Slides from FYS3150/FYS4150 Lectures

Morten Hjorth-Jensen

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

Fall 2007

Week 34, 20-24 August

- 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: thursday and friday 9am-7pm. Presentation of hardware and software at room FV329. Exercises 1 and 2.

Lectures and ComputerLab

- Lectures: monday (12.15pm-2pm) and wednesday (12.15pm-14pm)
- Detailed lecture notes, exercises, all programs presented, projects etc can be found at the homepage of the course.
- Computerlab: 9am-7 pm thursday and friday, room FV329. Four groups. Each group has four hours at its disposal.
- Weekly plans and all other information are on the official webpage.

Course Format

- Several computer exercises, 5 compulsory projects. Electronic reports only. Fronter as course organizer `http://blyant.uio.no`.
- Oral examination based on chosen report from one of the projects 3-5 and five selected topics, see the syllabus link on the webpage. Dates to be settled, most likely start, 10 December with end 14 december.
- The computer lab (room FV329)consists of 16 Linux PCs. C/C++ is the default programming language, but Fortran95 is also used. All source codes discussed during the lectures can be found at the webpage of the course. Alternatively you can also use Python or compiled languages like Matlab. We recommend either C/C++, Fortran95 or Python as languages.

ComputerLab

day	teacher
Thursday 9am-1 pm	Filip Nicolaisen
Thursday 1pm-5pm	Filip Nicolaisen
Friday 9am-1pm	Marius Lysebo
Fredag 1pm-5pm	Marius Lysebo

The lab is open till 7pm, from 4pm till 7pm thursday and friday a student assistant will be present.
Set up your preferred lab time today, see separate list.

Topics covered

- 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

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.

Limits

C++ and Fortran declarations

type in C/C++ and Fortran 90/95	bits	range
char/CHARACTER	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
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	$3.4e^{-38}$ to $3.4e^{+38}$
double/REAL(8)	64	$1.7e^{-308}$ to $1.7e^{+308}$
long double	64	$1.7e^{-308}$ to $1.7e^{+308}$

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 $(417)_{10} = (110100001)_2$ since we have

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

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=27	remainder 0	coefficient of 2^3 is 0
26/2=13	remainder 0	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()
```

Brief summary from Monday

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 return 0 (everything correct) or 1 (something went wrong)
- Standard **if**, **while** and **for** statements as in Java
- Integers have a very limited range.

Brief summary from Monday

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.

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

$$x = 1.234567891112131468 = 0.1234567891112131468 \times 10^1 \qquad (1)$$

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.

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

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

$$x = \pm r \times 10^{\mathrm{n}}, \tag{2}$$

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

$$x = \pm q \times 2^{\mathrm{m}}, \tag{3}$$

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

$$(0.a_{-1}a_{-2}\dots a_{-n})_2 = a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \dots + a_{-n} \times 2^{-n}. \tag{4}$$

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

$$x = (-1)^s \times \text{mantissa} \times 2^{\text{exponent}}, \tag{5}$$

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

$$(1.a_{-1}a_{-2}\dots a_{-23})_2 = 1 \times 2^0 + a_{-1} \times 2^{-1} + +a_{-2} \times 2^{-2} + \dots + a_{-n} \times 2^{-23}. \quad (6)$$

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

$$fl(x) = x(1 \pm \epsilon_x), \qquad (7)$$

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

$$fl(x) = x(1+\epsilon) \tag{8}$$

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

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

or

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

meaning that

$$fl(a)/a = 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a}, \tag{11}$$

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

$$|fl(a) - a|, \tag{12}$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a. \tag{13}$$

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - f(c) - (b - c)|}{a}, \tag{14}$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}. \tag{15}$$

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 Precision

To understand roundoff errors in general, it is customary to regard it as a random process. One can represent these errors by a semi-empirical expression if the roundoff errors come in randomly up or down

$$\epsilon_{ro} \approx \sqrt{N}\epsilon_M, \tag{16}$$

where N is e.g., the number of terms in the summation over n for the exponential. Note well that this estimate can be wrong especially if the roundoff errors accumulate in one specific direction. One special case is that of subtraction of two almost equal numbers. The total error will then be the sum of a roundoff error and an approximation error of the algorithm. The latter would correspond to the truncation test of examples 2 and 3. Let us assume that the approximation error goes like

$$\epsilon_{approx} = \frac{\alpha}{N^\beta}, \tag{17}$$

with the obvious limit $\epsilon_{approx} \rightarrow 0$ when $N \rightarrow \infty$. The total error reads then

$$\epsilon_{tot} = \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_M \tag{18}$$

Week 35, 27 August - 31 August

- Monday: Repetition from last week
- Numerical differentiation
- C/C++ programming details, pointers, read/write to/from file
- Wednesday: Intro to linear Algebra and presentation of project 1.
- Matrices in C++ and Fortran90/95
- Dynamic memory allocation in C/C++ and Fortran90/95 , use of the library package Blitz++ for C++ users
- Computer-Lab: thursday and friday 9am-7pm, Exercise 3 and presentation of Blitz++.

Message: Revised lecture notes (chapters 2-15) will be put on webpage september 5 2007.

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

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

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

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

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

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

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

It is then natural to ask which value of h yields the smallest total error. Taking the derivative of $|\epsilon_{\mathrm{tot}}|$ 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, still 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	$0.0000000000000000 \times 10^{0}$

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

