dy$$

which we rewrite as

$$\int_0^\infty \exp(-y)G(y)dy = \int_0^\infty \frac{dx}{dy}G(y)dy \approx \frac{1}{N}\sum_{i=1}^{N} G(y(x_i)),$$

where x_i is a random number in the interval [0,1].

Note that in practical implementations, our random number generators for the uniform distribution never return exactly 0 or 1, but we we may come very close. We should thus in principle set $x \in (0,1)$.

Example 2, from Uniform to Exponential

The algorithm is rather simple. In the function which sets up the integral, we simply need the random number generator for the uniform distribution in order to obtain numbers in the interval [0,1]. We obtain y by the taking the logarithm of $(1-x)$. Our calling function which sets up the new random variable y may then include statements like

```
.....
idum=-1;
x=ran0(&idum);
y=-log(1.-x);
.....
```

Example 3

Another function which provides an example for a PDF is

$$p(y)dy = \frac{dy}{(a+by)^n},$$

with $n > 1$. It is normalizable, positive definite, analytically integrable and the integral is invertible, allowing thereby the expression of a new variable in terms of the old one. The integral

$$\int_0^\infty \frac{dy}{(a+by)^n} = \frac{1}{(n-1)ba^{n-1}},$$

gives

$$p(y)dy = \frac{(n-1)ba^{n-1}}{(a+by)^n}dy,$$

which in turn gives the cumulative function

$$x(y) = P(y) = \int_0^y \frac{(n-1)ba^{n-1}}{(a+bx)^n}dy' =,$$

resulting in

$$y = \frac{a}{b}\left((1-x)^{-1/(n-1)} - 1\right).$$

Example 4, from Uniform to Normal

For the normal distribution, expressed here as

$$g(x,y) = \exp\left(-(x^2+y^2)/2\right)dxdy.$$

it is rather difficult to find an inverse since the cumulative distribution is given by the error function $erf(x)$.
If we however switch to polar coordinates, we have for x and y

$$r = \left(x^2+y^2\right)^{1/2} \qquad \theta = tan^{-1}\frac{x}{y},$$

resulting in

$$g(r,\theta) = r\exp\left(-r^2/2\right)drd\theta,$$

where the angle θ could be given by a uniform distribution in the region $[0, 2\pi]$.

Following example 1 above, this implies simply multiplying random numbers $x \in [0, 1]$ by 2π.

Example 4, from Uniform to Normal

The variable r, defined for $r \in [0, \infty)$ needs to be related to to random numbers $x' \in [0, 1]$. To achieve that, we introduce a new variable

$$u = \frac{1}{2}r^2,$$

and define a PDF

$$\exp(-u)du,$$

with $u \in [0, \infty)$. Using the results from example 2, we have that

$$u = -ln(1 - x'),$$

where x' is a random number generated for $x' \in [0, 1]$. With

$$x = r\cos(\theta) = \sqrt{2u}\cos(\theta),$$

and

$$y = r\sin(\theta) = \sqrt{2u}\sin(\theta),$$

we can obtain new random numbers x, y through

$$x = \sqrt{-2ln(1 - x')}\cos(\theta),$$

and

$$y = \sqrt{-2ln(1 - x')}\sin(\theta),$$

with $x' \in [0, 1]$ and $\theta \in 2\pi[0, 1]$.

Example 4, from Uniform to Normal

A function which yields such random numbers for the normal distribution would include statements like

```
.....
idum=-1;
radius=sqrt(-2*ln(1.-ran0(idum)));
theta=2*pi*ran0(idum);
x=radius*cos(theta);
y=radius*sin(theta);
.....
```

Box-Mueller Method for Normal Deviates

```
// random numbers with gaussian distribution
double gaussian_deviate(long * idum)
{
  static int iset = 0;
  static double gset;
  double fac, rsq, v1, v2;
  if ( idum < 0) iset =0;
  if (iset == 0) {
    do {
      v1 = 2.*ran0(idum) -1.0;
      v2 = 2.*ran0(idum) -1.0;
      rsq = v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.);
    fac = sqrt(-2.*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    return v2*fac;
  } else {
    iset =0;
    return gset;
  }
```

Importance Sampling, chapter 11.4

With the aid of the above variable transformations we address now one of the most widely used approaches to Monte Carlo integration, namely importance sampling. Let us assume that $p(y)$ is a PDF whose behavior resembles that of a function F defined in a certain interval $[a, b]$. The normalization condition is

$$\int_a^b p(y)dy = 1.$$

We can rewrite our integral as

$$I = \int_a^b F(y)dy = \int_a^b p(y)\frac{F(y)}{p(y)}dy.$$

Importance Sampling

Since random numbers are generated for the uniform distribution $p(x)$ with $x \in [0, 1]$, we need to perform a change of variables $x \to y$ through

$$x(y) = \int_a^y p(y')dy',$$

where we used

$$p(x)dx = dx = p(y)dy.$$

If we can invert $x(y)$, we find $y(x)$ as well.

Importance Sampling

With this change of variables we can express the integral of Eq. (388) as

$$I = \int_a^b p(y) \frac{F(y)}{p(y)} dy = \int_a^b \frac{F(y(x))}{p(y(x))} dx,$$

meaning that a Monte Carlo evalutaion of the above integral gives

$$\int_a^b \frac{F(y(x))}{p(y(x))} dx = \frac{1}{N} \sum_{i=1}^{N} \frac{F(y(x_i))}{p(y(x_i))}.$$

The advantage of such a change of variables in case $p(y)$ follows closely F is that the integrand becomes smooth and we can sample over relevant values for the integrand. It is however not trivial to find such a function p. The conditions on p which allow us to perform these transformations are

1. p is normalizable and positive definite,
2. it is analytically integrable and
3. the integral is invertible, allowing us thereby to express a new variable in terms of the old one.

Importance Sampling

The algorithm for this procedure is

- Use the uniform distribution to find the random variable y in the interval [0,1]. $p(x)$ is a user provided PDF.
- Evaluate thereafter

$$I = \int_a^b F(x)dx = \int_a^b p(x)\frac{F(x)}{p(x)}dx,$$

by rewriting

$$\int_a^b p(x)\frac{F(x)}{p(x)}dx = \int_a^b \frac{F(x(y))}{p(x(y))}dy,$$

since

$$\frac{dy}{dx} = p(x).$$

- Perform then a Monte Carlo sampling for

$$\int_a^b \frac{F(x(y))}{p(x(y))}dy, \approx \frac{1}{N}\sum_{i=1}^{N}\frac{F(x(y_i))}{p(x(y_i))},$$

with $y_i \in [0,1]$,

- Evaluate the variance

Demonstration of Importance Sampling

$$I = \int_0^1 F(x)dx = \int_0^1 \frac{1}{1+x^2}dx = \frac{\pi}{4}.$$

We choose the following PDF (which follows closely the function to integrate)

$$p(x) = \frac{1}{3}(4-2x) \qquad \int_0^1 p(x)dx = 1,$$

resulting

$$\frac{F(0)}{p(0)} = \frac{F(1)}{p(1)} = \frac{3}{4}.$$

Check that it fullfils the requirements of a PDF. We perform then the change of variables (via the Cumulative function)

$$y(x) = \int_0^x p(x')dx' = \frac{1}{3}x(4-x),$$

or

$$x = 2 - (4-3y)^{1/2}$$

We have that when $y = 0$ then $x = 0$ and when $y = 1$ we have $x = 1$.

Simple Code

```
//   evaluate the integral with importance sampling
     for ( int i = 1;  i <= n; i++){
       x = ran0(&idum);  // random numbers in [0,1]
       y = 2 - sqrt(4-3*x);  // new random numbers
       fy=3*func(y)/(4-2*y); // weighted function
       int_mc += fy;
       sum_sigma += fy*fy;
     }
     int_mc = int_mc/((double) n );
     sum_sigma = sum_sigma/((double) n );
     variance=(sum_sigma-int_mc*int_mc);
```

Test Runs and Comparison with Brute Force for $\pi = 3.14159$

The suffix *cr* stands for the brute force approach while *is* stands for the use of importance sampling. All calculations use ran0 as function to generate the uniform distribution.

N	I_{cr}	σ_{cr}	I_{is}	σ_{is}
10000	3.13395E+00	4.22881E-01	3.14163E+00	6.49921E-03
100000	3.14195E+00	4.11195E-01	3.14163E+00	6.36837E-03
1000000	3.14003E+00	4.14114E-01	3.14128E+00	6.39217E-03
10000000	3.14213E+00	4.13838E-01	3.14160E+00	6.40784E-03

However, it is unfair to study one-dimensional integrals with MC methods!

Multidimensional Integrals

When we deal with multidimensional integrals of the form

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \dots \int_0^1 dx_d g(x_1, \dots, x_d),$$

with x_i defined in the interval $[a_i, b_i]$ we would typically need a transformation of variables of the form

$$x_i = a_i + (b_i - a_i)t_i,$$

if we were to use the uniform distribution on the interval $[0, 1]$. In this case, we need a Jacobi determinant (useful in project 3)

$$\prod_{i=1}^{d}(b_i - a_i),$$

and to convert the function $g(x_1, \dots, x_d)$ to

$$g(x_1, \dots, x_d) \rightarrow g(a_1 + (b_1 - a_1)t_1, \dots, a_d + (b_d - a_d)t_d).$$

Optimization and profiling, useful for project 3

Till now we have not paid much attention to speed and possible optimization possibilities inherent in the various compilers. We have compiled and linked as

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

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 to 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

```
man c++  >  out_to_file
```

Optimization and profiling

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 is the recommended option. **But you must check that you get the same results as previously**.

Optimization and profiling

It is also useful to profile your program under the development stage. You would then compile with (Mac and unix/linux)

```
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 >  out_to_profile
```

When you have profiled properly your code, you must take out this option as it increases your CPU expenditure.

Optimization and profiling

Other hints

- avoid if tests or call to functions inside loops, if possible.
- avoid multiplication with constants inside loops if possible

Bad code

```
for i = 1:n
    a(i) = b(i) +c*d
    e = g(k)
end
```

Better code

```
temp = c*d
for i = 1:n
    a(i) = b(i) + temp
end
e = g(k)
```

Example: 6-dimensional Integral

Consider the following six-dimensional integral

$$\int_{-\infty}^{\infty} \mathbf{dxdy} g(\mathbf{x}, \mathbf{y}),$$

where

$$g(\mathbf{x}, \mathbf{y}) = \exp\left(-\mathbf{x}^2 - \mathbf{y}^2 - (\mathbf{x} - \mathbf{y})^2/2\right),$$

with $d = 6$.

Example: 6-dimensional Integral

We can solve this integral by employing our brute force scheme, or using importance sampling and random variables distributed according to a gaussian PDF. For the latter, if we set the mean value $\mu = 0$ and the standard deviation $\sigma = 1/\sqrt{2}$, we have

$$\frac{1}{\sqrt{\pi}} \exp(-x^2),$$

and through

$$\pi^3 \int \prod_{i=1}^{6} \left(\frac{1}{\sqrt{\pi}} \exp(-x_i^2) \right) \exp(-(\mathbf{x}-\mathbf{y})^2/2) dx_1 \dots dx_6,$$

we can rewrite our integral as

$$\int f(x_1, \dots, x_d) F(x_1, \dots, x_d) \prod_{i=1}^{6} dx_i,$$

where f is the gaussian distribution.

Brute Force I

```
.....
//   evaluate the integral without importance sampling
//   Loop over Monte Carlo Cycles
     for ( int i = 1;  i <= n; i++){
//   x[] contains the random numbers for all dimensions
       for (int j = 0; j< 6; j++) {
           x[j]=-length+2*length*ran0(&idum);
       }
       fx=brute_force_MC(x);
       int_mc += fx;
       sum_sigma += fx*fx;
     }
     int_mc = int_mc/((double) n );
     sum_sigma = sum_sigma/((double) n );
     variance=sum_sigma-int_mc*int_mc;
......
```

Brute Force II

```
double  brute_force_MC(double *x)
{
   double a = 1.; double b = 0.5;
// evaluate the different terms of the exponential
   double xx=x[0]*x[0]+x[1]*x[1]+x[2]*x[2];
   double yy=x[3]*x[3]+x[4]*x[4]+x[5]*x[5];
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-a*xx-a*yy-b*xy);
```

Importance Sampling I

```
..........
//   evaluate the integral with importance sampling
     for ( int i = 1;  i <= n; i++){
//   x[] contains the random numbers for all dimensions
       for (int j = 0; j < 6; j++) {
 x[j] = gaussian_deviate(&idum)*sqrt2;
       }
       fx=gaussian_MC(x);
       int_mc += fx;
       sum_sigma += fx*fx;
     }
     int_mc = int_mc/((double) n );
     sum_sigma = sum_sigma/((double) n );
     variance=sum_sigma-int_mc*int_mc;
.............
```

Importance Sampling II

```
// this function defines the integrand to integrate

double  gaussian_MC(double *x)
{
   double a = 0.5;
// evaluate the different terms of the exponential
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-a*xy);
} // end function for the integrand
```

Test Runs for six-dimensional Integral

Results for as function of number of Monte Carlo samples N. The exact answer is $I \approx 10.9626$ for the integral. The suffix *cr* stands for the brute force approach while *gd* stands for the use of a Gaussian distribution function. All calculations use ran0 as function to generate the uniform distribution.

N	I_{cr}	I_{gd}
10000	1.15247E+01	1.09128E+01
100000	1.29650E+01	1.09522E+01
1000000	1.18226E+01	1.09673E+01
10000000	1.04925E+01	1.09612E+01

Example, six-dimensional integral (project 3 2011)

The task of this project is to integrate in a brute force manner a six-dimensional integral which is used to determine the ground state correlation energy between two electrons in a helium atom. We will employ both Gaussian quadrature and Monte-Carlo integration. We assume that the wave function of each electron can be modelled like the single-particle wave function of an electron in the hydrogen atom. The single-particle wave function for an electron i in the $1s$ state is given in terms of a dimensionless variable (the wave function is not properly normalized)

$$\mathbf{r}_i = x_i\mathbf{e}_x + y_i\mathbf{e}_y + z_i\mathbf{e}_z,$$

as

$$\psi_{1s}(\mathbf{r}_i) = e^{-\alpha r_i},$$

where α is a parameter and

$$r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}.$$

We will fix $\alpha = 2$, which should correspond to the charge of the helium atom $Z = 2$.

Switch to spherical coordinates

Useful to change to spherical coordinates

$$d\mathbf{r}_1 d\mathbf{r}_2 = r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2,$$

and

$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 cos(\beta)}}$$

with

$$\cos(\beta) = \cos(\theta_1)\cos(\theta_2) + \sin(\theta_1)\sin(\theta_2)\cos(\phi_1 - \phi_2))$$

How do I do importance sampling in spherical coordinates

- $r_{1,2} \in [0,\infty)$, here we use the mapping $r_{1,2} = -ln(1-ran)$ with $ran \in [0,1]$, a uniform distribution point.
- $\theta_{1,2} \in [0,\pi]$, use mapping $\theta_{1,2} = \pi * ran$ with $ran \in [0,1]$ a uniform distribution point.
- $\phi_{1,2} \in [0,2\pi]$, use mapping $\phi_{1,2} = 2\pi * ran$ with $ran \in [0,1]$ a uniform distribution point.
- Be careful with the integrand

$$\frac{\exp\left(-4(r_1+r_2)\right) r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2}{\sqrt{r_1^2+r_2^2-2r_1r_2cos(\beta)}}$$

Example, six-dimensional integral (project 3 2011)

The ansatz for the wave function for two electrons is then given by the product of two $1s$ wave functions as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1+r_2)}.$$

Note that it is not possible to find an analytic solution to Schrödinger's equation for two interacting electrons in the helium atom.
The integral we need to solve is the quantum mechanical expectation value of the correlation energy between two electrons, namely

$$\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} = \frac{5\pi^2}{16^2} = 0.192765711. \qquad (42)$$

Note that our wave function is not normalized. There is a normalization factor missing, but for this project we don't need to worry about that.

Example, six-dimensional integral (project 3 2011)

a-b) Use Gaussian quadrature and compute the integral by integrating for each variable $x_1, y_1, z_1, x_2, y_2, z_2$ from $-\infty$ to ∞. How many mesh points do you need before the results converges at the level of the fourth leading digit? Hint: the single-particle wave function $e^{-\alpha r_i}$ is more or less zero at $r_i \approx ?$. You can therefore replace the integration limits $-\infty$ and ∞ with $-\Lambda$ and Λ, respectively. You need to check that this approximation is satisfactory.

c) Compute the same integral but now with brute force Monte Carlo and compare your results with those from the previous point. Discuss the differences. With bruce force we mean that you should use the uniform distribution.

d) Improve your brute force Monte Carlo calculation by using importance sampling. Hint: use the exponential distribution. Does the variance decrease? Does the CPU time used compared with the brute force Monte Carlo decrease in order to achieve the same accuracy? Comment your results.

Example, six-dimensional integral, Gauss-Legendre

```
     double *x = new double [N];
     double *w = new double [N];
//   set up the mesh points and weights
     gauleg(a,b,x,w, N);

//   evaluate the integral with the Gauss-Legendre method
//   Note that we initialize the sum
     double int_gauss = 0.;
     for (int i=0;i<N;i++){
     for (int j = 0;j<N;j++){
     for (int k = 0;k<N;k++){
     for (int l = 0;l<N;l++){
     for (int m = 0;m<N;m++){
     for (int n = 0;n<N;n++){
        int_gauss+=w[i]*w[j]*w[k]*w[l]*w[m]*w[n]
       *int_function(x[i],x[j],x[k],x[l],x[m],x[n]);
      }}}}}
}
```

Example, six-dimensional integral, Gauss-Legendre

```
//  this function defines the function to integrate
double int_function(double x1, double y1, double z1,
                    double x2, double y2, double z2)
{
   double alpha = 2.;
// evaluate the different terms of the exponential
   double exp1=-2*alpha*sqrt(x1*x1+y1*y1+z1*z1);
   double exp2=-2*alpha*sqrt(x2*x2+y2*y2+z2*z2);
   double deno=sqrt(pow((x1-x2),2)+pow((y1-y2),2)+pow((z1-z2),2));
  if(deno <pow(10.,-6.)) { return 0;}
  else return exp(exp1+exp2)/deno;
} // end of function to evaluate
```

Example, six-dimensional integral, brute force MC

```
      double int_mc = 0.;  double variance = 0.;
      double sum_sigma= 0. ; long idum=-1 ;
      double length=1.5; // we fix the max size of the box to L=3
      double volume=pow((2*length),6.);

 //   evaluate the integral with importance sampling
      for ( int i = 1;  i <= n; i++){
 //   x[] contains the random numbers for all dimensions
        for (int j = 0; j< 6; j++) {
            // Maps U[0,1] to U[-L,L]
            x[j]=-length+2*length*ran0(&idum);
        }
        fx=brute_force_MC(x);
        int_mc += fx;
        sum_sigma += fx*fx;
      }
      int_mc = int_mc/((double) n );
      sum_sigma = sum_sigma/((double) n );
      variance=sum_sigma-int_mc*int_mc;
      ....
```

Example, six-dimensional integral, brute force MC

```
double brute_force_MC(double *x)
{
   double alpha = 2.;
// evaluate the different terms of the exponential
   double exp1=-2*alpha*sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
   double exp2=-2*alpha*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5]);
   double deno=sqrt(pow((x[0]-x[3]),2)
          +pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2));
   double value=exp(exp1+exp2)/deno;
return value;
} // end function for the integrand
```

Example, six-dimensional integral, importance sampling

```
double int_mc = 0.;  double variance = 0.;
double sum_sigma= 0. ; long idum=-1 ;
// The 'volume' contains 4 jacobideterminants(pi,pi,2pi,2pi)
// and a scaling factor 1/16
double volume=4*pow(acos(-1.),4.)*1./16;
//   evaluate the integral with importance sampling
for ( int i = 1;  i <= n; i++){
   for (int j = 0; j < 2; j++) {
y=ran0(&idum);
x[j]=-0.25*log(1.-y);
     }
   for (int j = 2; j < 4; j++) {
x[j] = 2*acos(-1.)*ran0(&idum);
   }
   for (int j = 4; j < 6; j++) {
 x[j] = acos(-1.)*ran0(&idum);
      }
fx=integrand_MC(x);
        ....
```

Example, six-dimensional integral, importance sampling

```
// this function defines the integrand to integrate

double  integrand_MC(double *x)
{
double num=x[0]*x[0]*x[1]*x[1]*sin(x[4])*sin(x[5]);
double deno=sqrt(x[0]*x[0]+x[1]*x[1]-2*x[0]*x[1]*
(sin(x[4])*sin(x[5])*cos(x[2]-x[3])+cos(x[4])*cos(x[5])));
return num/deno;
} // end function for the integrand
```

Test Runs and Comparison with Brute Force and Gauss-Legendre

The suffix *br* stands for the brute force approach while *is* stands for the use of importance sampling.