```
 using namespace std; // note use of namespace
 int main()
{
  int var;
  int *p;
  p = &var;
  var  = 421;
  printf("Address of integer variable var : %p\n",&var);
  printf("Its value: %d\n", var);
  printf("Value of integer pointer p : %p\n",p);
  printf("The value p points at :  %d\n",*p);
  printf("Address of the pointer p : %p\n",&p);
  return 0;
}
```

Pointer example I

Discussion

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

Pointer example II

```
      ....
int matr[2];
int *p;
p = &matr[0];
matr[0] = 321;
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 $\mathbf{Ax} = 0$ 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

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij}, \tag{19}$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma\mathbf{B} \Longrightarrow a_{ij} = \gamma b_{ij}, \tag{20}$$

vector-matrix multiplication

$$\mathbf{y} = \mathbf{A}\mathbf{x} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij}x_j, \tag{21}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{B}\mathbf{C} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik}c_{kj}, \tag{22}$$

and transposition

$$\mathbf{A} = \mathbf{B}^T \Longrightarrow a_{ij} = b_{ji} \tag{23}$$

Important Mathematical Operations

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

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \Longrightarrow x_i = y_i \pm z_i, \tag{24}$$

scalar-vector multiplication

$$\mathbf{x} = \gamma \mathbf{y} \Longrightarrow x_i = \gamma y_i, \tag{25}$$

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{y}\mathbf{z} \Longrightarrow x_i = y_i z_i, \tag{26}$$

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

$$x = \mathbf{y}^T \mathbf{z} \Longrightarrow x = \sum_{j=1}^{n} y_j z_j, \tag{27}$$

and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \Longrightarrow a_{ij} = y_i z_j, \tag{28}$$

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, 3-7 September

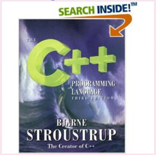
Linear Algebra

- Monday: Repetition from last week
- Presentation of Project 1, deadline 17 september (midnight).
- Gausssian elimination, LU decomposition and linear equations
- Wednesday: Further discussion of linear algebra methods, numerical stability
- Cholesky decomposition, triangular matrices, inverse of a matrix
- Classes in C++ (From INF-VERK3830 slides)
- 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.
- Computer-Lab: thursday and friday 9am-7pm, Project 1.

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/`

JOHN J. BARTON
LEE R. NACKMAN
Scientific and Engineering
C++
An Introduction with Advanced Techniques and Examples



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

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

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

$$x_m = \frac{1}{b_{mm}}\left(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\right) \qquad m = n-1, n-2, \dots, 1. \tag{29}$$

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 **A** is not singular.

Gaussian Elimination

Our actual 4×4 example reads after the first operation

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

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= \quad y_1 \\ a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= \quad w_2^{(2)} \\ a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 &= \quad w_3^{(2)} \\ a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 &= \quad w_4^{(2)}, \end{aligned} \tag{30}$$

Gaussian Elimination

The new coefficients are

$$b_{1k} = a_{1k}^{(1)} \quad k = 1, \dots, n, \tag{31}$$

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

$$a_{jk}^{(2)} = a_{jk}^{(1)} - \frac{a_{j1}^{(1)} a_{1k}^{(1)}}{a_{11}^{(1)}} \quad j, k = 2, \dots, n, \tag{32}$$

with a new right-hand side given by

$$y_1 = w_1^{(1)}, \quad w_j^{(2)} = w_j^{(1)} - \frac{a_{j1}^{(1)} w_1^{(1)}}{a_{11}^{(1)}} \quad j = 2, \dots, n. \tag{33}$$

We have also set $w_1^{(1)} = w_1$, the original vector element. We see that the system of unknowns $x_1, \dots, x_n$ 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

$$a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)} a_{mk}^{(m)}}{a_{mm}^{(m)}} \quad j, k = m+1, \dots, n, \tag{34}$$

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

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)} w_m^{(m)}}{a_{mm}^{(m)}} \quad j = m+1, \dots, n. \tag{35}$$

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.

LU Decomposition

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

$$\left(\begin{array}{cccc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) = \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{array}\right) \left(\begin{array}{cccc} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{array}\right).$$

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

$$det\{\mathbf{A}\} = c_{11}c_{22}\dots c_{nn}.$$

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

$$\left(\begin{array}{cccc} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{array}\right) \rightarrow \left(\begin{array}{cccc} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{array}\right) \rightarrow \left(\begin{array}{cccc} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{array}\right)$$

and finally

$$\left(\begin{array}{cccc} d_1 & e_1 & 0 & 0 \\ 0 & \tilde{d}_2 & e_2 & 0 \\ 0 & 0 & \tilde{d}_3 & e_3 \\ 0 & 0 & 0 & \tilde{d}_4 \end{array}\right)$$

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 $k = 2, \dots, n$.

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.
- Not discussed: Singular value decomposition
- the QR method will be discussed in chapter 12 in connection with eigenvalue systems. $O(4/3n^3)$ flops.
- Basic functions in libraries are LUDCMP and LUBKSB.

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 4 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.

Blitz++ http://www.oonumerics.org/blitz/

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, 11-15 September

Numerical Integration and Monte Carlo

- Monday: Repetition from last week
- Trapezoidal and Simpson's rules
- Gaussian quadrature
- Wednesday: Gaussian quadrature continues
- Integral equations
- Intro to Monte Carlo integration
- Computer-Lab: thursday and friday 9am-7pm, Project 1.

Question: Newton studied his famous apple in 1665, but published his theory of universal gravity only in 1687. Why?