N	I_{br}	σ_{br}	time(s)	I_{is}	σ_{is}	time(s)
1E6	0.19238	3.85124E-4	0.6	0.19176	1.01515E-4	1.4
10E6	0.18607	1.18053E-4	6	0.192254	1.22430E-4	14
100E6	0.18846	4.37163E-4	57	0.192720	1.03346E-4	138
1000E6	0.18843	1.35879E-4	581	0.192789	3.28795E-5	1372

Gauss-Legendre results:

N	time(s)	I_n	$\|I_n - I$
20	31	0.18047	1.123E-2
30	354	0.18501	7.76E-3
40	1999	0.18653	6.24E-3
50	7578	0.18722	5.54E-3

Random Numbers, chapter 11.3

Random Numbers

Most used are so-called 'Linear congruential'

$$N_i = (aN_{i-1} + c)MOD(M),$$

and to find a number in $x \in [0, 1]$

$$x_i = N_i/M$$

M is called the period and should be as big as possible. The start value is N_0 and is called the seed.

- The random variables should result in the uniform distribution
- No correlations between numbers (zero covariance)
- As big as possible period M
- Fast algo

Random Numbers

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most M. If however the parameters a and c are badly chosen, the period may be even shorter.
Consider the following example

$$N_i = (6N_{i-1} + 7)\mathrm{MOD}(5),$$

with a seed $N_0 = 2$. This generator produces the sequence $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, ... \ldots,$ i.e., a sequence with period 5. However, increasing M may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\mathrm{MOD}(54),$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \ldots,$ a period of just 2.

Random Numbers

Typical periods for the random generators provided in the program library are of the order of $\sim 10^9$ or larger. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could make a shift register generator whose lth number is the sum of the $l-i$th and $l-j$th values with modulo M,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M). \tag{43}$$

Such a generator again produces a sequence of pseudorandom numbers but this time with a period much larger than M. It is also possible to construct more elaborate algorithms by including more than two past terms in the sum of each iteration. One example is the generator of Marsaglia and Zaman (Computers in Physics **8** (1994) 117) which consists of two congruential relations

$$N_l = (N_{l-3} - N_{l-1})\mathrm{MOD}(2^{31} - 69),$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\mathrm{MOD}(2^{32}),$$

which according to the authors has a period larger than 2^{94}.

Random Numbers

Using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j}) \tag{44}$$

where the bitwise action of $\oplus$ means that if $N_{l-i} = N_{l-j}$ the result is 0 whereas if $N_{l-i} \neq N_{l-j}$ the result is 1. As an example, consider the case where $N_{l-i} = 6$ and $N_{l-j} = 11$. The first one has a bit representation (using 4 bits only) which reads 0110 whereas the second number is 1011. Employing the $\oplus$ operator yields 1101, or $2^3 + 2^2 + 2^0 = 13$.

In Fortran90, the bitwise $\oplus$ operation is coded through the intrinsic function $\mathrm{IEOR}(m, n)$ where m and n are the input numbers, while in C it is given by $m \wedge n$.

Random Numbers

The function *ran*0 implements

$$N_i = (aN_{i-1})\mathrm{MOD}(M).$$

Note that $c = 0$ and that it cannot be initialized with $N_0 = 0$. However, since a and N_{i-1} are integers and their multiplication could become greater than the standard 32 bit integer, there is a trick via Schrage's algorithm which approximates the multiplication of large integers through the factorization

$$M = aq + r,$$

where we have defined

$$q = [M/a],$$

and

$$r = M \ \mathrm{MOD} \ a.$$

where the brackets denote integer division. In the code below the numbers q and r are chosen so that $r < q$.

Random Numbers

To see how this works we note first that

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\mathrm{MOD}(M),$$

since we can add or subtract any integer multiple of M from aN_{i-1}. The last term $[N_{i-1}/q]M\mathrm{MOD}(M)$ is zero since the integer division $[N_{i-1}/q]$ just yields a constant which is multiplied with M. Rewrite as

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq + r))\mathrm{MOD}(M),$$

Random Numbers

It gives

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r))\,\mathrm{MOD}(M),$$

yielding

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1}\mathrm{MOD}(q)) - [N_{i-1}/q]r))\,\mathrm{MOD}(M).$$

- $[N_{i-1}/q]r$ is always smaller or equal $N_{i-1}(r/q)$ and with $r < q$ we obtain always a number smaller than N_{i-1}, which is smaller than M.
- $N_{i-1}\mathrm{MOD}(q)$ is between zero and $q-1$ then $a(N_{i-1}\mathrm{MOD}(q)) < aq$.
- Our definition of $q = [M/a]$ ensures that this term is also smaller than M meaning that both terms fit into a 32-bit signed integer. None of these two terms can be negative, but their difference could.

Random Numbers

```
/*  ran0() is an "Minimal" random number generator of Park and Miller
** Set or reset the input value
** idum to any integer value (except the unlikely value MASK)
** to initialize the sequence; idum must not be altered between
** calls for sucessive deviates in a sequence.
** The function returns a uniform deviate between 0.0 and 1.0.
*/
double ran0(long &idum)
{
   const int a = 16807, m = 2147483647, q = 127773;
   const int r = 2836, MASK = 123459876;
   const double am = 1./m;
   long     k;
   double   ans;
   idum ^= MASK;
   k = (*idum)/q;
   idum = a*(idum - k*q) - r*k;
   // add m if negative difference
   if(idum < 0) idum += m;
   ans=am*(idum);
   idum ^= MASK;
   return ans;
} // End: function ran0()
```

Random Numbers

Important tests of random numbers are the standard deviation σ and the mean $\mu = \langle x \rangle$.
For the uniform distribution we have

$$\langle x^k \rangle = \int_0^1 dx p(x) x^k = \int_0^1 dx x^k = \frac{1}{k+1},$$

since $p(x) = 1$. The mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886.$$

Random Numbers

Number of x-values for various intervals generated by 4 random number generators, their corresponding mean values and standard deviations. All calculations have been initialized with the variable $idum = -1$.

x-bin	ran0	ran1	ran2	ran3
0.0-0.1	1013	991	938	1047
0.1-0.2	1002	1009	1040	1030
0.2-0.3	989	999	1030	993
0.3-0.4	939	960	1023	937
0.4-0.5	1038	1001	1002	992
0.5-0.6	1037	1047	1009	1009
0.6-0.7	1005	989	1003	989
0.7-0.8	986	962	985	954
0.8-0.9	1000	1027	1009	1023
0.9-1.0	991	1015	961	1026
μ	0.4997	0.5018	0.4992	0.4990
σ	0.2882	0.2892	0.2861	0.2915

Variance, covariance, errors etc, chapter 11.2

Statistical analysis

- Monte Carlo simulations can be treated as *computer experiments*
- The results can be analysed with the same statistics tools we would use in analysing laboraty experiments
- As in all other experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., the error

Variance, covariance, errors etc

Statistical analysis

- As in other experiments, Monte Carlo experiments have two classes of errors:
  - Statistical errors
  - Systematic errors
- Statistical errors can be estimated using standard tools from statistics
- Systematic errors are method specific and must be treated differently from case to case.

Variance, covariance, errors etc

A *stochastic process* is a process that produces sequentially a chain of values:

$$\{x_1, x_2, \dots x_k, \dots\}.$$

We will call these values our *measurements* and the entire set as our measured *sample*. The action of measuring all the elements of a sample we will call a stochastic *experiment* (since, operationally, they are often associated with results of empirical observation of some physical or mathematical phenomena; precisely an experiment). We assume that these values are distributed according to some PDF $p_X(x)$, where X is just the formal symbol for the stochastic variable whose PDF is $p_X(x)$. Instead of trying to determine the full distribution p we are often only interested in finding the few lowest moments, like the mean μ_X and the variance σ_X.

Variance, covariance, errors etc

The *probability distribution function (PDF)* is a function $p(x)$ on the domain which, in the discrete case, gives us the probability or relative frequency with which these values of X occur:

$$p(x) = \mathrm{Prob}(X = x)$$

In the continuous case, the PDF does not directly depict the actual probability. Instead we define the probability for the stochastic variable to assume any value on an infinitesimal interval around x to be $p(x)dx$. The continuous function $p(x)$ then gives us the *density* of the probability rather than the probability itself. The probability for a stochastic variable to assume any value on a non-infinitesimal interval $[a, b]$ is then just the integral:

$$\mathrm{Prob}(a \leq X \leq b) = \int_a^b p(x)dx$$

Qualitatively speaking, a stochastic variable represents the values of numbers chosen as if by chance from some specified PDF so that the selection of a large set of these numbers reproduces this PDF.

Variance, covariance, errors etc

Also of interest to us is the *cumulative probability distribution function (CDF)*, $P(x)$, which is just the probability for a stochastic variable X to assume any value less than x:

$$P(x) = \mathrm{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'$$

The relation between a CDF and its corresponding PDF is then:

$$p(x) = \frac{d}{dx}P(x)$$

Variance, covariance, errors etc

A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows:

$$\langle x^n \rangle \equiv \int x^n p(x)\, dx$$

The zero-th moment $\langle 1 \rangle$ is just the normalization condition of p. The first moment, $\langle x \rangle$, is called the *mean* of p and often denoted by the letter μ:

$$\langle x \rangle = \mu \equiv \int x p(x)\, dx$$

Variance, covariance, errors etc

A special version of the moments is the set of *central moments*, the n-th central moment defined as:

$$\langle (x - \langle x \rangle)^n \rangle \equiv \int (x - \langle x \rangle)^n p(x)\, dx$$

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{Var}(X)$:

$$\begin{aligned}
\sigma_X^2 = \mathrm{Var}(X) &= \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x)\, dx \\
&= \int \left(x^2 - 2x\langle x \rangle + \langle x \rangle^2 \right) p(x)\, dx \\
&= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 \\
&= \langle x^2 \rangle - \langle x \rangle^2
\end{aligned}$$

The square root of the variance, $\sigma = \sqrt{\langle (x - \langle x \rangle)^2 \rangle}$ is called the *standard deviation* of p. It is clearly just the RMS (root-mean-square) value of the deviation of the PDF from its mean value, interpreted qualitatively as the "spread" of p around its mean.

Variance, covariance, errors etc

Another important quantity is the so called covariance, a variant of the above defined variance. Consider again the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1, \dots, x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows:

$$\begin{aligned} \mathrm{Cov}(X_i,\, X_j) &\equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle)\, P(x_1, \dots, x_n)\, dx_1 \dots dx_n \end{aligned} \tag{45}$$

with

$$\langle x_i \rangle = \int \cdots \int x_i\, P(x_1, \dots, x_n)\, dx_1 \dots dx_n$$

Variance, covariance, errors etc

If we consider the above covariance as a matrix $C_{ij} = \mathrm{Cov}(X_i,\, X_j)$, then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{Cov}(X_i,\, X_i) = \mathrm{Var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated. This is easy to show, keeping in mind the linearity of the expectation value. Consider the stochastic variables X_i and X_j, $(i \neq j)$:

$$
\begin{aligned}
\mathrm{Cov}(X_i,\, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\
&= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle \\
&= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle \\
&= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle \\
&= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle
\end{aligned}
$$

Variance, covariance, errors etc

If X_i and X_j are independent, we get $\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle$, resulting in $\mathrm{Cov}(X_i, X_j) = 0 \ \ (i \neq j)$.
Also useful for us is the covariance of linear combinations of stochastic variables. Let $\{X_i\}$ and $\{Y_i\}$ be two sets of stochastic variables. Let also $\{a_i\}$ and $\{b_i\}$ be two sets of scalars. Consider the linear combination:

$$U = \sum_i a_i X_i \qquad V = \sum_j b_j Y_j$$

By the linearity of the expectation value

$$\mathrm{Cov}(U, V) = \sum_{i,j} a_i b_j \mathrm{Cov}(X_i, Y_j)$$

Variance, covariance, errors etc

Now, since the variance is just $\mathrm{Var}(X_i) = \mathrm{Cov}(X_i, X_i)$, we get the variance of the linear combination $U = \sum_i a_i X_i$:

$$\mathrm{Var}(U) = \sum_{i,j} a_i a_j \mathrm{Cov}(X_i, X_j) \quad (46)$$

And in the special case when the stochastic variables are uncorrelated, the off-diagonal elements of the covariance are as we know zero, resulting in:

$$\mathrm{Var}(U) = \sum_i a_i^2 \mathrm{Cov}(X_i, X_i) = \sum_i a_i^2 \mathrm{Var}(X_i)$$

$$\mathrm{Var}(\sum_i a_i X_i) = \sum_i a_i^2 \mathrm{Var}(X_i)$$

which will become very useful in our study of the error in the mean value of a set of measurements.

Variance, covariance, errors etc

In practical situations a sample is always of finite size. Let that size be n. The expectation value of a sample, the *sample mean*, is then defined as follows:

$$\bar{x}_n \equiv \frac{1}{n} \sum_{k=1}^{n} x_k$$

The *sample variance* is:

$$\mathrm{Var}(x) \equiv \frac{1}{n} \sum_{k=1}^{n} (x_k - \bar{x}_n)^2$$

its square root being the *standard deviation of the sample*. The *sample covariance* is:

$$\mathrm{Cov}(x) \equiv \frac{1}{n} \sum_{kl} (x_k - \bar{x}_n)(x_l - \bar{x}_n)$$

Variance, covariance, errors etc

Note that the sample variance is the sample covariance without the cross terms. In a similar manner as the covariance in eq. (45) is a measure of the correlation between two stochastic variables, the above defined sample covariance is a measure of the sequential correlation between succeeding measurements of a sample.

These quantities, being known experimental values, differ significantly from and must not be confused with the similarly named quantities for stochastic variables, mean μ_X, variance $\mathrm{Var}(X)$ and covariance $\mathrm{Cov}(X, Y)$.

Variance, covariance, errors etc

The law of large numbers states that as the size of our sample grows to infinity, the sample mean approaches the true mean μ_X of the chosen PDF:

$$\lim_{n\to\infty} \bar{x}_n = \mu_X$$

The sample mean $\bar{x}_n$ works therefore as an estimate of the true mean μ_X.
What we need to find out is how good an approximation $\bar{x}_n$ is to μ_X. In any stochastic measurement, an estimated mean is of no use to us without a measure of its error. A quantity that tells us how well we can reproduce it in another experiment. We are therefore interested in the PDF of the sample mean itself. Its standard deviation will be a measure of the spread of sample means, and we will simply call it the *error* of the sample mean, or just sample error, and denote it by err_X. In practice, we will only be able to produce an *estimate* of the sample error since the exact value would require the knowledge of the true PDFs behind, which we usually do not have.

Variance, covariance, errors etc

The straight forward brute force way of estimating the sample error is simply by producing a number of samples, and treating the mean of each as a measurement. The standard deviation of these means will then be an estimate of the original sample error. If we are unable to produce more than one sample, we can split it up sequentially into smaller ones, treating each in the same way as above. This procedure is known as *blocking* and will be given more attention shortly. At this point it is worth while exploring more indirect methods of estimation that will help us understand some important underlying principles of correlational effects.

Variance, covariance, errors etc

Let us first take a look at what happens to the sample error as the size of the sample grows. In a sample, each of the measurements x_i can be associated with its own stochastic variable X_i. The stochastic variable $\overline{X}_n$ for the sample mean $\bar{x}_n$ is then just a linear combination, already familiar to us:

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

All the coefficients are just equal $1/n$. The PDF of $\overline{X}_n$, denoted by $p_{\overline{X}_n}(x)$ is the desired PDF of the sample means.

Variance, covariance, errors etc

The probability density of obtaining a sample mean $\bar{x}_n$ is the product of probabilities of obtaining arbitrary values $x_1, x_2, \ldots, x_n$ with the constraint that the mean of the set $\{x_i\}$ is $\bar{x}_n$:

$$p_{\overline{X}_n}(x) = \int p_X(x_1) \cdots \int p_X(x_n)\, \delta\left(x - \frac{x_1 + x_2 + \cdots + x_n}{n}\right) dx_n \cdots dx_1$$

And in particular we are interested in its variance $\mathrm{Var}(\overline{X}_n)$.

Variance, covariance, errors etc

It is generally not possible to express $p_{\overline{X}_n}(x)$ in a closed form given an arbitrary PDF p_X and a number n. But for the limit $n \to \infty$ it is possible to make an approximation. The very important result is called *the central limit theorem*. It tells us that as n goes to infinity, $p_{\overline{X}_n}(x)$ approaches a Gaussian distribution whose mean and variance equal the true mean and variance, μ_X and σ_X^2, respectively:

$$\lim_{n\to\infty} p_{\overline{X}_n}(x) = \left(\frac{n}{2\pi \mathrm{Var}(X)}\right)^{1/2} e^{-\frac{n(x-\bar{x}_n)^2}{2\mathrm{Var}(X)}} \tag{47}$$

Variance, covariance, errors etc

The desired variance $\mathrm{Var}(\overline{X}_n)$, i.e. the sample error squared err_X^2, is given by:

$$\mathrm{err}_X^2 = \mathrm{Var}(\overline{X}_n) = \frac{1}{n^2} \sum_{ij} \mathrm{Cov}(X_i, X_j) \tag{48}$$

We see now that in order to calculate the exact error of the sample with the above expression, we would need the true means μ_{X_i} of the stochastic variables X_i. To calculate these requires that we know the true multivariate PDF of all the X_i. But this PDF is unknown to us, we have only got the measurements of one sample. The best we can do is to let the sample itself be an estimate of the PDF of each of the X_i, estimating all properties of X_i through the measurements of the sample.

Variance, covariance, errors etc

Our estimate of μ_{X_i} is then the sample mean $\bar{x}$ itself, in accordance with the the central limit theorem:

$$\mu_{X_i} = \langle x_i \rangle \approx \frac{1}{n} \sum_{k=1}^{n} x_k = \bar{x}$$

Using $\bar{x}$ in place of μ_{X_i} we can give an *estimate* of the covariance in eq. (48):

$$\begin{aligned}
\mathrm{Cov}(X_i, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \approx \langle (x_i - \bar{x})(x_j - \bar{x}) \rangle \\
&\approx \frac{1}{n} \sum_{l}^{n} \left(\frac{1}{n} \sum_{k}^{n} (x_k - \bar{x}_n)(x_l - \bar{x}_n) \right) = \frac{1}{n} \frac{1}{n} \sum_{kl} (x_k - \bar{x}_n)(x_l - \bar{x}_n) \\
&= \frac{1}{n} \mathrm{Cov}(x)
\end{aligned}$$

Variance, covariance, errors etc

By the same procedure we can use the sample variance as an estimate of the variance of any of the stochastic variables X_i:

$$\begin{aligned} \mathrm{Var}(X_i) &= \langle x_i - \langle x_i \rangle \rangle \approx \langle x_i - \bar{x}_n \rangle \\ &\approx \frac{1}{n} \sum_{k=1}^{n} (x_k - \bar{x}_n) \\ &= \mathrm{Var}(x) \end{aligned} \tag{49}$$

Now we can calculate an estimate of the error err_X of the sample mean $\bar{x}_n$:

$$\begin{aligned} \mathrm{err}_X^2 &= \frac{1}{n^2} \sum_{ij} \mathrm{Cov}(X_i, X_j) \\ &\approx \frac{1}{n^2} \sum_{ij} \frac{1}{n} \mathrm{Cov}(x) = \frac{1}{n^2} n^2 \frac{1}{n} \mathrm{Cov}(x) \\ &= \frac{1}{n} \mathrm{Cov}(x) \end{aligned} \tag{50}$$

which is nothing but the sample covariance divided by the number of measurements in the sample.

Variance, covariance, errors etc

In the special case that the measurements of the sample are uncorrelated (equivalently the stochastic variables X_i are uncorrelated) we have that the off-diagonal elements of the covariance are zero. This gives the following estimate of the sample error:

$$
\begin{aligned}
\mathrm{err}_X^2 &= \frac{1}{n^2}\sum_{ij}\mathrm{Cov}(X_i, X_j) = \frac{1}{n^2}\sum_{i}\mathrm{Var}(X_i) \\
&\approx \frac{1}{n^2}\sum_{i}\mathrm{Var}(x) \\
&= \frac{1}{n}\mathrm{Var}(x) \qquad (51)
\end{aligned}
$$

where in the second step we have used eq. (49). The error of the sample is then just its standard deviation divided by the square root of the number of measurements the sample contains. This is a very useful formula which is easy to compute. It acts as a first approximation to the error, but in numerical experiments, we cannot overlook the always present correlations.

Variance, covariance, errors etc

For computational purposes one usually splits up the estimate of err_X^2, given by eq. (50), into two parts:

$$\begin{aligned}\mathrm{err}_X^2 &= \frac{1}{n}\mathrm{Var}(x) + \frac{1}{n}(\mathrm{Cov}(x) - \mathrm{Var}(x)) \\ &= \frac{1}{n^2}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2 + \frac{2}{n^2}\sum_{k<l}(x_k - \bar{x}_n)(x_l - \bar{x}_n) \qquad (52)\end{aligned}$$

The first term is the same as the error in the uncorrelated case, eq. (51). This means that the second term accounts for the error correction due to correlation between the measurements. For uncorrelated measurements this second term is zero.