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 trapezoidal_rule
```

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

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

In the Lagrange representation this interpolation polynomial is given by

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

with the Lagrange factors

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

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 $k = 0, \dots, n$.

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

Equal Step Methods, Polynomials and Newton-Cotes

Example:

$$ln2 = \int_0^1 \frac{dx}{1+x}$$

The Trapezoidal rule gives

$$ln2 \approx \frac{1}{2}(1+\frac{1}{2}) = 0.75.$$

The error is $1/6$ (max error) and we obtain the estimate $|ln2 - 0.75| \leq 0.167$. The true error is $ln2 - 0.75 = -0.056$. Simpson's rule gives $|ln2 - 0.6944| \leq 0.0084$. The true error is $ln2 - 0.6944 = -0.0012$.
Normally we never use such large intervals. It is common to split the interval into many small sub intervals and use the composite rule.

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.
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

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

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

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

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

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

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

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0.$$

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.

Gaussian Quadrature, general Properties

The weights are positive and the sequence of Gaussian quadrature formulae is convergent if the sequence Q_n of quadrature formulae

$$Q_n(f) \rightarrow Q(f) = \int_a^b f(x)dx,$$

in the limit $n \rightarrow \infty$. Then we say that the sequence

$$Q_n(f) = \sum_{i=0}^{n} \omega_i^{(n)} f(x_i^{(n)}),$$

is convergent for all polynomials p, that is

$$Q_n(p) = Q(p)$$

if there exits a constant C such that

$$\sum_{i=0}^{n} |\omega_i^{(n)}| \leq C,$$

for all n which are natural numbers.

Gaussian Quadrature, Error

Let $f \in C^{2n+2}[a,b]$, viz the space of all real or complex $2n+2$ times continuously differentiable functions, then the error for the Gaussian quadrature formula of order n is given by

$$\int_a^b W(x)f(x)dx - \sum_{k=0}^{n} w_k f(x_k) = \frac{f^{2n+2}(\xi)}{(2n+2)!}\int_a^b W(x)[q_{n+1}(x)]^2 dx$$

where q_{n+1} is the chosen orthogonal polynomial and $\xi \in [a,b]$. See the lecture for a simple example.

Week 38, 17-21 September

Numerical Integration and Monte Carlo

- Monday: Repetition from last week
- Brief discussion of project 2.
- Monte Carlo integration
- Random numbers (details next week)
- P(robability)D(istribution)F(unctions)
- Wednesday: Monte Carlo integration continues
- How to parallelize a code
- Computer-Lab: thursday and friday 9am-7pm, Project 2.

Material covered this week: chapters 7.5 (MPI), 8.1 and 8.4
Delay in revising lecture notes, hopefully ready on wednesday.
Please send mail about wanted exam time.

Plan for Monte Carlo Lectures

- This week: intro, random numbers, MC integration and probability distribution functions (PDFs)
- Next week: random walks and statistical physics
- Third week: Statistical physics
- Fourth week: Statistical physics and quantum Monte Carlo
- Fifth and sixth week: quantum Monte Carlo

Monte Carlo Keywords

Consider it is a numerical experiment

- Be able to generate random variables following a given 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	$\{x_1, x_2, x_3, \dots, x_N\}$	$[a, b]$
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 $[-\infty, \infty]$

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 Tflops 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 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 measurement 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?

- Last week 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 $d > 2k$

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;
  return 0;
}
```

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 one can, with appropriate redefinitions of the integral be made smaller. A smaller variance yields also a smaller standard deviation.

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;
}
```

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 (project 2)

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t),$$

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
```

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.

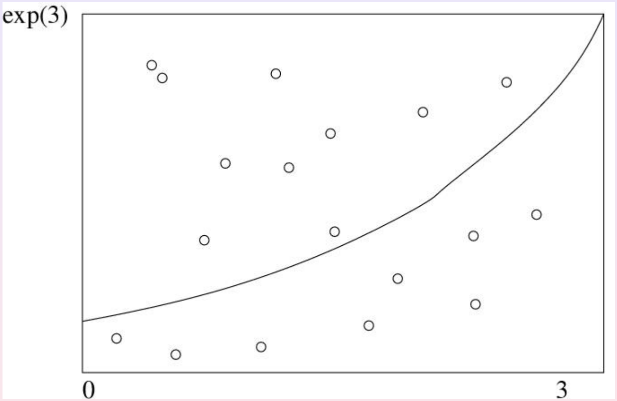
Acceptance-Rejection Method

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
           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
```

Transformation of Variables

The starting point is always the uniform distribution

$$p(x)dx = \left\{ \begin{array}{cc} dx & 0 \le x \le 1 \\ 0 & else \end{array} \right.$$

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 $x \rightarrow y$ 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

$$g(x,y) = \exp{(-(x^2+y^2)/2)}dxdy.$$

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

$$g(r,\theta) = r\exp{(-r^2/2)}drd\theta,$$

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

$$x = rcos(\theta) = \sqrt{2u}cos(\theta),$$

and

$$y = rsin(\theta) = \sqrt{2u}sin(\theta),$$

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

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 $x \rightarrow y$ 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. (178) 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);
```

Code at `http://folk.uio.no/mhjensen/fys3150/2005/programs/chapter8/example2.cpp`.

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

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 point b of project 2)

$$\prod_{i=1}^{d}(b_i - a_i),$$

and to convert the function $g(x_1, \dots, x_d)$ to

$$g(x_1, \dots, x_d) \rightarrow g(a_1 + (b_1 - a_1)t_1, \dots, a_d + (b_d - a_d)t_d).$$

Example: 6-dimensional Integral

As an example, 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

$$\pi^3\int\prod_{i=1}^{6}\left(\frac{1}{\sqrt{\pi}}\exp{(-x_i^2)}\right)\exp{(-(\mathbf{x}-\mathbf{y})^2/2)}dx_1....dx_6,$$

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);
```

Full code at `http://folk.uio.no/mhjensen/fys3150/2005/programs/chapter8/example3.cpp`.

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
```

Full code at `http://folk.uio.no/mhjensen/fys3150/2005/programs/chapter8/example4.cpp`.

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

Hints for project 2c)

Useful to change to spherical coordinates (many did this last week)

$$d\mathbf{r}_1 d\mathbf{r}_2 = r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2,$$

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

$$\frac{\exp\left(-4(r_1+r_2)\right) r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2}{\sqrt{r_1^2+r_2^2-2r_1r_2cos(\beta)}}$$

If you use the mapping $r_{1,2} \in [0,\infty)$, you should remove from the integrand $\exp\left(-(r_1+r_2)\right)$. Why not $\exp\left(-4(r_1+r_2)\right)$?

Hint for part e) (the prize part)

$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1r_2cos(\beta)}}$$

can be rewritten in terms of spherical harmonics as

$$\frac{1}{r_{12}} = \sum_{l=0}^{\infty}\sum_{m=-l}^{m=l}\frac{4\pi}{2l+1}\frac{(r_<)^l}{(r_>)^{l+1}}Y_{lm}^*(\theta_1, phi_1)Y_{lm}(\theta_2, phi_2),$$

with $r_<$ the smaller of r_1 and r_2 and $r_{>0}$ the larger. You will need the orthogonality properties of the spherical harmonics

$$\int Y_{lm}^*(\theta, phi)Y_{l'm'}(\theta, phi)d\Omega = \delta_{l,l'}\delta_{m,m'},$$

Ω being the solid angle and δ being the Kroenecker δ. I will not say more!

Week 39, 24-38 September

Random Walks, Markov Chains, Diffusion and Metropolis

- Monday: Repetition from last week
- More hints for project 2.
- Parallelization of the code for project 2
- Random numbers and their statistical properties
- Wednesday:
- Random walks and diffusion
- Markov chain Monte Carlo

EXAM: PLEASE SELECT DATE, SEE EXAM LIST ON WEBPAGE!!

Plan for Monte Carlo Lectures

- Last week: intro, MC integration and probability distribution functions (PDFs)
- This week: MPI, random numbers, random walks and statistical physics
- Third week: Statistical physics
- Fourth week: Statistical physics and quantum Monte Carlo
- Fifth and sixth week: quantum Monte Carlo

Repetition: Monte Carlo Keywords

Consider it is a numerical experiment

- Be able to generate random variables following a given 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!

The MC Philosophy in a Nutshell

- Choose the number of Monte Carlo samples N. Think of every sample as an experiment. Make a loop over N. These samples are often called Monte Carlo cycles or just samples.
- Within one experiment you may study a given physical system, say the alpha decay of 100 nuclei every day.
- You need a sampling rule. For this decay you choose a random variable from the uniform distribution with x_i in the interval $x_i \in [0, 1]$ by calling a random number generator. This number is compared with your decay probability. If smaller diminish the number of particles, if bigger keep the number. **This is the sampling rule**
- Every experiment has its mean and variance. Find the contribution to the variance and the mean value for every loop contribution.
- After N samplings, compute the final mean value, variance, standard deviation and possibly the covariance.

Repetition: Probability Distribution Functions PDF

	Discrete PDF	continuous PDF
Domain	$\{x_1, x_2, x_3, \dots, x_N\}$	$[a, b]$
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	$0 \le p(x_i) \le 1$	$p(x) \ge 0$
Positivity	$0 \le P_i \le 1$	$0 \le P(x) \le 1$
Monotonuous	$P_i \ge P_j$ if $x_i \ge x_j$	$P(x_i) \ge P(x_j)$ if $x_i \ge x_j$
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 $[-\infty,\infty]$ Standard normal distribution has $\mu=0$ and $\sigma^2=1$.

Going Parallel with MPI

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
```

Message Passing Interface (MPI) and the local cluster

Basic info for C/C++ users

The basic instructions for using the local cluster are at `http://folk.uio.no/jonkni/gvd/workdoc/mpich2-fys/mpich2-fys.html`. Follow these instructions in detail. The file 'machines' can be pulled down from MHJ's part under the webpage of the course, see programs and MPI programs. Go to chapter 7 and pull down the file machines.