Variance, covariance, errors etc

Computationally the uncorrelated first term is much easier to treat efficiently than the second.

$$\mathrm{Var}(x) = \frac{1}{n}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2 = \left(\frac{1}{n}\sum_{k=1}^{n} x_k^2\right) - \bar{x}_n^2$$

We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

Week 42

Monte Carlo methods

- Monday: Repetition from last week
- Central limit theorem and finalization of discussion on variance, covariance and standard deviation (first lecture)
- Random walks and Brownian motion
- Diffusion and Master equation
- Wednesday:
- Detailed balance and the Metropolis(-Hastings) algorithm
- Examples of the use of the Metropolis(-Hastings) algorithm: Maxwell-Boltzmann distribution.

Chapter 12 of lecture notes. Focus on 12.2, 12.4 and 12.5. Typo hunting: reward for the one who finds most typos in the compendium (giftcard from Akademika). Deadline, November 30.

Why Markov Chains?

- We want to study a physical system which evolves towards equilibrium, from given initial conditions.
- We start with a PDF $w(x_0, t_0)$ and we want to understand how it evolves with time.
- We want to reach a situation where after a given number of time steps we obtain a steady state. This means that the system reaches its most likely state (equilibrium situation)
- Our PDF is normally a multidimensional object whose normalization constant is impossible to find.
- Analytical calculations from $w(x, t)$ are not possible.
- To sample directly from from $w(x, t)$ is not possible/difficult.
- The transition probability W is also not known.
- How can we establish that we have reached a steady state? Sounds impossible! Use Markov chain Monte Carlo

Brownian Motion and Markov Processes

A Markov process is a random walk with a selected probability for making a move. The new move is independent of the previous history of the system. The Markov process is used repeatedly in Monte Carlo simulations in order to generate new random states. The reason for choosing a Markov process is that when it is run for a long enough time starting with a random state, we will eventually reach the most likely state of the system. In thermodynamics, this means that after a certain number of Markov processes we reach an equilibrium distribution.

This mimicks the way a real system reaches its most likely state at a given temperature of the surroundings.

Brownian Motion and Markov Processes

To reach this distribution, the Markov process needs to obey two important conditions, that of **ergodicity** and **detailed balance**. These conditions impose then constraints on our algorithms for accepting or rejecting new random states.

The Metropolis algorithm discussed here abides to both these constraints. The Metropolis algorithm is widely used in Monte Carlo simulations and the understanding of it rests within the interpretation of random walks and Markov processes.

Brownian Motion and Markov Processes

In a random walk one defines a mathematical entity called a **walker**, whose attributes completely define the state of the system in question. The state of the system can refer to any physical quantities, from the vibrational state of a molecule specified by a set of quantum numbers, to the brands of coffee in your favourite supermarket.

The walker moves in an appropriate state space by a combination of deterministic and random displacements from its previous position.
This sequence of steps forms a **chain**.

Sequence of ingredients

- We want to study a physical system which evolves towards equilibrium, from given initial conditions.
- Markov chains are intimately linked with the physical process of diffusion. Proof in lecture notes.
- From a Markov chain we can then derive the conditions for detailed balance and ergodicity. These are the conditions needed for obtaining a steady state.
- The widely used algorithm for doing this is the so-called Metropolis algorithm, in its refined form the Metropolis-Hastings algorithm.

Applications: almost every field

- Financial engineering, see for example Patriarca *et al*, Physica **340**, page 334 (2004).
- Neuroscience, see for example Lipinski, Physics Medical Biology **35**, page 441 (1990) or Farnell and Gibson, Journal of Computational Physics **208**, page 253 (2005)
- Tons of applications in physics
- and chemistry
- and biology, medicine
- Nobel prize in economy to Black and Scholes

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0.$$

  The BlackâScholes equation is a partial differential equation, which describes the price of the option over time. The derivation is based on Brownian motion (Langevin and Fokker-Planck, 12.6)

- ...the list is almost infinite

A simple Example

The obvious case is that of a random walker on a one-, or two- or three-dimensional lattice (dubbed coordinate space hereafter)
Consider a system whose energy is defined by the orientation of single spins. Consider the state i, with given energy E_i represented by the following N spins

$$\begin{array}{ccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

We may be interested in the transition with one single spinflip to a new state j with energy E_j

$$\begin{array}{ccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

This change from one microstate i (or spin configuration) to another microstate j is the **configuration space** analogue to a random walk on a lattice. Instead of jumping from one place to another in space, we 'jump' from one microstate to another.

Diffusion from Markov Chain

From experiment there are strong indications that the flux of particles $j(x,t)$, viz., the number of particles passing x at a time t is proportional to the gradient of $w(x,t)$. This proportionality is expressed mathematically through

$$j(x,t) = -D\frac{\partial w(x,t)}{\partial x}, \tag{53}$$

where D is the so-called diffusion constant, with dimensionality length^2 per time. If the number of particles is conserved, we have the continuity equation

$$\frac{\partial j(x,t)}{\partial x} = -\frac{\partial w(x,t)}{\partial t}, \tag{54}$$

which leads to

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2}, \tag{55}$$

which is the diffusion equation in one dimension. Solved as a partial differential equation in chapter 10.

Diffusion from Markov Chain

With the probability distribution function $w(x,t)dx$ we can compute expectation values such as the mean distance

$$\langle x(t)\rangle = \int_{-\infty}^{\infty} xw(x,t)dx, \tag{56}$$

or

$$\langle x^2(t)\rangle = \int_{-\infty}^{\infty} x^2 w(x,t)dx, \tag{57}$$

which allows for the computation of the variance $\sigma^2 = \langle x^2(t)\rangle - \langle x(t)\rangle^2$. Note well that these expectation values are time-dependent. In a similar way we can also define expectation values of functions $f(x,t)$ as

$$\langle f(x,t)\rangle = \int_{-\infty}^{\infty} f(x,t)w(x,t)dx. \tag{58}$$

Diffusion from Markov Chain

Since $w(x,t)$ is now treated as a PDF, it needs to obey the same criteria as discussed in the previous chapter. However, the normalization condition

$$\int_{-\infty}^{\infty} w(x,t)dx = 1 \tag{59}$$

imposes significant constraints on $w(x,t)$. These are

$$w(x=\pm\infty,t)=0 \qquad \frac{\partial^n w(x,t)}{\partial x^n}|_{x=\pm\infty} = 0, \tag{60}$$

implying that when we study the time-derivative $\partial\langle x(t)\rangle/\partial t$, we obtain after integration by parts and using Eq. (55)

$$\frac{\partial\langle x\rangle}{\partial t} = \int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial t}dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 w(x,t)}{\partial x^2}dx, \tag{61}$$

leading to

$$\frac{\partial\langle x\rangle}{\partial t} = Dx\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - D\int_{-\infty}^{\infty} \frac{\partial w(x,t)}{\partial x}dx, \tag{62}$$

implying that

$$\frac{\partial\langle x\rangle}{\partial t} = 0. \tag{63}$$

Diffusion from Markov Chain

This means in turn that $\langle x \rangle$ is independent of time. If we choose the initial position $x(t=0)=0$, the average displacement $\langle x \rangle = 0$. If we link this discussion to a random walk in one dimension with equal probability of jumping to the left or right and with an initial position $x=0$, then our probability distribution remains centered around $\langle x \rangle = 0$ as function of time. However, the variance is not necessarily 0. Consider first

$$\frac{\partial \langle x^2 \rangle}{\partial t} = Dx^2 \frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - 2D\int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial x}dx, \tag{64}$$

where we have performed an integration by parts as we did for $\frac{\partial \langle x \rangle}{\partial t}$. A further integration by parts results in

$$\frac{\partial \langle x^2 \rangle}{\partial t} = -Dxw(x,t)|_{x=\pm\infty} + 2D\int_{-\infty}^{\infty} w(x,t)dx = 2D, \tag{65}$$

leading to

$$\langle x^2 \rangle = 2Dt, \tag{66}$$

and the variance as

$$\langle x^2 \rangle - \langle x \rangle^2 = 2Dt. \tag{67}$$

The root mean square displacement after a time t is then

$$\sqrt{\langle x^2 \rangle - \langle x \rangle^2} = \sqrt{2Dt}. \tag{68}$$

Random walks, chapter 12.2

Consider now a random walker in one dimension, with probability R of moving to the right and L for moving to the left. At $t = 0$ we place the walker at $x = 0$. The walker can then jump, with the above probabilities, either to the left or to the right for each time step. Note that in principle we could also have the possibility that the walker remains in the same position. This is not implemented in this example. Every step has length $\Delta x = l$. Time is discretized and we have a jump either to the left or to the right at every time step.

Random walks

Let us now assume that we have equal probabilities for jumping to the left or to the right, i.e., $L = R = 1/2$. The average displacement after n time steps is

$$\langle x(n) \rangle = \sum_i^n \Delta x_i = 0 \qquad \Delta x_i = \pm l,$$

since we have an equal probability of jumping either to the left or to right. The value of $\langle x(n)^2 \rangle$ is

$$\langle x(n)^2 \rangle = \left(\sum_i^n \Delta x_i \right)^2 = \sum_i^n \Delta x_i^2 + \sum_{i \neq j}^n \Delta x_i \Delta x_j = l^2 n.$$

For many enough steps the non-diagonal contribution is

$$\sum_{i \neq j}^N \Delta x_i \Delta x_j = 0,$$

since $\Delta x_{i,j} = \pm l$.

Random walks

The variance is then

$$\langle x(n)^2 \rangle - \langle x(n) \rangle^2 = l^2 n.$$

It is also rather straightforward to compute the variance for $L \neq R$. The result is

$$\langle x(n)^2 \rangle - \langle x(n) \rangle^2 = 4LRl^2 n.$$

The variable n represents the number of time steps. If we define $n = t/\Delta t$, we can then couple the variance result from a random walk in one dimension with the variance from diffusion by defining the diffusion constant as

$$D = \frac{l^2}{\Delta t}.$$

Diffusion from Markov Chain

When solving partial differential equations such as the diffusion equation numerically, the derivatives are always discretized. We can rewrite the time derivative as

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i,n+1) - w(i,n)}{\Delta t}, \tag{69}$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1,n) + w(i-1,n) - 2w(i,n)}{(\Delta x)^2}, \tag{70}$$

resulting in the discretized diffusion equation

$$\frac{w(i,n+1) - w(i,n)}{\Delta t} = D\frac{w(i+1,n) + w(i-1,n) - 2w(i,n)}{(\Delta x)^2}, \tag{71}$$

where n represents a given time step and i a step in the x-direction.

Diffusion from Markov Chain

A Markov process allows in principle for a microscopic description of Brownian motion. As with the random walk, we consider a particle which moves along the x-axis in the form of a series of jumps with step length $\Delta x = l$. Time and space are discretized and the subsequent moves are statistically independent, i.e., the new move depends only on the previous step and not on the results from earlier trials. We start at a position $x = jl = j\Delta x$ and move to a new position $x = i\Delta x$ during a step $\Delta t = \epsilon$, where $i \geq 0$ and $j \geq 0$ are integers. The original probability distribution function (PDF) of the particles is given by $w_i(t = 0)$ where i refers to a specific position on a grid, with $i = 0$ representing $x = 0$. The function $w_i(t = 0)$ is now the discretized version of $w(x, t)$. We can regard the discretized PDF as a vector.

Diffusion from Markov Chain

For the Markov process we have a transition probability from a position $x = jl$ to a position $x = il$ given by

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \left\{ \begin{array}{cc} \frac{1}{2} & |i - j| = 1 \\ 0 & \mathrm{else} \end{array} \right. \tag{72}$$

We call W_{ij} for the transition probability and we can represent it, see below, as a matrix. Our new PDF $w_i(t = \epsilon)$ is now related to the PDF at $t = 0$ through the relation

$$w_i(t = \epsilon) = W(j \rightarrow i) w_j(t = 0). \tag{73}$$

Diffusion from Markov Chain

This equation represents the discretized time-development of an original PDF. Since both W and w represent probabilities, they have to be normalized, i.e., we require that at each time step we have

$$\sum_i w_i(t) = 1, \tag{74}$$

and

$$\sum_j W(j \to i) = 1. \tag{75}$$

The further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$. Note that the probability for remaining at the same place is in general not necessarily equal zero. In our Markov process we allow only for jumps to the left or to the right.

Diffusion from Markov Chain

The time development of our initial PDF can now be represented through the action of the transition probability matrix applied n times. At a time $t_n = n\epsilon$ our initial distribution has developed into

$$w_i(t_n) = \sum_j W_{ij}(t_n) w_j(0), \tag{76}$$

and defining

$$W(il - jl, n\epsilon) = (W^n(\epsilon))_{ij} \tag{77}$$

we obtain

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0), \tag{78}$$

or in matrix form

$$w(\hat{n\epsilon}) = \hat{W}^n(\epsilon)\hat{w}(0). \tag{79}$$

Brownian Motion and Markov Processes

We wish to study the time-development of a PDF after a given number of time steps. We define our PDF by the function $w(t)$. In addition we define a transition probability W. The time development of our PDF $w(t)$, after one time-step from $t = 0$ is given by

$$w_i(t = \epsilon) = W(j \rightarrow i) w_j(t = 0).$$

Normally we don't know the form of W!! This equation represents the discretized time-development of an original PDF. We can rewrite this as a

$$w_i(t = \epsilon) = W_{ij} w_j(t = 0).$$

with the transition matrix W for a random walk left or right (cannot stay in the same position) given by

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \begin{cases} \frac{1}{2} & |i - j| = 1 \\ 0 & \text{else} \end{cases}$$

We call W_{ij} for the transition probability and we represent it as a matrix.

Brownian Motion and Markov Processes

Both W and w represent probabilities and they have to be normalized, meaning that that at each time step we have

$$\sum_i w_i(t) = 1,$$

and

$$\sum_j W(j \to i) = 1.$$

Further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$. We can thus write the action of W as

$$w_i(t+1) = \sum_j W_{ij} w_j(t),$$

or as vector-matrix relation

$$\hat{\mathbf{w}}(t+1) = \hat{\mathbf{W}}\hat{\mathbf{w}}(t),$$

and if we have that $||\hat{\mathbf{w}}(t+1) - \hat{\mathbf{w}}(t)|| \to 0$, we say that we have reached the most likely state of the system, the so-called steady state or equilibrium state. Another way of phrasing this is

$$\mathbf{w}(t=\infty) = \mathbf{W}\mathbf{w}(t=\infty).$$

Brownian Motion and Markov Processes, a simple Example

Consider the simple 3×3 matrix $\hat{W}$

$$\hat{W} = \begin{pmatrix} 1/4 & 1/8 & 2/3 \\ 3/4 & 5/8 & 0 \\ 0 & 1/4 & 1/3 \end{pmatrix},$$

and we choose our initial state as

$$\hat{w}(t=0) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

The first iteration is

$$w_i(t=\epsilon) = W(j \rightarrow i) w_j(t=0),$$

resulting in

$$\hat{w}(t=\epsilon) = \begin{pmatrix} 1/4 \\ 3/4 \\ 0 \end{pmatrix}.$$

Brownian Motion and Markov Processes, a simple Example

The next iteration results in

$$w_i(t = 2\epsilon) = W(j \to i) w_j(t = \epsilon),$$

resulting in

$$\hat{w}(t = 2\epsilon) = \begin{pmatrix} 5/32 \\ 21/32 \\ 6/32 \end{pmatrix}.$$

Note that the vector $\hat{w}$ is always normalized to 1. We find the steady state of the system by solving the linear set of equations

$$\mathbf{w}(t = \infty) = \mathbf{W}\mathbf{w}(t = \infty).$$

Brownian Motion and Markov Processes, a simple Example

This linear set of equations reads

$$\begin{aligned} W_{11} w_1(t=\infty) + W_{12} w_2(t=\infty) + W_{13} w_3(t=\infty) = & \quad w_1(t=\infty) \\ W_{21} w_1(t=\infty) + W_{22} w_2(t=\infty) + W_{23} w_3(t=\infty) = & \quad w_2(t=\infty) \\ W_{31} w_1(t=\infty) + W_{32} w_2(t=\infty) + W_{33} w_3(t=\infty) = & \quad w_3(t=\infty) \end{aligned} \tag{80}$$

with the constraint that

$$\sum_i w_i(t=\infty) = 1,$$

yielding as solution

$$\hat{w}(t=\infty) = \begin{pmatrix} 4/15 \\ 8/15 \\ 3/15 \end{pmatrix}.$$

Brownian Motion and Markov Processes, a simple Example

Convergence of the simple example

Iteration	w_1	w_2	w_3
0	1.00000	0.00000	0.00000
1	0.25000	0.75000	0.00000
2	0.15625	0.62625	0.18750
3	0.24609	0.52734	0.22656
4	0.27848	0.51416	0.20736
5	0.27213	0.53021	0.19766
6	0.26608	0.53548	0.19844
7	0.26575	0.53424	0.20002
8	0.26656	0.53321	0.20023
9	0.26678	0.53318	0.20005
10	0.26671	0.53332	0.19998
11	0.26666	0.53335	0.20000
12	0.26666	0.53334	0.20000
13	0.26667	0.53333	0.20000
$\hat{w}(t=\infty)$	0.26667	0.53333	0.20000

In a Markov chain Monte Carlo w is normally given, we need to find W!

Brownian Motion and Markov Processes, what is happening?

We have after t-steps

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^{\mathbf{t}}\hat{\mathbf{w}}(0),$$

with $\hat{\mathbf{w}}(0)$ the distribution at $t = 0$ and $\hat{\mathbf{W}}$ representing the transition probability matrix. We can always expand $\hat{\mathbf{w}}(0)$ in terms of the right eigenvectors $\hat{\mathbf{v}}$ of $\hat{\mathbf{W}}$ as

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i \hat{\mathbf{v}}_i,$$

resulting in

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t\hat{\mathbf{w}}(0) = \hat{\mathbf{W}}^t \sum_i \alpha_i \hat{\mathbf{v}}_i = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i,$$

with λ_i the ith eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

Brownian Motion and Markov Processes, what is happening?

If we assume that λ_0 is the largest eigenvector we see that in the limit $t \to \infty$, $\hat{\mathbf{w}}(t)$ becomes proportional to the corresponding eigenvector $\hat{\mathbf{v}}_0$. This is our steady state or final distribution.
In our discussion below in connection with the entropy of a system and later statistical physics and quantum physics applications, we will relate these properties to correlation functions such as the time-correlation function.

That will allow us to define the so-called *equilibration time*,viz the time needed for the system to reach its most likely state. From that state and on we can can compute contributions to various statistical variables.

Brownian Motion and Markov Processes, what is happening?

We anticipate parts of the discussion on statistical physics.
We can relate this property to an observable like the mean magnetization of say a magnetic material. With the probabilty $\hat{\mathbf{w}}(t)$ we can write the mean magnetization as

$$\langle \mathcal{M}(t) \rangle = \sum_{\mu} \hat{\mathbf{w}}(t)_{\mu} \mathcal{M}_{\mu},$$

or as the scalar of a vector product

$$\langle \mathcal{M}(t) \rangle = \hat{\mathbf{w}}(t) \mathbf{m},$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathcal{M}_{\mu}$ in its various microstates μ.
Recall our definition of an expectation value with a discrete PDF $p(x_i)$:

$$E[x^k] = \langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i),$$

provided that the sums (or integrals) $\sum_{i=1}^{N} p(x_i)$ converge absolutely (viz , $\sum_{i=1}^{N} |p(x_i)|$ converges)

Brownian Motion and Markov Processes, what is happening?

We rewrite the last relation as

$$\langle \mathcal{M}(t) \rangle = \hat{\mathbf{w}}(t)\mathbf{m} = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i \mathbf{m}_i.$$

If we define $m_i = \hat{\mathbf{v}}_i \mathbf{m}_i$ as the expectation value of $\mathcal{M}$ in the ith eigenstate we can rewrite the last equation as

$$\langle \mathcal{M}(t) \rangle = \sum_i \lambda_i^t \alpha_i m_i.$$

Since we have that in the limit $t \to \infty$ the mean magnetization is dominated by the largest eigenvalue λ_0, we can rewrite the last equation as

$$\langle \mathcal{M}(t) \rangle = \langle \mathcal{M}(\infty) \rangle + \sum_{i \neq 0} \lambda_i^t \alpha_i m_i.$$

Brownian Motion and Markov Processes, what is happening?

We define the quantity

$$\tau_i = -\frac{1}{log\lambda_i},$$

and rewrite the last expectation value as

$$\langle\mathcal{M}(t)\rangle = \langle\mathcal{M}(\infty)\rangle + \sum_{i\neq 0}\alpha_i m_i e^{-t/\tau_i}.$$

The quantities τ_i are the correlation times for the system. They control also the time-correlation functions.

The longest correlation time is obviously given by the second largest eigenvalue τ_1, which normally defines the correlation time discussed above. For large times, this is the only correlation time that survives. If higher eigenvalues of the transition matrix are well separated from λ_1 and we simulate long enough, τ_1 may well define the correlation time. In other cases we may not be able to extract a reliable result for τ_1.

Week 43

Monte Carlo methods, chapters 12 and 14

- Monday: Repetition from last week
- More on Markov chains (chapter 12.4 and 12.5)
- The Metropolis algorithm (chapter 12.5)
- Wednesday:
- Introduction to quantum Monte Carlo.
- How to perform variational Monte Carlo calculations
- How to determine the trial wave function
- Implementation of the variational Monte Carlo method to studies of the helium atom

Project 4 will be available tomorrow morning.

Entropy and Equilibrium, section 12.4

The definition of the entropy S (as a dimensionless quantity here) is

$$S = -\sum_i w_i ln(w_i), \tag{81}$$

where w_i is the probability of finding our system in a state i. For our one-dimensional randow walk it represents the probability for being at position $i = i\Delta x$ after a given number of time steps.

Assume now that we have N random walkers at $i = 0$ and $t = 0$ and let these random walkers diffuse as function of time.

Entropy and Equilibrium, section 12.4

We compute then the probability distribution for N walkers after a given number of steps i along x and time steps j. We can then compute an entropy S_j for a given number of time steps by summing over all probabilities i. The code used to compute these results is in programs/chapter12/program4.cpp. Here we have used 100 walkers on a lattice of length from $L=-50$ to $L=50$ employing periodic boundary conditions meaning that if a walker reaches the point $x=L$ it is shifted to $x=-L$ and if $x=-L$ it is shifted to $x=L$.

Entropy

```
// loop over all time steps
  for (int step=1; step <= time_steps; step++){
    // move all walkers with periodic boundary
        conditions
    for (int walks = 1; walks <= walkers; walks++){
      if (ran0(&idum) <= move_probability) {
        if ( x[walks] +1 > length) {
          x[walks] = -length;
        }
        else{
          x[walks] += 1;
        }
      }
```

Entropy

```
      else {
        if ( x[walks] -1 < -length) {
          x[walks] = length;
        }
        else{
          x[walks] -= 1;
        }
      }
    }  // end of loop over walks
  } // end of loop over trials
```

Entropy

```
// at the final time step we compute the
    probability
// by counting the number of walkers at every
    position
for ( int i = -length; i <= length; i++){
  int count = 0;
  for( int j = 1; j <= walkers; j++){
    if ( x[j] == i ) {
      count += 1;
    }
  }
  probability[i+length] = count;
}
```

Entropy

```
// Writes the results to screen
void output(int length, int time_steps, int walkers
    , int *probability)
{
  double entropy, histogram;
  // find norm of probability
  double norm = 1.0/walkers;
  // compute the entropy
  entropy = 0.; histogram = 0.;
  for( int  i = -length; i <=  length; i++){
    histogram = (double) probability[i+length]*norm
        ;
    if ( histogram > 0.0) {
    entropy -= histogram*log(histogram);
    }
  }
// then write entropy to file
```

Entropy

At small time steps the entropy is very small, reflecting the fact that we have an ordered state. As time elapses, the random walkers spread out in space (here in one dimension) and the entropy increases as there are more states, that is positions accesible to the system. We say that the system shows an increased degree of disorder. After several time steps, we see that the entropy reaches a constant value, a situation called a steady state. This signals that the system has reached its equilibrium situation and that the random walkers spread out to occupy all possible available states. At equilibrium it means thus that all states are equally probable and this is not baked into any dynamical equations such as Newton's law of motion.

Entropy

It occurs because the system is allowed to explore all possibilities. An important hypothesis is the ergodic hypothesis which states that in equilibrium all available states of a closed system have equal probability. This hypothesis states also that if we are able to simulate long enough, then one should be able to trace through all possible paths in the space of available states to reach the equilibrium situation. Our Markov process should be able to reach any state of the system from any other state if we run for long enough.

Detailed Balance

In a Markov Monte Carlo w is normally given, we need to find W! But we need to find which distribution we obtain when the steady state has been achieved.

- Markov process with transition probability from a state j to another state i

$$\sum_j W(j \rightarrow i) = 1$$

  Note that the probability for remaining at the same place is not necessarily equal zero.

- PDF w_i at time $t = n\epsilon$

$$w_i(t) = \sum_j W(j \rightarrow i)^n w_j(t = 0)$$

- $$\sum_i w_i(t) = 1$$

Detailed Balance

- Detailed balance condition

$$\sum_i W(j \to i) w_j = \sum_i W(i \to j) w_i$$

  Ensures that it is the correct distribution which is achieved when equilibrium is reached.

- When a Markow process reaches equilibrium we have

$$\mathbf{w}(t=\infty) = \mathbf{W}\mathbf{w}(t=\infty)$$

- General condition at equilibrium

$$W(j \to i) w_j = W(i \to j) w_i$$

  which is the detailed balance condition. Proof is simple.

Detailed Balance

To derive the conditions for equilibrium, we start from the so-called Master equation, which relates the temporal development of a PDF $w_i(t)$. The equation is given

$$\frac{dw_i(t)}{dt} = \sum_j \left[W(j \rightarrow i) w_j - W(i \rightarrow j) w_i \right],$$

which simply states that the rate at which the systems moves from a state j to a final state i (the first term on the right-hand side of the last equation) is balanced by the rate at which the systems undergoes transitions from the state i to a state j (the second term). If we have reached the so-called steady state, then the temporal dependence is zero. This means that in equilibrium we have

$$\frac{dw_i(t)}{dt} = 0. \tag{82}$$

Ergodicity

It should be possible for any Markov process to reach every possible state of the system from any starting point if the simulations is carried out for a long enough time.
Any state in a Boltzmann distribution has a probability different from zero and if such a state cannot be reached from a given starting point, then the system is not ergodic.

Example: Boltzmann Distribution

- At equilibrium detailed balance gives

$$\frac{W(j \to i)}{W(i \to j)} = \frac{w_i}{w_j}$$

- Boltzmann distribution

$$\frac{w_i}{w_j} = \exp\left(-\beta(E_i - E_j)\right)$$

Selection Rule

- In general

$$W(i \to j) = g(i \to j)A(i \to j)$$

  where g is a selection probability while A is the probability for accepting a move. It is also called the acceptance ratio.
- With detailed balance this gives

$$\frac{g(j \to i)A(j \to i)}{g(i \to j)A(i \to j)} = \exp\left(-\beta(E_i - E_j)\right)$$

Metropolis Algorithm

For a system which follows the Boltzmann distribution the Metropolis algorithm reads

$$A(j \to i) = \begin{cases} \exp\left(-\beta(E_i - E_j)\right) & E_i - E_j > 0 \\ 1 & \textit{else} \end{cases}$$

This algorithm satisfies the condition for detailed balance and ergodicity.

Implementation

- Establish an initial energy E_b
- Do a random change of this initial state by e.g., flipping an individual spin. This new state has energy E_t. Compute then $\Delta E = E_t - E_b$
- If $\Delta E \leq 0$ accept the new configuration.
- If $\Delta E > 0$, compute $w = e^{-(\beta \Delta E)}$.
- Compare w with a random number r. If $r \leq w$ accept, else keep the old configuration.
- Compute the terms in the sums $\sum A_s P_s$.
- Repeat the above steps in order to have a large enough number of microstates
- For a given number of MC cycles, compute then expectation values.

Test of the Metropolis Algorithm

Want to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{83}$$

with $\beta = 1/kT$ being the inverse temperature, E is the energy of the system and Z is the partition function. The only functions you will need are those to generate random numbers.

We are going to study one single particle in equilibrium with its surroundings, the latter modeled via a large heat bath with temperature T.

The model used to describe this particle is that of an ideal gas in **one** dimension and with velocity $-v$ or v. We are interested in finding $P(v)dv$, which expresses the probability for finding the system with a given velocity $v \in [v, v + dv]$. The energy for this one-dimensional system is

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{84}$$

with mass $m = 1$.

Test of the Metropolis Algorithm

Want to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{85}$$

with $\beta = 1/kT$ being the inverse temperature, E is the energy of the system and Z is the partition function. The only functions you will need are those to generate random numbers.

We are going to study one single particle in equilibrium with its surroundings, the latter modeled via a large heat bath with temperature T.

The model used to describe this particle is that of an ideal gas in **one** dimension and with velocity $-v$ or v. We are interested in finding $P(v)dv$, which expresses the probability for finding the system with a given velocity $v \in [v, v + dv]$. The energy for this one-dimensional system is

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{86}$$

with mass $m = 1$.

Test of the Metropolis Algorithm, closed form results

The partition function of the system of interest is:

$$Z = \int_{-\infty}^{+\infty} e^{-\beta v^2/2} dv = \sqrt{2\pi}\beta^{-1/2}$$

The mean velocity

$$\langle v \rangle = \int_{-\infty}^{+\infty} v e^{-\beta v^2/2} dv = 0$$

The expressions for $\langle E \rangle$ and σ_E assume the following form:

$$\langle E \rangle = \int_{-\infty}^{+\infty} \frac{v^2}{2} e^{-\beta v^2/2} dv = -\frac{1}{Z}\frac{\partial Z}{\partial \beta} = \frac{1}{2}\beta^{-1} = \frac{1}{2}T$$

$$\langle E^2 \rangle = \int_{-\infty}^{+\infty} \frac{v^4}{4} e^{-\beta v^2/2} dv = \frac{1}{Z}\frac{\partial^2 Z}{\partial \beta^2} = \frac{3}{4}\beta^{-2} = \frac{3}{4}T^2$$

and

$$\sigma_E = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{2}T^2$$

Test of the Metropolis Algorithm

```
for( montecarlo_cycles=1; Max_cycles; montecarlo_cycles++) {
   ...
   // change speed as function of delta v
   v_change = (2*ran1(&idum) -1 )* delta_v;
   v_new = v_old+v_change;
   // energy change
   delta_E = 0.5*(v_new*v_new - v_old*v_old) ;
   ......
   // Metropolis algorithm begins here
     if ( ran1(&idum) <= exp(-beta*delta_E)  ) {
         accept_step = accept_step + 1 ;
         v_old = v_new ;
     }
   // thereafter we must fill in  P[N] as a function of
   // the new speed
   // upgrade mean velocity, energy and variance
   }
```

Test of the Metropolis Algorithm

Analytical vs numerical results. $T = 4$, 10^8 MC tries, $\Delta v = 0.2$

Observable	Analytical value	Numerical value
$\langle v \rangle$	0.00000	-0.00679
$\langle E \rangle$	2.00000	1.99855
σ_E	8.00000	8.06669

Code for Metropolis test

```
v_current = v0;

// start simulation
ofile.open("evsmc.dat");
for (tries = 1; tries <= MC; tries++){

  v_change = (2.*ran0(&idum) - 1.) * dv;
  v_trial  = v_current + v_change;

  // evaluate dE
  delta_E = 0.5 * ( v_trial * v_trial - v_current * v_current );
```

Code for Metropolis test

```
// Metropolis test
if (delta_E <= 0) {
  acceptance++; v_current = v_trial;
}
else if (ran0(&idum) <= exp( -beta * delta_E )){
  acceptance++; v_current = v_trial;
}

// check if velocity value lies within given limits
if (abs(v_current) > v_max) {
  cout<<"Velocity out of range."; exit(1);
}
```

Keep or scratch?

Code for Metropolis test

```
// save event in P array
address = (int) floor( v_current / dv ) + N/2 + 1;
P[address]++;

// update mean velocity, mean energy and energy variance values
mean_v += v_current;
mean_E += 0.5 * v_current * v_current;
E_variance += 0.25 * v_current * v_current * v_current * v_current
```

Code for Metropolis test

```
    // initialize model parameters
    beta = 1./T; v_max = 10. * sqrt (T);
    // calculate amount of P-array elements
    N = 2 * (int)(v_max/dv) + 1;
    // initialize P-array
    P = new int [N];

    for (int i=0; i < N; i++) P[i] = 0;

    mean_v = 0.; mean_E = 0.; E_variance = 0.;
    acceptance = 0;

  }// initialize
```

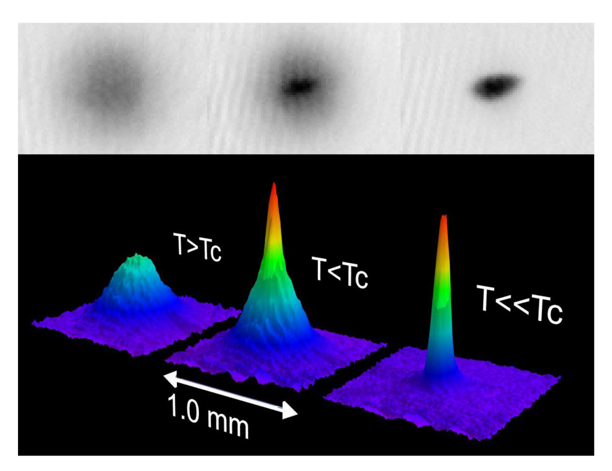
Pros and Cons of Monte Carlo

- Is physically intuitive.
- Allows one to study systems with many degrees of freedom. Diffusion Monte Carlo (DMC) and Green's function Monte Carlo (GFMC) yield in principle the exact solution to Schrödinger's equation.
- Variational Monte Carlo (VMC) is easy to implement but needs a reliable trial wave function, can be difficult to obtain.
- DMC/GFMC for fermions (spin with half-integer values, electrons, baryons, neutrinos, quarks) has a sign problem. Nature prefers an anti-symmetric wave function. PDF in this case given distribution of random walkers ($p \geq 0$).
- The solution has a statistical error, which can be large.
- There is a limit for how large systems one can study, DMC needs a huge number of random walkers in order to achieve stable results.
- Obtain only the lowest-lying states with a given symmetry. Can get excited states.

Where and why do we use Monte Carlo Methods in Quantum Physics

- Quantum systems with many particles at finite temperature: Path Integral Monte Carlo with applications to dense matter and quantum liquids (phase transitions from normal fluid to superfluid). Strong correlations.
- Bose-Einstein condensation of dilute gases, method transition from non-linear PDE to Diffusion Monte Carlo as density increases.
- Light atoms, molecules, solids and nuclei.
- Lattice Quantum-Chromo Dynamics. Impossible to solve without MC calculations.
- Simulations of systems in solid state physics, from semiconductors to spin systems. Many electrons active and possibly strong correlations.

Bose-Einstein Condensation of atoms, thousands of Atoms in one State, Possible project 5



Quantum Monte Carlo and Schrödinger's equation

For one-body problems (one dimension)

$$-\frac{\hbar^2}{2m}\nabla^2\Psi(x,t) + V(x,t)\Psi(x,t) = \imath\hbar\frac{\partial\Psi(x,t)}{\partial t},$$

$$P(x,t) = \Psi(x,t)^*\Psi(x,t)$$

$$P(x,t)dx = \Psi(x,t)^*\Psi(x,t)dx$$

Interpretation: probability of finding the system in a region between x and $x + dx$. Always real

$$\Psi(x,t) = R(x,t) + \imath I(x,t)$$

yielding

$$\Psi(x,t)^*\Psi(x,t) = (R - \imath I)(R + \imath I) = R^2 + I^2$$

Variational Monte Carlo uses only $P(x,t)$!!

Quantum Monte Carlo and Schrödinger's equation

Petit digression

Choose $\tau = it/\hbar$.
The time-dependent (1-dim) Schrödinger equation becomes then

$$\frac{\partial \Psi(x,\tau)}{\partial \tau} = \frac{\hbar^2}{2m}\frac{\partial^2 \Psi(x,\tau)}{\partial x^2} - V(x,\tau)\Psi(x,\tau).$$

With $V = 0$ we have a diffusion equation in complex time with diffusion constant

$$D = \frac{\hbar^2}{2m}.$$

Used in diffusion Monte Carlo calculations. Topic for FYS4411, Computational Physics II

Quantum Monte Carlo and Schrödinger's equation

Conditions which Ψ has to satisfy:

1. Normalization

$$\int_{-\infty}^{\infty} P(x,t)dx = \int_{-\infty}^{\infty} \Psi(x,t)^* \Psi(x,t)dx = 1$$

   meaning that

$$\int_{-\infty}^{\infty} \Psi(x,t)^* \Psi(x,t)dx < \infty$$

2. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be finite
3. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be continuous.
4. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be single valued

Square integrable functions.

First Postulate

Any physical quantity $A(\boldsymbol{r}, \boldsymbol{p})$ which depends on position $\boldsymbol{r}$ and momentum $\boldsymbol{p}$ has a corresponding quantum mechanical operator by replacing $\boldsymbol{p}$ $-i\hbar\nabla$, yielding the quantum mechanical operator

$$\widehat{\mathbf{A}} = A(\boldsymbol{r}, -i\hbar\nabla).$$

Quantity	Classical definition	QM operator
Position	$\boldsymbol{r}$	$\widehat{\boldsymbol{r}} = \boldsymbol{r}$
Momentum	$\boldsymbol{p}$	$\widehat{\boldsymbol{p}} = -i\hbar\nabla$
Orbital momentum	$\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{p}$	$\widehat{\boldsymbol{L}} = \boldsymbol{r} \times (-i\hbar\nabla)$
Kinetic energy	$T = (\boldsymbol{p})^2/2m$	$\widehat{\mathbf{T}} = -(\hbar^2/2m)(\nabla)^2$
Total energy	$H = (p^2/2m) + V(\boldsymbol{r})$	$\widehat{\mathbf{H}} = -(\hbar^2/2m)(\nabla)^2 + V(\boldsymbol{r})$

Second Postulate

The only possible outcome of an ideal measurement of the physical quantity A are the eigenvalues of the corresponding quantum mechanical operator $\widehat{\mathbf{A}}$.

$$\widehat{\mathbf{A}}\psi_\nu = a_\nu \psi_\nu,$$

resulting in the eigenvalues $a_1, a_2, a_3, \cdots$ as the only outcomes of a measurement.

The corresponding eigenstates $\psi_1, \psi_2, \psi_3 \cdots$ contain all relevant information about the system.

Third Postulate

Assume Φ is a linear combination of the eigenfunctions ψ_ν for $\widehat{\mathbf{A}}$,

$$\Phi = c_1\psi_1 + c_2\psi_2 + \cdots = \sum_\nu c_\nu \psi_\nu .$$

The eigenfunctions are orthogonal and we get

$$c_\nu = \int (\Phi)^* \psi_\nu d\tau.$$

From this we can formulate the third postulate:

When the eigenfunction is Φ, the probability of obtaining the value a_ν as the outcome of a measurement of the physical quantity A is given by $|c_\nu|^2$ and ψ_ν is an eigenfunction of $\widehat{\mathbf{A}}$ with eigenvalue a_ν.

Third Postulate

As a consequence one can show that:
when a quantal system is in the state Φ, the mean value or expectation value of a physical quantity $A(\boldsymbol{r}, \boldsymbol{p})$ is given by

$$\langle A \rangle = \int (\Phi)^* \widehat{\mathbf{A}}(\boldsymbol{r}, -i\hbar\bigtriangledown)\Phi d\tau.$$

We have assumed that Φ has been normalized, viz., $\int (\Phi)^* \Phi d\tau = 1$. Else

$$\langle A \rangle = \frac{\int (\Phi)^* \widehat{\mathbf{A}} \Phi d\tau}{\int (\Phi)^* \Phi d\tau}.$$

Fourth Postulate

The time development of of a quantal system is given by

$$i\hbar\frac{\partial \Psi}{\partial t} = \widehat{\mathbf{H}}\Psi,$$

with $\widehat{\mathbf{H}}$ the quantal Hamiltonian operator for the system.

Quantum Monte Carlo

Most quantum mechanical problems of interest in e.g., atomic, molecular, nuclear and solid state physics consist of a large number of interacting electrons and ions or nucleons. The total number of particles N is usually sufficiently large that an exact solution cannot be found. Typically, the expectation value for a chosen hamiltonian for a system of N particles is

$$\langle H \rangle =$$

$$\frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_N) H(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_N)},$$

an in general intractable problem.

Quantum Monte Carlo

As an example from the nuclear many-body problem, we have Schrödinger's equation as a differential equation

$$\hat{H}\Psi(\mathbf{r}_1,..,\mathbf{r}_A,\alpha_1,..,\alpha_A) = E\Psi(\mathbf{r}_1,..,\mathbf{r}_A,\alpha_1,..,\alpha_A)$$

where

$$\mathbf{r}_1,..,\mathbf{r}_A,$$

are the coordinates and

$$\alpha_1,..,\alpha_A,$$

are sets of relevant quantum numbers such as spin and isospin for a system of A nucleons ($A = N + Z$, N being the number of neutrons and Z the number of protons).

Quantum Monte Carlo

There are

$$2^A \times \begin{pmatrix} A \\ Z \end{pmatrix}$$

coupled second-order differential equations in $3A$ dimensions.

For a nucleus like ^{10}Be this number is **215040**. This is a truely challenging many-body problem.

Quantum Monte Carlo

Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

$$E[H] = \langle H \rangle = \frac{\int d\mathbf{R}\Psi^*_T(\mathbf{R})H(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})},$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \langle H \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system.

Quantum Monte Carlo

The trial wave function can be expanded in the eigenstates of the hamiltonian since they form a complete set, viz.,

$$\Psi_T(\mathbf{R}) = \sum_i a_i \Psi_i(\mathbf{R}),$$

and assuming the set of eigenfunctions to be normalized one obtains

$$\frac{\sum_{nm} a_m^* a_n \int d\mathbf{R} \Psi_m^*(\mathbf{R}) H(\mathbf{R}) \Psi_n(\mathbf{R})}{\sum_{nm} a_m^* a_n \int d\mathbf{R} \Psi_m^*(\mathbf{R}) \Psi_n(\mathbf{R})} = \frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0,$$

where we used that $H(\mathbf{R})\Psi_n(\mathbf{R}) = E_n \Psi_n(\mathbf{R})$. In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. The variational principle yields the lowest state of a given symmetry.

Quantum Monte Carlo

In most cases, a wave function has only small values in large parts of configuration space, and a straightforward procedure which uses homogenously distributed random points in configuration space will most likely lead to poor results. This may suggest that some kind of importance sampling combined with e.g., the Metropolis algorithm may be a more efficient way of obtaining the ground state energy. The hope is then that those regions of configurations space where the wave function assumes appreciable values are sampled more efficiently.

Quantum Monte Carlo

The tedious part in a VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). DMC is a way of solving exactly the many-body Schrödinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

Quantum Monte Carlo

- Construct first a trial wave function $\psi_T^{\alpha}(\mathbf{R})$, for a many-body system consisting of N particles located at positions $\mathbf{R} = (\mathbf{R_1}, \dots, \mathbf{R_N})$. The trial wave function depends on α variational parameters $\alpha = (\alpha_1, \dots, \alpha_N)$.
- Then we evaluate the expectation value of the hamiltonian H

$$E[H] = \langle H \rangle = \frac{\int d\mathbf{R}\Psi^{*}_{T_\alpha}(\mathbf{R})H(\mathbf{R})\Psi_{T_\alpha}(\mathbf{R})}{\int d\mathbf{R}\Psi^{*}_{T_\alpha}(\mathbf{R})\Psi_{T_\alpha}(\mathbf{R})}.$$

- Thereafter we vary α according to some minimization algorithm and return to the first step.

Quantum Monte Carlo

Choose a trial wave function $\psi_T(\mathbf{R})$.

$$P(\mathbf{R}) = \frac{|\psi_T(\mathbf{R})|^2}{\int |\psi_T(\mathbf{R})|^2 \, d\mathbf{R}}.$$

This is our new probability distribution function (PDF). The approximation to the expectation value of the Hamiltonian is now

$$E[H] \approx \frac{\int d\mathbf{R}\Psi^*_T(\mathbf{R})H(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})}.$$

Define a new quantity

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})}H\psi_T(\mathbf{R}),$$

called the local energy, which, together with our trial PDF yields

$$E[H] = \langle H \rangle \approx \int P(\mathbf{R})E_L(\mathbf{R})d\mathbf{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L(\mathbf{R_i}$$

with N being the number of Monte Carlo samples.

Quantum Monte Carlo

Algo:

- Initialisation: Fix the number of Monte Carlo steps. Choose an initial $\mathbf{R}$ and variational parameters α and calculate $\left|\psi_T^{\alpha}(\mathbf{R})\right|^2$.
- Initialise the energy and the variance and start the Monte Carlo calculation (thermalize)
  1. Calculate a trial position $\mathbf{R}_p = \mathbf{R} + r * step$ where r is a random variable $r \in [0, 1]$.
  2. Metropolis algorithm to accept or reject this move

     $$w = P(\mathbf{R}_p)/P(\mathbf{R}).$$

  3. If the step is accepted, then we set $\mathbf{R} = \mathbf{R}_p$. Update averages
- Finish and compute final averages.

Observe that the jumping in space is governed by the variable *step*. Called brute-force sampling. Need importance sampling to get more relevant sampling.

Quantum Monte Carlo

The radial Schrödinger equation for the hydrogen atom can be written as

$$-\frac{\hbar^2}{2m}\frac{\partial^2 u(r)}{\partial r^2} - \left(\frac{ke^2}{r} - \frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r) = Eu(r),$$

or with dimensionless variables

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{u(\rho)}{\rho} + \frac{l(l+1)}{2\rho^2}u(\rho) - \lambda u(\rho) = 0,$$

with the hamiltonian

$$H = -\frac{1}{2}\frac{\partial^2}{\partial \rho^2} - \frac{1}{\rho} + \frac{l(l+1)}{2\rho^2}.$$

Use variational parameter α in the trial wave function

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho}.$$

Quantum Monte Carlo

Inserting this wave function into the expression for the local energy E_L gives

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right).$$

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03

Quantum Monte Carlo

We note that at $\alpha = 1$ we obtain the exact result, and the variance is zero, as it should. The reason is that we then have the exact wave function, and the action of the hamiltonan on the wave function

$$H\psi = \mathrm{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle H^n \rangle = \frac{\int d\mathbf{R}\Psi^*_T(\mathbf{R})H^n(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})} = \mathrm{constant} \times \frac{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})} = \mathrm{constant}.$$

This gives an important information: the exact wave function leads to zero variance! Variation is then performed by minimizing both the energy and the variance.

Quantum Monte Carlo

The helium atom consists of two electrons and a nucleus with charge $Z = 2$. The contribution to the potential energy due to the attraction from the nucleus is

$$-\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2},$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1, r_2) = -\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}},$$

with the electrons separated at a distance $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$.

Quantum Monte Carlo

The hamiltonian becomes then

$$\widehat{\mathbf{H}} = -\frac{\hbar^2\nabla_1^2}{2m} - \frac{\hbar^2\nabla_2^2}{2m} - \frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}},$$

and Schrödingers equation reads

$$\widehat{\mathbf{H}}\psi = E\psi.$$

All observables are evaluated with respect to the probability distribution

$$P(\mathbf{R}) = \frac{\left|\psi_T(\mathbf{R})\right|^2}{\int \left|\psi_T(\mathbf{R})\right|^2 d\mathbf{R}}.$$

generated by the trial wave function. The trial wave function must approximate an exact eigenstate in order that accurate results are to be obtained. Improved trial wave functions also improve the importance sampling, reducing the cost of obtaining a certain statistical accuracy.

Quantum Monte Carlo

Choice of trial wave function for Helium: Assume $r_1 \to 0$.

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H\psi_T(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} \left(-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1}\right) \psi_T(\mathbf{R}) + \text{finite terms.}$$

$$E_L(R) = \frac{1}{\mathcal{R}_T(r_1)} \left(-\frac{1}{2}\frac{d^2}{dr_1^2} - \frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right) \mathcal{R}_T(r_1) + \text{finite terms}$$

For small values of r_1, the terms which dominate are

$$\lim_{r_1 \to 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_1)} \left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right) \mathcal{R}_T(r_1),$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin.

Quantum Monte Carlo

This results in

$$\frac{1}{\mathcal{R}_T(r_1)}\frac{d\mathcal{R}_T(r_1)}{dr_1} = -Z,$$

and

$$\mathcal{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have

$$\frac{1}{\mathcal{R}_T(r)}\frac{d\mathcal{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Similalry, studying the case $r_{12} \rightarrow 0$ we can write a possible trial wave function as

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}e^{\beta r_{12}}.$$

The last equation can be generalized to

$$\psi_T(\mathbf{R}) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)\dots\phi(\mathbf{r}_N)\prod_{i<j}f(r_{ij}),$$

for a system with N electrons or particles.

VMC code for helium, vmc_para.cpp

```
//  Here we define global variables  used in
    various functions
//  These can be changed by reading from file the
    different parameters
int dimension = 3; // three-dimensional system
int charge = 2;  //  we fix the charge to be that
    of the helium atom
int my_rank, numprocs;  //  these are the
    parameters used by MPI  to
                        //    define which node and
                              how many
double step_length = 1.0;  //  we fix the brute
    force jump to 1 Bohr radius
int number_particles  = 2;  //  we fix also the
    number of electrons to be 2
```

VMC code for helium, vmc_para.cpp, main part

```
//  MPI initializations
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
time_start = MPI_Wtime();

if (my_rank == 0 && argc <= 2) {
  cout << "Bad Usage: " << argv[0] <<
    " read also output file on same line" << endl
      ;
  exit(1);
}
if (my_rank == 0 && argc > 2) {
  outfilename=argv[1];
  ofile.open(outfilename);
}
```

VMC code for helium, vmc_para.cpp, main part

```
// Setting output file name for this rank:
ostringstream ost;
ost << "blocks_rank" << my_rank << ".dat";
// Open file for writing:
blockofile.open(ost.str().c_str(), ios::out | ios::
    binary);

total_cumulative_e = new double[max_variations+1];
total_cumulative_e2 = new double[max_variations+1];
cumulative_e = new double[max_variations+1];
cumulative_e2 = new double[max_variations+1];

//  initialize the arrays  by zeroing them
for( i=1; i <= max_variations; i++){
   cumulative_e[i] = cumulative_e2[i]  = 0.0;
   total_cumulative_e[i] = total_cumulative_e2[i]
      = 0.0;
}
```

VMC code for helium, vmc_para.cpp, main part

```
// broadcast the total number of  variations
MPI_Bcast (&max_variations, 1, MPI_INT, 0,
    MPI_COMM_WORLD);
MPI_Bcast (&number_cycles, 1, MPI_INT, 0,
    MPI_COMM_WORLD);
total_number_cycles = number_cycles*numprocs;
// array to store all energies for last variation
    of alpha
all_energies = new double[number_cycles+1];
//  Do the mc sampling  and accumulate data with
    MPI_Reduce
mc_sampling(max_variations, number_cycles,
    cumulative_e,
                cumulative_e2, all_energies);
//  Collect data in total averages
for( i=1; i <= max_variations; i++){
  MPI_Reduce(&cumulative_e[i], &total_cumulative_e[
      i], 1, MPI_DOUBLE,
                                          MPI_SUM, 0,
                                          MPI_COMM_WORL
```

VMC code for helium, vmc_para.cpp, main part

```
blockofile.write((char*)(all_energies+1),
                  number_cycles*sizeof(double));
blockofile.close();
delete [] total_cumulative_e; delete []
   total_cumulative_e2;
delete [] cumulative_e; delete [] cumulative_e2;
   delete [] all_energies;
// End MPI
MPI_Finalize ();
return 0;
}  //  end of main function
```

VMC code for helium, vmc_para.cpp, sampling

```
alpha = 0.5*charge;
// every node has its own seed for the random
   numbers
idum = -1-my_rank;
// allocate matrices which contain the position of
    the particles
r_old =(double **)matrix(number_particles,
   dimension,sizeof(double));
r_new =(double **)matrix(number_particles,
   dimension,sizeof(double));
for (i = 0; i < number_particles; i++) {
   for ( j=0; j < dimension; j++) {
     r_old[i][j] = r_new[i][j] = 0;
   }
 }
// loop over variational parameters
```

VMC code for helium, vmc_para.cpp, sampling

```
for (variate=1; variate <= max_variations;
    variate++){
  // initialisations of variational parameters
     and energies
  alpha += 0.1;
  energy = energy2 = 0; accept =0; delta_e=0;
  //  initial trial position, note calling with
     alpha
  for (i = 0; i < number_particles; i++) {
    for ( j=0; j < dimension; j++) {
        r_old[i][j] = step_length*(ran2(&idum)
            -0.5);
    }
  }
  wfold = wave_function(r_old, alpha);
```

VMC code for helium, vmc_para.cpp, sampling

```
// loop over monte carlo cycles
for (cycles = 1; cycles <= number_cycles; cycles++)
    {
    // new position
    for (i = 0; i < number_particles; i++) {
        for ( j=0; j < dimension; j++) {
            r_new[i][j] = r_old[i][j]+step_length*(
                ran2(&idum) -0.5);
      }
//  for the other particles we need to set the
    position to the old position since
//  we move only one particle at the time
      for (k = 0; k < number_particles; k++) {
          if ( k != i) {
              for ( j=0; j < dimension; j++) {
                r_new[k][j] = r_old[k][j];
              }
           }
        }
```

VMC code for helium, vmc_para.cpp, sampling

```
        wfnew = wave_function(r_new, alpha);
// The Metropolis test is performed by moving one
    particle at the time
        if(ran2(&idum) <= wfnew*wfnew/wfold/wfold ) {
            for ( j=0; j < dimension; j++) {
              r_old[i][j]=r_new[i][j];
            }
            wfold = wfnew;
          }
        }  // end of loop over particles
```

VMC code for helium, vmc_para.cpp, sampling

```
        // compute local energy
        delta_e = local_energy(r_old, alpha, wfold);
        // save all energies on last variate
        if(variate==max_variations){
            all_energies[cycles] = delta_e;
        }
        // update energies
        energy += delta_e;
        energy2 += delta_e*delta_e;
      }   // end of loop over MC trials
      // update the energy average and its squared
      cumulative_e[variate] = energy;
      cumulative_e2[variate] = energy2;
    }   // end of loop over variational  steps
```

VMC code for helium, vmc_para.cpp, wave function

```
// Function to compute the squared wave function,
   simplest form

double  wave_function(double **r, double alpha)
{
  int i, j, k;
  double wf, argument, r_single_particle, r_12;

  argument = wf = 0;
  for (i = 0; i < number_particles; i++) {
    r_single_particle = 0;
    for (j = 0; j < dimension; j++) {
      r_single_particle  += r[i][j]*r[i][j];
    }
    argument += sqrt(r_single_particle);
  }
  wf = exp(-argument*alpha) ;
  return wf;
}
```

VMC code for helium, vmc_para.cpp, local energy

```
// Function to calculate the local energy with num
    derivative

double  local_energy(double **r, double alpha,
    double wfold)
{
  int i, j , k;
  double e_local, wfminus, wfplus, e_kinetic,
      e_potential, r_12,
    r_single_particle;
  double **r_plus, **r_minus;
```

VMC code for helium, vmc_para.cpp, local energy

```
// allocate matrices which contain the position
    of the particles
// the function matrix is defined in the progam
    library
r_plus =(double **)matrix(number_particles,
    dimension,sizeof(double));
r_minus =(double **)matrix(number_particles,
    dimension,sizeof(double));
for (i = 0; i < number_particles; i++) {
  for ( j=0; j < dimension; j++) {
    r_plus[i][j] = r_minus[i][j] = r[i][j];
  }
}
```

VMC code for helium, vmc_para.cpp, local energy

```
    // compute the kinetic energy
    e_kinetic = 0;
    for (i = 0; i < number_particles; i++) {
      for (j = 0; j < dimension; j++) {
        r_plus[i][j] = r[i][j]+h;
        r_minus[i][j] = r[i][j]-h;
        wfminus = wave_function(r_minus, alpha);
        wfplus  = wave_function(r_plus, alpha);
        e_kinetic -= (wfminus+wfplus-2*wfold);
        r_plus[i][j] = r[i][j];
        r_minus[i][j] = r[i][j];
      }
    }
  // include electron mass and hbar squared and
     divide by wave function
    e_kinetic = 0.5*h2*e_kinetic/wfold;
```

VMC code for helium, vmc_para.cpp, local energy

```
// compute the potential energy
e_potential = 0;
// contribution from electron-proton potential
for (i = 0; i < number_particles; i++) {
  r_single_particle = 0;
  for (j = 0; j < dimension; j++) {
    r_single_particle += r[i][j]*r[i][j];
  }
  e_potential -= charge/sqrt(r_single_particle);
}
```

VMC code for helium, vmc_para.cpp, local energy

```
// contribution from electron-electron potential
for (i = 0; i < number_particles-1; i++) {
  for (j = i+1; j < number_particles; j++) {
    r_12 = 0;
    for (k = 0; k < dimension; k++) {
      r_12 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k])
          ;
    }
    e_potential += 1/sqrt(r_12);
  }
}
```

Structuring the code

During the development of our code we need to make several checks. It is also very instructive to compute a closed form expression for the local energy. Since our wave function is rather simple it is straightforward to find an analytic expressions. Consider first the case of the simple helium function

$$\Psi_T(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1+r_2)}$$

The local energy is for this case

$$E_{L1} = (\alpha - Z)\left(\frac{1}{r_1} + \frac{1}{r_2}\right) + \frac{1}{r_{12}} - \alpha^2$$

which gives an expectation value for the local energy given by

$$\langle E_{L1} \rangle = \alpha^2 - 2\alpha\left(Z - \frac{5}{16}\right)$$

Structuring the code

With analytic formulae we can speed up the computation of the correlation. In our case we write it as

$$\Psi_C = \exp\left\{\sum_{i<j} \frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

which means that the gradient needed for the local energy can be calculated analytically. This will speed up your code since the computation of the correlation part and the Slater determinant are the most time consuming parts in your code.
We will refer to this correlation function as Ψ_C or the *linear Padé-Jastrow*.

Structuring the code

We can test this by computing the local energy for our helium wave function

$$\psi_T(\mathbf{r}_1, \mathbf{r}_2) = \exp\left(-\alpha(r_1 + r_2)\right) \exp\left(\frac{r_{12}}{2(1 + \beta r_{12})}\right),$$

with α and β as variational parameters.
The local energy is for this case

$$E_{L2} = E_{L1} + \frac{1}{2(1 + \beta r_{12})^2}\left\{\frac{\alpha(r_1 + r_2)}{r_{12}}(1 - \frac{\mathbf{r}_1\mathbf{r}_2}{r_1 r_2}) - \frac{1}{2(1 + \beta r_{12})^2} - \frac{2}{r_{12}} + \frac{2\beta}{1 + \beta r_{12}}\right\}$$

It is very useful to test your code against these expressions. It means also that you don't need to compute derivatives numerically.

Structuring the code, simple task

- Make another copy of your code.
- Implement the closed form expression for the local energy
- Compile the new and old codes with the -pg option for profiling.
- Run both codes and profile them afterwards using
  gprof $<$ *executable* $>>$ *outprofile*
- Study the time usage in the file *outprofile*

Structuring the code, simple task

- Use also better compiler options like $c++-O3$
- Can speed up considerably your code

Week 44

Monte Carlo methods, chapter 14 and Ordinary diff equations, chapter 8

- Monday: Repetition from last week
- Quantum Monte Carlo (will also be one of the possible last projects)
- Summary of Monte Carlo methods
- Discussion of project 4.
- Introduction to differential equations
- Wednesday:
- Short discussion of project 4.
- Differential equations, general properties.
- Runge-Kutta methods

Chapter 13 on statistical physics is not part of this year's curriculum. Project 4 this and next week.

Differential equations program

- Ordinary differential equations, Runge-Kutta method,chapter 8
- Ordinary differential equations with boundary conditions: one-variable equations to be solved by shooting and Green's function methods, chapter 9
- We can solve such equations by a finite difference scheme as well, turning the equation into an eigenvalue problem. Still one variable. Done in projects 1 and 2.
- If we have more than one variable, we need to solve partial differential equations, which form the last part of this course. Chapter 10
- Fourier transforms and Fast Fourier transforms if we get time.

Till end of november.

The order of the ODE refers to the order of the derivative on the left-hand side in the equation

$$\frac{dy}{dt} = f(t, y). \tag{87}$$

This equation is of first order and f is an arbitrary function. A second-order equation goes typically like

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{88}$$

A well-known second-order equation is Newton's second law

$$m\frac{d^2x}{dt^2} = -kx, \tag{89}$$

where k is the force constant. ODE depend only on one variable

Differential Equations

partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x},t), \qquad (90)$$

may depend on several variables. In certain cases, like the above equation, the wave function can be factorized in functions of the separate variables, so that the Schrödinger equation can be rewritten in terms of sets of ordinary differential equations. These equations are discussed in chapter 10. Involve boundary conditions in addition to initial conditions.

Differential Equations

We distinguish also between linear and non-linear differential equation where e.g.,

$$\frac{dy}{dt} = g^3(t)y(t), \tag{91}$$

is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{92}$$

is a non-linear ODE.

Differential Equations

Another concept which dictates the numerical method chosen for solving an ODE, is that of initial and boundary conditions. To give an example, if we study white dwarf stars or neutron stars we will need to solve two coupled first-order differential equations, one for the total mass m and one for the pressure P as functions of ρ

$$\frac{dm}{dr} = 4\pi r^2 \rho(r)/c^2,$$

and

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

where ρ is the mass-energy density. The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, yielding $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star.
In the solution of the Schrödinger equation for a particle in a potential, we may need to apply boundary conditions as well, such as demanding continuity of the wave function and its derivative.

Differential Equations

In many cases it is possible to rewrite a second-order differential equation in terms of two first-order differential equations. Consider again the case of Newton's second law in Eq. (89). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{93}$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{94}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{95}$$

Differential Equations, Finite Difference

These methods fall under the general class of one-step methods. The algoritm is rather simple. Suppose we have an initial value for the function $y(t)$ given by

$$y_0 = y(t = t_0). \tag{96}$$

We are interested in solving a differential equation in a region in space [a,b]. We define a step h by splitting the interval in N sub intervals, so that we have

$$h = \frac{b - a}{N}. \tag{97}$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h), \tag{98}$$

and so forth.

Differential Equations

If the function is rather well-behaved in the domain [a,b], we can use a fixed step size. If not, adaptive steps may be needed. Here we concentrate on fixed-step methods only. Let us try to generalize the above procedure by writing the step y_{i+1} in terms of the previous step y_i

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{99}$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{100}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{101}$$

Differential Equations

We define

$$y'(t_i) = f(t_i, y_i) \tag{102}$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{103}$$

which when complemented with $t_{i+1} = t_i + h$ forms the algorithm for the well-known Euler method. Note that at every step we make an approximation error of the order of $O(h^2)$, however the total error is the sum over all steps $N = (b - a)/h$, yielding thus a global error which goes like $NO(h^2) \approx O(h)$.

Differential Equations

To make Euler's method more precise we can obviously decrease h (increase N). However, if we are computing the derivative f numerically by e.g., the two-steps formula

$$f'_{2c}(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$

we can enter into roundoff error problems when we subtract two almost equal numbers $f(x+h) - f(x) \approx 0$. Euler's method is not recommended for precision calculation, although it is handy to use in order to get a first view on how a solution may look like. As an example, consider Newton's equation rewritten in Eqs. (94) and (95). We define $y_0 = y^{(1)}(t=0)$ an $v_0 = y^{(2)}(t=0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{104}$$

and

$$y_1^{(2)} = v_0 - hy_0 k/m + O(h^2). \tag{105}$$

Differential Equations

The Euler method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at $y_1^{(1)}$ using the velocity at $y_0^{(2)} = v_0$. A simple variation is to determine $y_{n+1}^{(1)}$ using the velocity at $y_{n+1}^{(2)}$, that is (in a slightly more generalized form)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{106}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{107}$$

The acceleration a_n is a function of $a_n(y_n^{(1)}, y_n^{(2)}, t)$ and needs to be evaluated as well. This is the Euler-Cromer method.

Differential Equations

Let us then include the second derivative in our Taylor expansion. We have then

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \quad (108)$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f \quad (109)$$

and we can rewrite Eq. (100) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) + O(h^3), \quad (110)$$

which has a local approximation error $O(h^3)$ and a global error $O(h^2)$.

Differential Equations

These approximations can be generalized by using the derivative f to arbitrary order so that we have

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \ldots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}). \quad (111)$$

These methods, based on higher-order derivatives, are in general not used in numerical computation, since they rely on evaluating derivatives several times. Unless one has analytical expressions for these, the risk of roundoff errors is large.

Differential Equations

The most obvious improvements to Euler's and Euler-Cromer's algorithms, avoiding in addition the need for computing a second derivative, is the so-called midpoint method. We have then

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{112}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{113}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{114}$$

implying that the local truncation error in the position is now $O(h^3)$, whereas Euler's or Euler-Cromer's methods have a local error of $O(h^2)$.

Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.
One method that avoids this is the so-called half-step method. Here we define

$$y^{(2)}_{n+1/2} = y^{(2)}_{n-1/2} + ha_n + O(h^2), \tag{115}$$

and

$$y^{(1)}_{n+1} = y^{(1)}_{n} + hy^{(2)}_{n+1/2} + O(h^2). \tag{116}$$

Note that this method needs the calculation of $y^{(2)}_{1/2}$. This is done using e.g., Euler's method

$$y^{(2)}_{1/2} = y^{(2)}_{0} + ha_0 + O(h^2). \tag{117}$$

As this method is numerically stable, it is often used instead of Euler's method.

Differential Equations

Another method which one may encounter is the Euler-Richardson method with

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{118}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{119}$$

The program program2.cpp includes all of the above methods.

Week 45

Differential equations, with and without boundary conditions

- Monday: Repetition from last week
- Runge-Kutta methods and code examples, use of classes
- The chaotic pendulum and other ODE problems
- Wednesday:
- Ordinary differential equations with boundary conditions, chapter 9
- Begin partial differential equations, chapter 10, diffusion equation

Differential Equations, Runge-Kutta methods

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}.
To see this, consider first the following definitions

$$\frac{dy}{dt} = f(t, y), \tag{120}$$

and

$$y(t) = \int f(t, y)dt, \tag{121}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{122}$$

Differential Equations, Runge-Kutta methods

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t, y)$ around the center of the integration interval t_i to t_{i+1}, i.e., at $t_i + h/2$, h being the step. Using the midpoint formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{123}$$

This means in turn that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{124}$$

Differential Equations, Runge-Kutta methods

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We have then

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{125}$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i), \tag{126}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{127}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{128}$$

Differential Equations, Runge-Kutta methods

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely $t_i + h/2 = t_{(i+1/2)}$ where we evaluate the derivative f. This involves more operations, but the gain is a better stability in the solution.

Differential Equations, Runge-Kutta methods

The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, has the following algorithm

$$k_1 = hf(t_i, y_i), \tag{129}$$

$$k_2 = hf(t_i + h/2, y_i + k_1/2), \tag{130}$$

$$k_3 = hf(t_i + h/2, y_i + k_2/2) \tag{131}$$

$$k_4 = hf(t_i + h, y_i + k_3) \tag{132}$$

with the final value

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{133}$$

Thus, the algorithm consists in first calculating k_1 with t_i, y_1 and f as inputs. Thereafter, we increase the step size by $h/2$ and calculate k_2, then k_3 and finally k_4. Global error as $O(h^4)$.

Simple Example, Block tied to a Wall

Our first example is the classical case of simple harmonic oscillations, namely a block sliding on a horizontal frictionless surface. The block is tied to a wall with a spring. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx.$$

Simple Example, Block tied to a Wall

The negative sign means that the force acts to restore the object to an equilibrium position. Newton's equation of motion for this idealized system is then

$$m\frac{d^2x}{dt^2} = -kx,$$

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x,$$

with the angular frequency $\omega_0^2 = k/m$.
The above differential equation has the advantage that it can be solved analytically with solutions on the form

$$x(t) = Acos(\omega_0 t + \nu),$$

where A is the amplitude and ν the phase constant. This provides in turn an important test for the numerical solution and the development of a program for more complicated cases which cannot be solved analytically.

Simple Example, Block tied to a Wall

With the position $x(t)$ and the velocity $v(t) = dx/dt$ we can reformulate Newton's equation in the following way

$$\frac{dx(t)}{dt} = v(t),$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t).$$

We are now going to solve these equations using the Runge-Kutta method to fourth order discussed previously.

Simple Example, Block tied to a Wall

Before proceeding however, it is important to note that in addition to the exact solution, we have at least two further tests which can be used to check our solution.
Since functions like *cos* are periodic with a period 2π, then the solution $x(t)$ has also to be periodic. This means that

$$x(t+T) = x(t),$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}.$$

Observe that T depends only on k/m and not on the amplitude of the solution.

Simple Example, Block tied to a Wall

In addition to the periodicity test, the total energy has also to be conserved. Suppose we choose the initial conditions

$$x(t=0) = 1 \text{ m} \qquad v(t=0) = 0 \text{ m/s},$$

meaning that block is at rest at $t = 0$ but with a potential energy

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0 = \frac{1}{2}kx(t)^2 + \frac{1}{2}mv(t)^2.$$

Simple Example, Block tied to a Wall

An algorithm which implements these equations is included below.

1. Choose the initial position and speed, with the most common choice $v(t=0)=0$ and some fixed value for the position.
2. Choose the method you wish to employ in solving the problem.
3. Subdivide the time interval $[t_i, t_f]$ into a grid with step size

$$h = \frac{t_f - t_i}{N},$$

   where N is the number of mesh points.

4. Calculate now the total energy given by

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

5. The Runge-Kutta method is used to obtain x_{i+1} and v_{i+1} starting from the previous values x_i and v_i..
6. When we have computed $x(v)_{i+1}$ we upgrade $t_{i+1} = t_i + h$.
7. This iterative process continues till we reach the maximum time t_f.
8. The results are checked against the exact solution. Furthermore, one has to check the stability of the numerical solution against the chosen number of mesh points N.

Simple Example, Block tied to a Wall

```
 y[0] = initial_x;                    // initial position
 y[1] = initial_v;                   // initial velocity
 t=0.;                                // initial time
 E0 = 0.5*y[0]*y[0]+0.5*y[1]*y[1];  // the initial total energy
 // now we start solving the differential
 // equations using the RK4 method
 while (t <= tmax){
   derivatives(t, y, dydt);   // initial derivatives
   runge_kutta_4(y, dydt, n, t, h, yout, derivatives);
   for (i = 0; i < n; i++) {
y[i] = yout[i];
   }
   t += h;
   output(t, y, E0);   // write to file
 }
```

Simple Example, Block tied to a Wall

```
//   this function sets up the derivatives for this special case
void derivatives(double t, double *y, double *dydt)
{
  dydt[0]=y[1];    // derivative of x
  dydt[1]=-y[0]; // derivative of v
} // end of function derivatives
```

Runge-Kutta methods, code

```
void runge_kutta_4(double *y, double *dydx, int n,
                   double x, double h,
           double *yout, void (*derivs)(double, double *, double *))
{
  int i;
  double      xh,hh,h6;
  double *dym, *dyt, *yt;
  //   allocate space for local vectors
  dym = new double [n];
  dyt =  new double [n];
  yt =  new double [n];
  hh = h*0.5;
  h6 = h/6.;
  xh = x+hh;
```

Runge-Kutta methods, code

```
for (i = 0; i < n; i++) {
  yt[i] = y[i]+hh*dydx[i];
}
(*derivs)(xh,yt,dyt);     // computation of k2
for (i = 0; i < n; i++) {
  yt[i] = y[i]+hh*dyt[i];
}
(*derivs)(xh,yt,dym); //  computation of k3
for (i=0; i < n; i++) {
  yt[i] = y[i]+h*dym[i];
  dym[i] += dyt[i];
}
(*derivs)(x+h,yt,dyt);    // computation of k4
//      now we upgrade y in the array yout
for (i = 0; i < n; i++){
  yout[i] = y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
}
delete []dym;
delete [] dyt;
delete [] yt;
}       //  end of function Runge-kutta 4
```

The classical pendulum

The angular equation of motion of the pendulum is given by Newton's equation and with no external force it reads

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{134}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{135}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{136}$$

More on the Pendulum

We do however expect that the motion will gradually come to an end due a viscous drag torque acting on the pendulum. In the presence of the drag, the above equation becomes

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{137}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \tag{138}$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

More on the Pendulum

We define

$$\omega_0 = \sqrt{g/l},$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t} = \omega_0 t,$$

with the dimensionless driving frequency

$$\hat{\omega} = \frac{\omega}{\omega_0},$$

and introducing the quantity Q, called the *quality factor*,

$$Q = \frac{mg}{\omega_0 \nu},$$

and the dimensionless amplitude

$$\hat{A} = \frac{A}{mg}$$

More on the Pendulum

we have

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + sin(\theta) = \hat{A}cos(\hat{\omega}\hat{t}).$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v},$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - sin(\theta) + \hat{A}cos(\hat{\omega}\hat{t}).$$

These are the equations to be solved. The factor Q represents the number of oscillations of the undriven system that must occur before its energy is significantly reduced due to the viscous drag. The amplitude $\hat{A}$ is measured in units of the maximum possible gravitational torque while $\hat{\omega}$ is the angular frequency of the external torque measured in units of the pendulum's natural frequency.

Classes for ODE methods

It can be very useful to make a Class which contains all possible methods discussed. In Fortran we can use the MODULE keyword in order to can methods and keep the variables private and hidden from other parts of our code. This allows for a generalization which can be used to tackle other ODEs as well.

Classes for ODE methods

In program2.cpp of chapter 8 we have canned the following methods

- void euler();
- void euler_cromer();
- void midpoint();
- void euler_richardson();
- void half_step();
- void rk2(); //runge-kutta-second-order
- void rk4_step(double,double*,double*,double); // we need it in function rk4() and asc()
- void rk4(); //runge-kutta-fourth-order
- void asc(); //runge-kutta-fourth-order with adaptive stepsize control

Classes for ODE methods

```
class pendulum
 {
 private:
   double Q, A_roof, omega_0, omega_roof,g; //
   double y[2];          //for the initial-values of phi and v
   int n;                // how many steps
   double delta_t,delta_t_roof;

 public:
   void derivatives(double,double*,double*);
   void initialise();
   void euler();
   void euler_cromer();
   void midpoint();
   void euler_richardson();
   void half_step();
   void rk2(); //runge-kutta-second-order
   void rk4_step(double,double*,double*,double); // we need it in func
   void rk4(); //runge-kutta-fourth-order
   void asc(); //runge-kutta-fourth-order with adaptive stepsize contr
 };
```

Classes for ODE methods

```
void pendulum::derivatives(double t, double* in, double* out)
{ /* Here we are calculating the derivatives at (dimensionless) time t
     'in' are the values of phi and v, which are used for the calculat
     The results are given to 'out' */

  out[0]=in[1];                 //out[0] = (phi)'  = v
  if(Q)
    out[1]=-in[1]/((double)Q)-sin(in[0])+A_roof*cos(omega_roof*t);  //
  else
    out[1]=-sin(in[0])+A_roof*cos(omega_roof*t);  //out[1] = (phi)''
}
```

Classes for ODE methods

```
int main()
{
  pendulum testcase;
  testcase.initialise();
  testcase.euler();
  testcase.euler_cromer();
  testcase.midpoint();
  testcase.euler_richardson();
  testcase.half_step();
  testcase.rk2();
  testcase.rk4();
  return 0;
}  // end of main function
```

Classes for ODE methods

In Fortran we would use

```
MODULE pendulum
   USE CONSTANTS
   IMPLICIT NONE
   REAL(DP), PRIVATE :: Q, A_roof, omega_0, omega_roof,g
   REAL(DP), PRIVATE :: y(2)         ! for the initial-values of phi a
   INTEGER, PRIVATE ::  n               ! how many steps
   REAL(DP), PRIVATE :: delta_t,delta_t_roof

   CONTAINS
    SUBROUTINE derivatives(..)
    SUBROUTINE initialise(..)
    SUBROUTINE euler(..)
    SUBROUTINE euler_cromer(..)
    SUBROUTINE midpoint(..)
    etc

END MODULE pendulum
```

Boundary value problems, chapter 9

Suppose we want to solve the following boundary value equation

$$\frac{d^2u(x)}{dx^2} + g(x)u(x) = f(x, u(x)),$$

with $x \in (a, b)$ and with for example boundary conditions $u(a) = u(b) = 0$. We assume that f is a continuous function in the domain $x \in (a, b)$.
We have already seen this in project 1 ($g(x) = 0$.
Another example of an equation with similar boundary conditions is

$$\frac{d^2u(x)}{dx^2} + g(x)u(x) = \lambda u(x),$$

which is often called an eigenvalue equation.

How do we solve such equations?

Boundary Value Problems

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(r) + \left(V(\rho) + \frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho) = Eu(\rho).$$

In our case we are interested in attractive potentials

$$V(r) = -V_0 f(r),$$

where $V_0 > 0$ and analyze bound states where $E < 0$.

Boundary Value Problems

The final equation can be written as

$$\frac{d^2}{d\rho^2}u(\rho) + k(\rho)u(\rho) = 0,$$

where

$$\begin{aligned} k(\rho) &= \gamma\left(f(\rho) - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} - \epsilon\right) \\ \gamma &= \frac{2m\alpha^2 V_0}{\hbar^2} \\ \epsilon &= \frac{|E|}{V_0} \end{aligned}$$

Boundary Value Problems

$$f(r) = \begin{cases} 1 & \text{for} \quad r \leq a \\ 0 & \phantom{\text{for}} \quad r > a \end{cases}$$

and choose $\alpha = a$. Then

$$k(\rho) = \gamma \begin{cases} 1 - \epsilon - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} & \text{for} \quad r \leq a \\ -\epsilon - -\frac{1}{\gamma}\frac{l(l+1)}{\rho^2} & \phantom{\text{for}} \quad r > a \end{cases}$$

Boundary Value Problems

For small ρ we get

$$\frac{d^2}{d\rho^2}u(\rho) - \frac{l(l+1)}{\rho^2}u(\rho) = 0,$$

with solutions $u(\rho) = \rho^{l+1}$ or $u(\rho) = \rho^{-l}$. Since the final solution must be finite everywhere we get the condition for our numerical solution

$$u(\rho) = \rho^{l+1} \quad \text{for small } \rho$$

Boundary Value Problems

For large ρ we get

$$\frac{d^2}{d\rho^2}u(\rho) - \gamma\epsilon u(\rho) = 0 \quad \gamma > 0,$$

with solutions $u(\rho) = \exp(\pm\gamma\epsilon\rho)$ and the condition for large ρ means that our numerical solution must satisfy

$$u(\rho) = e^{-\gamma\epsilon\rho} \quad \text{for large } \rho$$

Boundary Value Problems

In order to find a bound state we start integrating, with a trial negative value for the energy, from small values of the variable ρ, usually zero, and up to some large value of ρ. As long as the potential is significantly different from zero the function oscillates. Outside the range of the potential the function will approach an exponential form. If we have chosen a correct eigenvalue the function decreases exponetially as $u(\rho) = e^{-\gamma\epsilon\rho}$. However, due to numerical inaccuracy the solution will contain small admixtures of the undesireable exponential growing function $u(\rho) = e^{+\gamma\epsilon\rho}$.

Boundary Value Problems

The final solution will then become unstable. Therefore, it is better to generate two solutions, with one starting from small values of ρ and integrate outwards to some matching point $\rho = \rho_m$. We call that function $u^{<}(\rho)$. The next solution $u^{>}(\rho)$ is then obtained by integrating from some large value ρ where the potential is of no importance, and inwards to the same matching point ρ_m. Due to the quantum mechanical requirements the logarithmic derivative at the matching point ρ_m should be well defined.

Boundary Value Problems

We obtain the following condition

$$\frac{\frac{d}{d\rho}u^<(\rho)}{u^<(\rho)} = \frac{\frac{d}{d\rho}u^>(\rho)}{u^>(\rho)} \quad \text{at} \quad \rho = \rho_m.$$

We can modify this expression by normalizing the function $u^< u^<(\rho_m) = Cu^> u^<(\rho_m)$.

$$\frac{d}{d\rho}u^{<}(\rho) = \frac{d}{d\rho}u^{>}(\rho) \quad \text{at} \quad \rho = \rho_m$$

We can calculate the first order derivatives by

$$\frac{d}{d\rho}u^{<}(\rho_m) \approx \frac{u^{<}(\rho_m) - u^{<}(\rho_m - h)}{h}$$
$$\frac{d}{d\rho}u^{>}(\rho_m) \approx \frac{u^{>}(\rho_m + h) - u^{>}(\rho_m)}{h}$$

Thus the criterium for a proper eigenfunction will be

$$f = u^{<}(\rho_m - h) - u^{>}(\rho_m + h)$$

which should be smaller than a fixed number.

Boundary Value Problems

The algorithm could then take the following form

- Initialise the problem by choosing minimum and maximum values for the energy, $E_{\rm min}$ and $E_{\rm max}$, the maximum number of iterations $\rm max_iter$ and the desired numerical precision.
- Search then for the roots of the function $f(E)$, where the root(s) is(are) in the interval $E \in [E_{\rm min}, E_{\rm max}]$ using e.g., the bisection method.

Boundary Value Problems

```
do {
    i++;
    e = (e_min+e_max)/2.;  //bisection
    if ( f(e)*f(e_max) > 0 ) {
       e_max = e; //change search interval
    }
    else { e_min = e; }
} while ( (fabs(f(e) > convergence_test) !! (i <= max_iterations))
```

Boundary Value Problems

- The use of a root-searching method forms the shooting part of the algorithm. We have however not yet specified the matching part.
- The matching part is given by the function $f(e)$ which receives as argument the present value of E. This function forms the core of the method and is based on an integration of Schrödinger's equation from $\rho = 0$ and $\rho = \infty$. If

  $$f = u^{<}(\rho_m - h) - u^{>}(\rho_m + h) \leq \mathrm{test}$$

  we have a solution.

The function $f(E)$ receives as input a guess for the energy. In the version implemented below, we use the standard three-point formula for the second derivative, namely

$$f_0'' \approx \frac{f_h - 2f_0 + f_{-h}}{h^2}.$$

Boundary Value Problems

```
void f(double step, int max_step, double energy, double *w, double *wf
{
   int      loop, loop_1,match;
   double   fac, wwf, norm;
// adding the energy guess to the array containing the potential
   for(loop = 0; loop <= max_step; loop ++) {
      w[loop] = (w[loop] - energy) * step * step + 2;
   }
}
```

Boundary Value Problems

```
// integrating from large r-values
   wf[max_step]      = 0.0;
   wf[max_step - 1] = 0.5 * step * step;
// search for matching point
   for(loop = max_step - 2; loop > 0; loop--) {
      wf[loop] = wf[loop + 1] * w[loop + 1] - wf[loop + 2];
      if(wf[loop] <= wf[loop + 1]) break;
   }
   match = loop + 1;
   wwf   = wf[match];
```

Boundary Value Problems

```
// start integrating up to matching point from r =0
   wf[0] = 0.0; wf[1] = 0.5 * step * step;
   for(loop = 2; loop <= match; loop++) {
      wf[loop] = wf[loop -1] * w[loop - 1] - wf[loop - 2];
      if(fabs(wf[loop]) > INFINITY) {
         for(loop_1 = 0; loop_1 <= loop; loop_1++) {
            wf[loop_1] /= INFINITY;
         }
      }
   }
   return fabs(wf[match-1]-wf[match+1]);
```

Week 46

Partial differential equations

- Monday: Repetition from last week
- Discussion of projects (nr 5, three different versions, diffusion equation, quantum mechanical Monte Carlo and solar system (ODE))
- How to use titan.uio.no (the quantum mechanical project)
- Diffusion equation, implicit, explicit and the Crank-Nicolson schemes
- Wednesday:
- Diffusion equation continued
- Discussion of projects (diffusion project and quantum mechanics)
- Poisson's and Laplace's equations
- Parallel diffusion equation and Poisson equation next week.

How do I use the titan.uio.no cluster?

hpc@usit.uio.no

- Computational Physics requires High Performance Computing (HPC) resources
- USIT and the Research Computing Services (RCS) provides HPC resources and HPC support
- Resources: `titan.uio.no`
- Support: 14 people
- Contact: `hpc@usit.uio.no`

Titan

Hardware

- 304 dual-cpu quad-core SUN X2200 Opteron nodes (total 2432 cores), 2.2 Ghz, and 8 - 16 GB RAM and 250 - 1000 GB disk on each node
- 3 eight-cpu quad-core Sun X4600 AMD Opteron nodes (total 96 cores), 2.5 Ghz, and 128, 128 and 256 GB memory, respectively
- Infiniband interconnect
- Heterogenous cluster!

Titan

Software

- Batch system: SLURM and MAUI
- Message Passing Interface (MPI):
  - OpenMPI
  - Scampi
  - MPICH2
- Compilers: GCC, Intel, Portland and Pathscale
- Optimized math libraries and scientific applications
- All you need may be found under `/site`
- Available software: `http://www.hpc.uio.no/index.php/Titan_software`

Getting started

Batch systems

- A batch system controls the use of the cluster resources
- Submits the job to the right resource
- Monitors the job while executing
- Restarts the job in case of failure
- Takes care of priorities and queues to control execution order of unrelated jobs

Sun Grid Engine

- SGE is the batch system used on Titan
- Jobs are executed either interactively or through job scripts
- Useful commands: `showq`, `qlogin`, `sbatch`
- `http://hpc.uio.no/index.php/Titan_User_Guide`

Getting started

Modules

- Different compilers, MPI-versions and applications need different sets of user environment variables
- The `modules` package lets you load and remove the different variable sets
- Useful commands:
  - List available modules: `module avail`
  - Load module: `module load <environment>`
  - Unload module: `module unload <environment>`
  - Currently loaded: `module list`
- `http://hpc.uio.no/index.php/Titan_User_Guide`

Example

Interactively

```
# login to titan
$ ssh titan.uio.no
# ask for 4 cpus
$ qlogin --account=fys3150 --ntasks=4
# start a job setup, note the punktum!
$ source  /site/bin/jobsetup
# we want to use the intel module
$ module load intel
$ module load openmpi/1.2.8.intel
$ mkdir -p fys3150/mpiexample/
$ cd fys3150/mpiexample/
# Use program6.cpp from the course pages, see chapter 4
# compile the program
$ mpic++ -O3 -o program6.x program6.cpp
# and execute it
$ mpirun ./program6.x
$ Trapezoidal rule = 3.14159
$ Time = 0.000378132 on number of processors: 4
```

The job script

job.sge

```
#!/bin/sh
# Call this file job.slurm
# 4 cpus with mpi (or other communication)
#SBATCH --ntasks=4
# 10 mins of walltime
#SBATCH --time=0:10:00
# project fys3150
#SBATCH --account=fys3150
# we need 2000 MB of memory per process
#SBATCH --mem-per-cpu=2000M
# name of job
#SBATCH --job-name=program5

source /site/bin/jobsetup

# load the module used when we compiled the program
module load intel
module load openmpi/1.2.8.intel


# start program
mpirun ./program5.x

#END OF SCRIPT
```

Example

Submitting

```
# login to titan
$ ssh titan.uio.no
# and submit it
$ sbatch job.slurm
$ exit
```

Example

Checking execution

```
# check if job is running:
$ showq -u mhjensen
ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE  PROC   REMAINING            STARTTIME

883129             mhjensen    Running     4    10:31:17  Fri Oct  2 13:59:25

     1 Active Job      2692 of 4252 Processors Active (63.31%)
                        482 of  602 Nodes Active      (80.07%)

IDLE JOBS----------------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME


0 Idle Jobs

BLOCKED JOBS----------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME


Total Jobs: 1   Active Jobs: 1   Idle Jobs: 0   Blocked Jobs: 0
```

Tips and admonitions

Tips

- Titan FAQ: `http://www.hpc.uio.no/index.php/FAQ`
- man-pages, e.g. `man sbatch`
- Ask us

Admonitions

- Remember to exit from qlogin-sessions; the resource is reserved for you untill you exit
- Don't run jobs on login-nodes; these are only for compiling and editing files

Partial Differential Equations, chapter 10

General 2+1-dim PDE

$$A(x, y)\frac{\partial^2 U}{\partial x^2} + B(x, y)\frac{\partial^2 U}{\partial x \partial y} + C(x, y)\frac{\partial^2 U}{\partial y^2} = F(x, y, U, \frac{\partial U}{\partial x}, \frac{\partial U}{\partial y})$$

Examples

$$B = C = 0,$$

give e.g., 1+1-dim diffusion equation

$$A\frac{\partial^2 U}{\partial x^2} = \frac{\partial U}{\partial t}$$

and is an example of a parabolic PDE

Partial Differential Equations

More examples 2+1-dim wave equation

$$A\frac{\partial^2 U}{\partial x^2} + C\frac{\partial^2 U}{\partial y^2} = \frac{\partial^2 U}{\partial t^2}$$

Poisson's (Laplace's $\rho = 0$) equation

$$\nabla^2 U(\mathbf{x}) = -4\pi\rho(\mathbf{x}).$$

Heat/Diffusion Equation

Diffusion equation

$$\frac{\kappa}{C\rho}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}$$

$$\frac{\kappa}{C\rho(\mathbf{x},t)}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}$$

Explicit Scheme for the Diffusion Equation

In one dimension we have thus the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t}, \tag{139}$$

or

$$u_{xx} = u_t, \tag{140}$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = g(x) \qquad 0 \leq x \leq L \tag{141}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = a(t) \qquad t \geq 0, \tag{142}$$

and

$$u(L,t) = b(t) \qquad t \geq 0, \tag{143}$$

where $a(t)$ and $b(t)$ are two functions which depend on time only, while $g(x)$ depends only on the position x.

Explicit Scheme, Forward Euler

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}, \tag{144}$$

and

$$u_{xx} \approx \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{145}$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{146}$$

Defining $\alpha = \Delta t/\Delta x^2$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{147}$$

Explicit Scheme

$$V_{j+1} = AV_j$$

with

$$A = \begin{pmatrix} 1-2\alpha & \alpha & 0 & 0\dots \\ \alpha & 1-2\alpha & \alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & \alpha & 1-2\alpha \end{pmatrix}$$

yielding

$$V_{j+1} = AV_j = \dots = A^j V_0$$

The explicit scheme, although being rather simple to implement has a very weak stability condition given by

$$\Delta t/\Delta x^2 \leq 1/2 \tag{148}$$

Implicit Scheme

Choose now

$$u_t \approx \frac{u(x_i, t_j) - u(x_i, t_j - k)}{k}$$

and

$$u_{xx} \approx \frac{u(x_i + h, t_j) - 2u(x_i, t_j) + u(x_i - h, t_j)}{h^2}$$

Define $\alpha = k/h^2$. Gives

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} - \alpha u_{i+1,j}$$

Here $u_{i,j-1}$ is the only unknown quantity.

Have

$$AV_j = V_{j-1}$$

with

$$A = \begin{pmatrix} 1+2\alpha & -\alpha & 0 & 0\ldots \\ -\alpha & 1+2\alpha & -\alpha & 0\ldots \\ \ldots & \ldots & \ldots & \ldots \\ 0\ldots & 0\ldots & -\alpha & 1+2\alpha \end{pmatrix}$$

which gives

$$V_j = A^{-1}V_{j-1} = \cdots = A^{-j}V_0$$

Need only to invert a matrix

Brute Force Implicit Scheme, inefficient algo

```
!  now invert the matrix
      CALL matinv( a, ndim, det)
      DO i = 1, m
         DO l=1, ndim
            u(l) = DOT_PRODUCT(a(l,:),v(:))
         ENDDO
         v = u
         t = i*k
         DO  j=1, ndim
             WRITE(6,*) t, j*h, v(j)
         ENDDO
      ENDDO
```

Brief Summary of the Explicit and the Implicit Methods

- Explicit is straightforward to code, but avoid doing the matrix vector multiplication since the matrix is tridiagonal.

$$u_t \approx \frac{u(x, t+\Delta t) - u(x, t)}{\Delta t} = \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}$$

- The implicit method can be applied in a brute force way as well as long as the element of the matrix are constants.

$$u_t \approx \frac{u(x, t) - u(x, t-\Delta t)}{\Delta t} = \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t}$$

- However, it is more efficient to use a linear algebra solver for tridiagonal matrices.

Crank-Nicolson

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j}-2u_{i,j}+u_{i+1,j}\right)+\frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1}-2u_{i,j-1}+u_{i-1,j-1}\right)=\frac{1}{\Delta t}\left(u_{i,j}-u_{i,j-1}\right),$$

which for $\theta = 0$ yields the forward formula for the first derivative and the explicit scheme, while $\theta = 1$ yields the backward formula and the implicit scheme. These two schemes are called the backward and forward Euler schemes, respectively. For $\theta = 1/2$ we obtain a new scheme after its inventors, Crank and Nicolson.

Crank Nicolson

Using our previous definition of $\alpha = \Delta t/\Delta x^2$ we can rewrite the latter equation as

$$-\alpha u_{i-1,j} + (2+2\alpha)\, u_{i,j} - \alpha u_{i+1,j} = \alpha u_{i-1,j-1} + (2-2\alpha)\, u_{i,j-1} + \alpha u_{i+1,j-1},$$

or in matrix-vector form as

$$\left(2\hat{I} + \alpha\hat{B}\right) V_j = \left(2\hat{I} - \alpha\hat{B}\right) V_{j-1},$$

where the vector V_j is the same as defined in the implicit case while the matrix $\hat{B}$ is

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & 0\dots \\ -1 & 2 & -1 & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & & 2 \end{pmatrix}$$

Analysis of diffusion equation

We start with the forward Euler scheme and Taylor expand $u(x, t + \Delta t)$, $u(x + \Delta x, t)$ and $u(x - \Delta x, t)$

$$
\begin{aligned}
u(x + \Delta x, t) &= u(x, t) + \frac{\partial u(x,t)}{\partial x}\Delta x + \frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2 + \mathcal{O}(\Delta x^3), \qquad (149) \\
u(x - \Delta x, t) &= u(x, t) - \frac{\partial u(x,t)}{\partial x}\Delta x + \frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2 + \mathcal{O}(\Delta x^3), \\
u(x, t + \Delta t) &= u(x, t) + \frac{\partial u(x,t)}{\partial t}\Delta t + \mathcal{O}(\Delta t^2).
\end{aligned}
$$

Analysis of diffusion equation

With these Taylor expansions the approximations for the derivatives takes the form

$$\left[\frac{\partial u(x,t)}{\partial t}\right]_{\text{approx}} = \frac{\partial u(x,t)}{\partial t} + \mathcal{O}(\Delta t), \tag{150}$$

$$\left[\frac{\partial^2 u(x,t)}{\partial x^2}\right]_{\text{approx}} = \frac{\partial^2 u(x,t)}{\partial x^2} + \mathcal{O}(\Delta x^2).$$

It is easy to convince oneself that the backward Euler method must have the same truncation errors as the forward Euler scheme.

Analysis of diffusion equation

For the Crank-Nicolson scheme we also need to Taylor expand $u(x+\Delta x, t+\Delta t)$ and $u(x-\Delta x, t+\Delta t)$ around $t' = t+\Delta t/2$.

$$
\begin{aligned}
u(x+\Delta x, t+\Delta t) &= u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} + \\
&\qquad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x-\Delta x, t+\Delta t) &= u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} - \\
&\qquad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x+\Delta x, t) &= u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} - \\
&\qquad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x-\Delta x, t) &= u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} + \\
&\qquad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x, t+\Delta t) &= u(x,t') + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3) \\
u(x, t) &= u(x,t') - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3)
\end{aligned}
$$

Analysis of diffusion equation

We now insert these expansions in the approximations for the derivatives to find

$$\left[\frac{\partial u(x,t')}{\partial t}\right]_{\text{approx}} = \frac{\partial u(x,t')}{\partial t} + \mathcal{O}(\Delta t^2), \tag{151}$$

$$\left[\frac{\partial^2 u(x,t')}{\partial x^2}\right]_{\text{approx}} = \frac{\partial^2 u(x,t')}{\partial x^2} + \mathcal{O}(\Delta x^2).$$

Analysis of diffusion equation

The following table summarizes the three methods.

Scheme:	*Truncation Error:*	*Stability requirements:*
Crank-Nicolson	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t^2)$	Stable for all Δt and Δx.
Backward Euler	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	Stable for all Δt and Δx.
Forward Euler	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	$\Delta t \leq \frac{1}{2}\Delta x^2$

Table: Comparison of the different schemes.

Analysis of diffusion equation

It cannot be repeated enough, it is always useful to find cases where one can compare the numerics and the developed algorithms and codes with analytic solution. The above case is also particularly simple. We have the following partial differential equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t},$$

with initial conditions

$$u(x,0) = g(x) \qquad 0 < x < L.$$

Analysis of diffusion equation

The boundary conditions are

$$u(0,t) = 0 \qquad t \geq 0, \qquad\qquad u(L,t) = 0 \qquad t \geq 0,$$

We assume that we have solutions of the form (separation of variable)

$$u(x,t) = F(x)G(t), \tag{152}$$

which inserted in the partial differential equation results in

$$\frac{F''}{F} = \frac{G'}{G}, \tag{153}$$

where the derivative is with respect to x on the left hand side and with respect to t on right hand side. This equation should hold for all x and t. We must require the rhs and lhs to be equal to a constant.

Analysis of diffusion equation

We call this constant $-\lambda^2$. This gives us the two differential equations,

$$F'' + \lambda^2 F = 0; \qquad G' = -\lambda^2 G, \tag{154}$$

with general solutions

$$F(x) = A\sin(\lambda x) + B\cos(\lambda x); \qquad G(t) = Ce^{-\lambda^2 t}. \tag{155}$$

Analysis of diffusion equation

To satisfy the boundary conditions we require $B = 0$ and $\lambda = n\pi/L$. One solution is therefore found to be

$$u(x,t) = A_n \sin(n\pi x/L) e^{-n^2\pi^2 t/L^2}. \tag{156}$$

But there are infinitely many possible n values (infinite number of solutions). Moreover, the diffusion equation is linear and because of this we know that a superposition of solutions will also be a solution of the equation. We may therefore write

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L) e^{-n^2\pi^2 t/L^2}. \tag{157}$$

Analysis of diffusion equation

The coefficient A_n is in turn determined from the initial condition. We require

$$u(x,0) = g(x) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L). \tag{158}$$

The coefficient A_n is the Fourier coefficients for the function $g(x)$. Because of this, A_n is given by (from the theory on Fourier series)

$$A_n = \frac{2}{L}\int_0^L g(x)\sin(n\pi x/L)\mathrm{d}x. \tag{159}$$

Different $g(x)$ functions will obviously result in different results for A_n.

Week 47

Partial differential equations

- Monday: Repetition from last week
- Discussion of projects (nr 5, three different versions, diffusion equation, quantum mechanical Monte Carlo and solar system (ODE))
- Poisson's and Laplace's equations
- Parallel diffusion equation and Poisson
- Discussion on report
- Wednesday:
- Discussion of projects
- Parallel diffusion equation and Poisson

The report

What should it contain? A possible structure

- An introduction where you explain the rational for the physics case and what you have done. At the end of the introduction you should give a brief summary of the structure of the report
- Theoretical models and technicalities. This is the methods section.
- Results and discussion
- Conclusions and perspectives
- Appendix with extra material
- Bibliography

Projects: quantum mechanics

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$K_i = -\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}.$$

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_J}{\Psi_J} + 2\frac{\mathbf{\nabla}\Psi_D}{\Psi_D}\cdot\frac{\mathbf{\nabla}\Psi_J}{\Psi_J} \tag{160}$$

Quantum mechanical project

We define the correlated function as

$$\Psi_J = \prod_{i<j} g(r_{ij}) = \prod_{i<j}^{N} g(r_{ij}) = \prod_{i=1}^{N} \prod_{j=i+1}^{N} g(r_{ij}),$$

with $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ for three dimensions and $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ for two dimensions.
In our particular case we have

$$\Psi_J = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\} = \exp\left\{\sum_{i<j} \frac{a r_{ij}}{1 + \beta r_{ij}}\right\},$$

Quantum mechanical project

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2 \Psi_J}{\Psi_J}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

But we have a simple form for the function, namely

$$\Psi_J = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2 \Psi_J}{\Psi_J} = \sum_{ij\neq k}\frac{(\mathbf{r}_k - \mathbf{r}_i)(\mathbf{r}_k - \mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Jastrow gradient

We have

$$\Psi_J = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} \frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

the gradient needed for the local energy is easy to compute. We get for particle k

$$\frac{\nabla_k \Psi_J}{\Psi_J} = \sum_{j\neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2},$$

which is rather easy to code. Remember to sum over all particles when you compute the local energy.

Laplace's and Poisson's equations

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{161}$$

with possible boundary conditions $u(x,y) = g(x,y)$ on the border. There is no time-dependence. Choosing equally many steps in both directions we have a quadratic or rectangular grid, depending on whether we choose equal steps lengths or not in the x and the y directions. Here we set $\Delta x = \Delta y = h$ and obtain a discretized version

$$u_{xx} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}, \tag{162}$$

and

$$u_{yy} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2}, \tag{163}$$

Laplace's and Poisson's equations

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{164}$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{165}$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \tag{166}$$

This is our final numerical scheme for solving Laplace's equation. Poisson's equation adds only a minor complication to the above equation since in this case we have

$$u_{xx} + u_{yy} = -\rho(\mathbf{x}),$$

and we need only to add a discretized version of $\rho(\mathbf{x})$ resulting in

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \rho_{i,j}. \tag{167}$$

Solution Approach

The way we solve these equations is based on an iterative scheme we discussed in connection with linear algebra, namely the so-called Jacobi, Gauss-Seidel and relaxation methods. The steps are rather simple. We start with an initial guess for $u_{i,j}^{(0)}$ where all values are known. To obtain a new solution we solve Eq. (166) or Eq. (167) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (166). This solution is in turn used to obtain a new and improved $u_{i,j}^{(2)}$. We continue this process till we obtain a result which satisfies some specific convergence criterion.

Code example for the two-dimensional diff equation/Laplace

```
int DiffusionJacobi(int N, double dx, double dt,
                    double **A, double **q,
                      double abstol){
  int i,j,k;
  int maxit = 100000;
  double sum;
  double ** Aold = CreateMatrix(N,N);

  double D = dt/(dx*dx);
```

Code example for the two-dimensional diff equation/Laplace

```
for(i=1; i<N-1; i++)
  for(j=1;j<N-1;j++)
    Aold[i][j] = 1.0;
/* Boundary Conditions -- all zeros */
for(i=0;i<N;i++){
  A[0][i] = 0.0;
  A[N-1][i] = 0.0;
  A[i][0] = 0.0;
  A[i][N-1] = 0.0;
}
```

Code example for the two-dimensional diff equation/Laplace

```
for(k=0; k<maxit; k++){
  for(i = 1; i<N-1; i++){
    for(j=1; j<N-1; j++){
      A[i][j] = dt*q[i][j] + Aold[i][j] +
        D*(Aold[i+1][j] + Aold[i][j+1] - 4.0*Aold
            [i][j] +
           Aold[i-1][j] + Aold[i][j-1]);
    }
  }
  sum = 0.0;
  for(i=0;i<N;i++){
    for(j=0;j<N;j++){
      sum += (Aold[i][j]-A[i][j])*(Aold[i][j]-A[i
          ][j]);
      Aold[i][j] = A[i][j];
    }
  }
  if(sqrt(sum)<abstol){DestroyMatrix(Aold,N,N);
    return k;
```

Code example for the parallel two-dimensional diff equation

```
int Jacobi_P(int mynode, int numnodes, int N,
    double **A, double *x, double *b, double abstol
    ){
  int i,j,k,i_global;
  int maxit = 100000;
  int rows_local,local_offset,last_rows_local,*
      count,*displacements;
  double sum1,sum2,*xold;
  double error_sum_local, error_sum_global;
  MPI_Status status;

  rows_local = (int) floor((double)N/numnodes);
  local_offset = mynode*rows_local;
  if(mynode == (numnodes-1))
    rows_local = N - rows_local*(numnodes-1);
```

Code example for the parallel two-dimensional diff equation

```
/* Distribute the Matrix and R.H.S. among the
   processors */
if(mynode == 0){
  for(i=1;i<numnodes-1;i++){
    for(j=0;j<rows_local;j++)
      MPI_Send(A[i*rows_local+j],N,MPI_DOUBLE,i,j
             ,MPI_COMM_WORLD);
    MPI_Send(b+i*rows_local,rows_local,MPI_DOUBLE
          ,i,rows_local,
                MPI_COMM_WORLD);
  }
  last_rows_local = N-rows_local*(numnodes-1);
  for(j=0;j<last_rows_local;j++)
    MPI_Send(A[(numnodes-1)*rows_local+j],N,
          MPI_DOUBLE,numnodes-1,j,
                MPI_COMM_WORLD);
  MPI_Send(b+(numnodes-1)*rows_local,
        last_rows_local,MPI_DOUBLE,numnodes-1,
              last_rows_local,MPI_COMM_WORLD);
```

Code example for the parallel two-dimensional diff equation

```
else{
   A = CreateMatrix(rows_local,N);
   x = new double[rows_local];
   b = new double[rows_local];
   for(i=0;i<rows_local;i++)
      MPI_Recv(A[i],N,MPI_DOUBLE,0,i,MPI_COMM_WORLD
           ,&status);
   MPI_Recv(b,rows_local,MPI_DOUBLE,0,rows_local,
        MPI_COMM_WORLD,&status);
}
```

Code example for the parallel two-dimensional diff equation

```
xold = new double[N];
count = new int[numnodes];
displacements = new int[numnodes];
//set initial guess to all 1.0
for(i=0; i<N; i++){
  xold[i] = 1.0;
}
for(i=0;i<numnodes;i++){
  count[i] = (int) floor((double)N/numnodes);
  displacements[i] = i*count[i];
}
count[numnodes-1] = N - ((int)floor((double)N/
    numnodes))*(numnodes-1);
```

Code example for the parallel two-dimensional diff equation

```
for(k=0; k<maxit; k++){
  error_sum_local = 0.0;
  for(i = 0; i<rows_local; i++){
    i_global = local_offset+i;
    sum1 = 0.0; sum2 = 0.0;
    for(j=0; j < i_global; j++)
      sum1 = sum1 + A[i][j]*xold[j];
    for(j=i_global+1; j < N; j++)
      sum2 = sum2 + A[i][j]*xold[j];

    x[i] = (-sum1 - sum2 + b[i])/A[i][i_global];
    error_sum_local += (x[i]-xold[i_global])*(x[i
       ]-xold[i_global]);
  }
```

Code example for the parallel two-dimensional diff equation

```
MPI_Allreduce(&error_sum_local,&
    error_sum_global,1,MPI_DOUBLE,
                MPI_SUM,MPI_COMM_WORLD);
MPI_Allgatherv(x,rows_local,MPI_DOUBLE,xold,
    count,displacements,
                MPI_DOUBLE,MPI_COMM_WORLD);
```

Code example for the parallel two-dimensional diff equation

```
if(sqrt(error_sum_global)<abstol){
  if(mynode == 0){
    for(i=0;i<N;i++)
      x[i] = xold[i];
  }
  else{
    DestroyMatrix(A,rows_local,N);
    delete[] x;
    delete[] b;
  }
  delete[] xold;
  delete[] count;
  delete[] displacements;
  return k;
}
}
```

Code example for the parallel two-dimensional diff equation

```
cerr << "Jacobi: Maximum Number of Interations
    Reached Without Convergence\n";
if(mynode == 0){
  for(i=0;i<N;i++)
    x[i] = xold[i];
}
else{
  DestroyMatrix(A,rows_local,N);
  delete[] x;
  delete[] b;
}
delete[] xold;
delete[] count;
delete[] displacements;

return maxit;
}
```

Week 48

Summary and exam discussion

- Monday:
- Brief repetition from last week and discussion of projects
- Wave equation with examples
- Wednesday:
- Summary and exam discussion

Two-dimensional wave equation

Consider first the two-dimensional wave equation for a vibrating square membrane given by the following initial and boundary conditions

$$\begin{cases} \lambda \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial^2 u}{\partial t^2} & x, y \in [0, 1], t \geq 0 \\ u(x, y, 0) = sin(\pi x) sin(2\pi y) & x, y \in (0, 1) \\ u = 0 \;\; \text{boundary} & t \geq 0 \\ \partial u / \partial t|_{t=0} = 0 & x, y \in (0, 1) \end{cases} .$$

The boundary is defined by $x = 0$, $x = 1$, $y = 0$ and $y = 1$. Here we set $\lambda = 1$.

Two-dimensional wave equation

Our equations depend on three variables whose discretized versions are now

$$\begin{cases} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{cases}, \tag{168}$$

and we will let $\Delta x = \Delta y = h$ and $n_x = n_y$ for the sake of simplicity. We have now the following discretized partial derivatives

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2}, \tag{169}$$

and

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}, \tag{170}$$

and

$$u_{tt} \approx \frac{u^{l+1}_{i,j} - 2u^l_{i,j} + u^{l-1}_{i,j}}{\Delta t^2}. \tag{171}$$

Two-dimensional wave equation

We merge this into the discretized $2+1$-dimensional wave equation as

$$u_{i,j}^{l+1} = 2u_{i,j}^{l} - u_{i,j}^{l-1} + \frac{\Delta t^2}{h^2}\left(u_{i+1,j}^{l} - 4u_{i,j}^{l} + u_{i-1,j}^{l} + u_{i,j+1}^{l} + u_{i,j-1}^{l}\right), \tag{172}$$

where again we have an explicit scheme with $u_{i,j}^{l+1}$ as the only unknown quantity. It is easy to account for different step lengths for x and y. The partial derivative is treated in much the same way as for the one-dimensional case, except that we now have an additional index due to the extra spatial dimension, viz., we need to compute $u_{i,j}^{-1}$ through

$$u_{i,j}^{-1} = u_{i,j}^{0} + \frac{\Delta t}{2h^2}\left(u_{i+1,j}^{0} - 4u_{i,j}^{0} + u_{i-1,j}^{0} + u_{i,j+1}^{0} + u_{i,j-1}^{0}\right), \tag{173}$$

in our setup of the initial conditions.

Code example for the two-dimensional wave equation

We show here how to implement the two-dimensional wave equation

```
//  After initializations and declaration of
    variables
for ( int i = 0; i < n; i++ ) {
  u[i] = new double [n];
  uLast[i] = new double [n];
  uNext[i] = new double [n];
  x[i] = i*h;
  y[i] = x[i];
}
// initializing
for ( int i = 0; i < n; i++ ) {  // setting
    initial step
  for ( int j = 0; j < n; j++ ) {
    uLast[i][j] = sin(PI*x[i])*sin(2*PI*y[j]);
  }
}
```

Code example for the two-dimensional wave equation

```
for ( int i = 1; i < (n-1); i++ ) {  // setting
    first step using the initial derivative
  for ( int j = 1; j < (n-1); j++ ) {
    u[i][j] = uLast[i][j] - ((tStep*tStep)/(2.0*h
        *h))*
      (4*uLast[i][j] - uLast[i+1][j] - uLast[i
          -1][j] - uLast[i][j+1] - uLast[i][j-1])
          ;
  }
  u[i][0] = 0;  // setting boundaries once and
      for all
  u[i][n-1] = 0;
  u[0][i] = 0;
  u[n-1][i] = 0;

  uNext[i][0] = 0;
  uNext[i][n-1] = 0;
  uNext[0][i] = 0;
  uNext[n-1][i] = 0;
}
```

Code example for the two-dimensional wave equation

```
// iterating in time
double t = 0.0 + tStep;
int iter = 0;

while ( t < tFinal ) {
  iter ++;
  t = t + tStep;

  for ( int i = 1; i < (n-1); i++ ) {  //
     computing next step
    for ( int j = 1; j < (n-1); j++ ) {
      uNext[i][j] = 2*u[i][j] - uLast[i][j] - ((
          tStep*tStep)/(h*h))*
        (4*u[i][j] - u[i+1][j] - u[i-1][j] - u[i
            ][j+1] - u[i][j-1]);
    }
  }
```

Code example for the two-dimensional wave equation

```
for ( int i = 1; i < (n-1); i++ ) {   //
    shifting results down
  for ( int j = 1; j < (n-1); j++ ) {
    uLast[i][j] = u[i][j];
    u[i][j] = uNext[i][j];
  }
}
}
```

Closed form solution of the wave equation

We develop here the analytic solution for the $2+1$ dimensional wave equation with the following boundary and initial conditions

$$\begin{cases} c^2(u_{xx}+u_{yy}) = u_{tt} & x, y \in (0, L), t > 0 \\ u(x, y, 0) = f(x, y) & x, y \in (0, L) \\ u(0, 0, t) = u(L, L, t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = g(x, y) & x, y \in (0, L) \end{cases} .$$

Closed form solution of the wave equation

Our first step is to make the ansatz

$$u(x, y, t) = F(x, y)G(t),$$

resulting in the equation

$$FG_{tt} = c^2(F_{xx}G + F_{yy}G),$$

or

$$\frac{G_{tt}}{c^2 G} = \frac{1}{F}(F_{xx} + F_{yy}) = -\nu^2.$$

The lhs and rhs are independent of each other and we obtain two differential equations

$$F_{xx} + F_{yy} + F\nu^2 = 0,$$

and

$$G_{tt} + Gc^2\nu^2 = G_{tt} + G\lambda^2 = 0,$$

with $\lambda = c\nu$.

Closed form solution of the wave equation

We can in turn make the following ansatz for the x and y dependent part

$$F(x, y) = H(x)Q(y),$$

which results in

$$\frac{1}{H}H_{xx} = -\frac{1}{Q}(Q_{yy} + Q\nu^2) = -\kappa^2.$$

Since the lhs and rhs are again independent of each other, we can separate the latter equation into two independent equations, one for x and one for y, namely

$$H_{xx} + \kappa^2 H = 0,$$

and

$$Q_{yy} + \rho^2 Q = 0,$$

with $\rho^2 = \nu^2 - \kappa^2$.

Closed form solution of the wave equation

The second step is to solve these differential equations, which all have trigonometric functions as solutions, viz.

$$H(x) = A\cos(\kappa x) + B\sin(\kappa x),$$

and

$$Q(y) = C\cos(\rho y) + D\sin(\rho y).$$

The boundary conditions require that $F(x,y) = H(x)Q(y)$ are zero at the boundaries, meaning that $H(0) = H(L) = Q(0) = Q(L) = 0$. This yields the solutions

$$H_m(x) = \sin(\frac{m\pi x}{L}) \qquad Q_n(y) = \sin(\frac{n\pi y}{L}),$$

or

$$F_{mn}(x,y) = \sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

Closed form solution of the wave equation

With $\rho^2 = \nu^2 - \kappa^2$ and $\lambda = c\nu$ we have an eigenspectrum $\lambda = c\sqrt{\kappa^2 + \rho^2}$ or $\lambda_{mn} = c\pi/L\sqrt{m^2 + n^2}$. The solution for G is

$$G_{mn}(t) = B_{mn}\cos(\lambda_{mn}t) + D_{mn}\sin(\lambda_{mn}t),$$

with the general solution of the form

$$u(x,y,t) = \sum_{mn=1}^{\infty} u_{mn}(x,y,t) = \sum_{mn=1}^{\infty} F_{mn}(x,y)G_{mn}(t).$$

Closed form solution of the wave equation

The final step is to determine the coefficients B_{mn} and D_{mn} from the Fourier coefficients. The equations for these are determined by the initial conditions $u(x,y,0) = f(x,y)$ and $\partial u/\partial t|_{t=0} = g(x,y)$. The final expressions are

$$B_{mn} = \frac{2}{L}\int_0^L\int_0^L dxdyf(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}),$$

and

$$D_{mn} = \frac{2}{L}\int_0^L\int_0^L dxdyg(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

Inserting the particular functional forms of $f(x,y)$ and $g(x,y)$ one obtains the final analytic expressions.

Two-dimensional wave equation

We can check our results as function of the number of mesh points and in particular against the stability condition

$$\Delta t \leq \frac{1}{\sqrt{\lambda}} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$$

where Δt, Δx and Δy are the chosen step lengths. In our case $\Delta x = \Delta y = h$. How do we find this condition? In one dimension we can proceed as we did for the diffusion equation.

Two-dimensional wave equation

The analytic solution of the wave equation in $2+1$ dimensions has a characteristic wave component which reads

$$u(x,y,t) = A\exp\left(i(k_x x + k_y y - \omega t)\right)$$

Then from

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{\Delta x^2},$$

we get, with $u_i = \exp ikx_i$

$$u_{xx} \approx \frac{u_i}{\Delta x^2}\left(\exp ik\Delta x - 2 + \exp\left(-ik\Delta x\right)\right),$$

or

$$u_{xx} \approx 2\frac{u_i}{\Delta x^2}\left(cos(k\Delta x) - 1\right) = -4\frac{u_i}{\Delta x^2} sin^2(k\Delta x/2)$$

We get similar results for t and y.

Two-dimensional wave equation

We have

$$\lambda\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=\frac{\partial^2 u}{\partial t^2},$$

resulting in

$$\lambda\left(-4\frac{u_{ij}^l}{\Delta x^2}\sin^2\left(k_x\Delta x/2\right)-4\frac{u_{ij}^l}{\Delta y^2}\sin^2\left(k_y\Delta y/2\right)\right)=-4\frac{u_{ij}^l}{\Delta t^2}\sin^2\left(\omega\Delta t/2\right),$$

resulting in

$$\sin\left(\omega\Delta t/2\right)=\pm\sqrt{\lambda}\Delta t\left(\frac{1}{\Delta x^2}\sin^2\left(k_x\Delta x/2\right)+\frac{1}{\Delta y^2}\sin^2\left(k_y\Delta y/2\right)\right)^{1/2}.$$

The squared sine functions can at most be unity. The frequency ω is real and our wave is neither damped nor amplified.

Two-dimensional wave equation

We have

$$\sin(\omega\Delta t/2) = \pm\sqrt{\lambda}\Delta t \left(\frac{1}{\Delta x^2}\sin^2(k_x\Delta x/2) + \frac{1}{\Delta y^2}\sin^2(k_y\Delta y/2)\right)^{1/2}.$$

The squared sine functions can at most be unity. ω is real and our wave is neither damped nor amplified. The numerical ω must also be real which is the case when $\sin(\omega\Delta t/2)$ is less than or equal to unity, meaning that

$$\Delta t \leq \frac{1}{\sqrt{\lambda}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2}.$$

Two-dimensional wave equation

We modify now the wave equation in order to consider a $2+1$ dimensional wave equation with a position dependent velocity, given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (\lambda(x,y)\nabla u).$$

If λ is constant, we obtain the standard wave equation discussed in the two previous points. The solution $u(x,y,t)$ could represent a model for water waves. It represents then the surface elevation from still water. We can model λ as

$$\lambda = gH(x,y),$$

with g being the acceleration of gravity and $H(x,y)$ is the still water depth.

The function $H(x,y)$ simulates the water depth using for example measurements of still water depths in say a fjord or the north sea. The boundary conditions are then determined by the coast lines as discussed in point d) below. We have assumed that the vertical motion is negligible and that we deal with long wavelenghts $\tilde{\lambda}$ compared with the depth of the sea H, that is $\tilde{\lambda}/H \gg 1$. We neglect normally Coriolis effects.

Two-dimensional wave equation

You can discretize

$$\nabla \cdot (\lambda(x,y)\nabla u) = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right),$$

as follows using again a quadratic domain for x and y:

$$\frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) \approx \frac{1}{\Delta x}\left(\lambda_{i+1/2,j}\left[\frac{u^l_{i+1,j} - u^l_{i,j}}{\Delta x}\right] - \lambda_{i-1/2,j}\left[\frac{u^l_{i,j} - u^l_{i-1,j}}{\Delta x}\right]\right),$$

and

$$\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right) \approx \frac{1}{\Delta y}\left(\lambda_{i,j+1/2}\left[\frac{u^l_{i,j+1} - u^l_{i,j}}{\Delta y}\right] - \lambda_{i,j-1/2}\left[\frac{u^l_{i,j} - u^l_{i,j-1}}{\Delta y}\right]\right).$$

Two-dimensional wave equation

How did we do that? Look at the derivative wrt x only:
First we compute the derivative

$$\frac{d}{dx}\left(\lambda(x)\frac{du}{dx}\right)|_{x=x_i} \approx \frac{1}{\Delta x}\left(\lambda\frac{du}{dx}|_{x=x_{i+1/2}} - \lambda\frac{du}{dx}|_{x=x_{i-1/2}}\right),$$

where we approximated it at the midpoint by going half a step to the right and half a step to the left. Then we approximate

$$\lambda\frac{du}{dx}|_{x=x_{i+1/2}} \approx \lambda_{x_{i+1/2}}\frac{u_{i+1}-u_i}{\Delta x},$$

and similarly for $x = x_i - 1/2$.

Two-dimensional wave equation, Tsunami model

We assume that we can approximate the coastline with a quadratic grid. As boundary condition at the coastline we will employ

$$\frac{\partial u}{\partial n} = \nabla u \cdot \mathbf{n} = 0,$$

where $\partial u/\partial n$ is the derivative in the direction normal to the boundary.

Here you must pay particular attention to the endpoints.

Two-dimensional wave equation

We are going to model the impact of an earthquake on sea water. This is normally modelled via an elevation of the sea bottom. We will assume that the movement of the sea bottom is very rapid compared with the period of the propagating waves. This means that we can approximate the bottom elevation with an initial surface elevation. The initial conditions are then given by (with L the length of the grid)

$$u(x, y, 0) = f(x, y) \qquad x, y \in (0, L),$$

and

$$\partial u/\partial t|_{t=0} = 0 \qquad x, y \in (0, L).$$

We will approximate the initial elevation with the function

$$f(x, y) = A_0 \exp\left(-\left[\frac{x - x_c}{\sigma_x}\right]^2 - \left[\frac{y - y_c}{\sigma_y}\right]^2\right),$$

where A_0 is the elevation of the surface and is typically $1 - 2$ m. The variables σ_x and σ_y represent the extensions of the surface elevation. Here we let $\sigma_x = 80$ km and $\sigma_y = 200$ km. The 2004 tsunami had extensions of approximately 200 and 1000 km, respectively.

The variables x_c and y_c represent the epicentre of the earthquake.

Exam FYS3150

Pensum/syllabus

Go to `http://www.uio.no/studier/emner/matnat/fys/FYS3150/h11/` and click on syllabus.

Exam FYS3150

Topics

- Linear algebra and eigenvalue problems. (Lecture notes chapters 6.1-6.5 and 7.1-7.5 and projects 1 and 2).
- Numerical integration, standard methods and Monte Carlo methods (Lecture notes chapters 5.1-5.5 and 11 and project 3).
- Monte Carlo methods in physics (Lecture notes chapters 11, 12 and 14, project 4 and project 5)
- Ordinary differential equations (Lecture notes chapters 8 9.1-9.3 and project 5)
- Partial differential equations (Lecture notes chapter 10 and project 5)

Exam FYS3150

Linear algebra and eigenvalue problems, chapters 6.1-6.5 and 7.1-7.5

- Know Gaussian elimination and LU decomposition (project 1)
- How to solve linear equations (project 1)
- How to obtain the inverse and the determinant of a real symmetric matrix
- Cubic spline
- Tridiagonal matrix decomposition (project 1)
- Householder's tridiagonalization technique and finding eigenvalues based on this
- Jacobi's method for finding eigenvalues (project 2)

Exam FYS3150

Numerical integration, standard methods and Monte Carlo methods (5.1-5.5 and 11)

- Trapezoidal, rectangle and Simpson's rules
- Gaussian quadrature, emphasis on Legendre polynomials, but you need to know about other polynomials as well (project 3).
- Brute force Monte Carlo integration (project 3)
- Random numbers (simplest algo, ran0) and probability distribution functions, expectation values
- Improved Monte Carlo integration and importance sampling (project 3).

Exam FYS3150

Monte Carlo methods in physics (12 and 14)

- Random walks and Markov chains and relation with diffusion equation (project 4)
- Metropolis algorithm, detailed balance and ergodicity (project 4)
- Applications to quantum mechanical systems (project 5)

Exam FYS3150

Ordinary differential equations (Chapter 8)

- Euler's method and improved Euler's method, truncation errors (project 5)
- Runge Kutta methods, 2nd and 4th order, truncation errors (project 5)
- How to implement a second-order differential equation, both linear and non-linear. How to make your equations dimensionless.

Exam FYS3150

Partial differential equations, chapter 10

- Set up diffusion, Poisson and wave equations up to 2 spatial dimensions and time
- Set up the mathematical model and algorithms for these equations, with boundary and initial conditions. The stability conditions for the diffusion equation.
- Explicit, implicit and Crank-Nicolson schemes, and how to solve them. Remember that they result in triangular matrices.
- How to compute the Laplacian in Poisson's equation.
- How to solve the wave equation in one and two dimensions using an explicit scheme.

Other courses in Computational Science at UiO

Bachelor/Master/PhD Courses

- INF-MAT4350 Numerical linear algebra
- MAT-INF3300/3310/4300/4310, PDEs and Sobolev spaces I and II
- INF-MAT3360 Partial differential equation
- INF3380 Parallell programming for scientific problems
- INF5620/5630 Numerical methods for PDEs, finite element method
- FYS4411 Computational physics II, computational quantum mechanics.
- FYS4460: Computational statistical mechanics

AND, GOOD LUCK TO YOU ALL!