```
Use the example code program2.cpp
compile and load with
mpicxx -O2 -o prog2.x  program2.cpp
run with 20 nodes
mpiexec -np 20 ./progx.x
```

Message Passing Interface (MPI) and the local cluster

Basic info for Fortran95 users

```
Use the example code calc_pi.f90
compile and load with
mpif90 -O2 -o calcpi.exe  calc_pi.f90
run with 20 nodes
mpirun -np 20 ./calpi.exe
```

Message Passing Interface (MPI) and the local cluster

Profiling

For both Fortran and C/C++ users it is recommended that you compile with the profiling option $-pg$ in the beginning. This allows you to profile the code and figure out possible bottlenecks of CPU consumption.

```
mpicxx -O2 -pg -o prog2.x  program2.cpp
```

Your codes produces the profiling info in *gmon*.*out*. Run thereafter

```
gprof prog2.x  >  profileinfo
```

The file *profileinfo* contains then information about the CPU consumption of the individual functions. The strategy is then to start with the most time-consuming functions in order to see where to improve. When you are done, you should remove the $-pg$ option. Saves CPU time.

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 7.5 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.
- A recent trend is ccNUMA (cache-coherent-non-uniform-memory- access) systems which are clusters of SMP (symmetric multi-processing) machines and have a virtual shared memory.

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.

Different parallel programming paradigms

- **Message-passing** all involved processors have an independent memory address space. The user is responsible for partition- ing the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.
- This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult.

SPMD

Although message-passing programming supports MIMD, it suffices with an SPMD (single-program-multiple-data) model, which is flexible enough for practical cases:

- Same executable for all the processors.
- Each processor works primarily with its assigned local data.
- Progression of code is allowed to differ between synchronization points.
- Possible to have a master/slave model. The standard option in Monte Carlo calculations and numerical integration.

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 pro cessing) 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 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.

Some useful concepts

- Speed-up
$$S(P) = \frac{T(1)}{T(P)}$$
$T(1)$ is the time for one processor while $T(P)$ is the time for P processors.
- Efficiency
$$\eta(P) = \frac{S(P)}{P}$$
- Latency and bandwidth – the cost model of sending a message of length L between two processors:

$$t_C(L) = \tau + \beta L.$$

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,

$$T(P) \geq \frac{T(1)}{P} \rightarrow S(P) \leq P$$

However, superlinear speed-up ($S(P) > P$) may sometimes occur due to for example cache effects.

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.
- However, we only consider situations where one processor is responsible for one process and therefore use the two terms interchangeably.

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 6 most important 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_Send - send a message to a particular pro cess within an MPI communicator
- MPI_Recv - receive a message from a particular pro cess within an MPI communicator

The first MPI C/C++ program

Let every process write ”Hello world” on the standard output.
This is program2.cpp

```
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 <<
     << numprocs << endl;
//  End MPI
MPI_Finalize ();
```

The Fortran program

```
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 o
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 7.5 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.

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 <<
<< 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 dat
                  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.

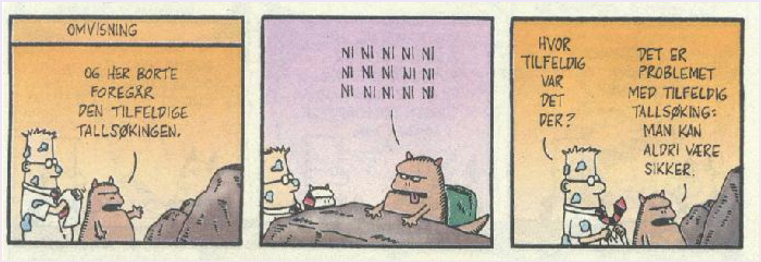
Note 4

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 da
            int tag, MPI_Comm comm, MPI_Status *sta
```

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.

Random Numbers


OMVISNING
OG HER BORTE FOREGÅR DEN TILFELDIGE TALLSØKINGEN.
NI NI NI NI NI NI NI NI NI NI NI NI NI NI NI
HVOR TILFELDIG VAR DET DER?
DET ER PROBLEMET MED TILFELDIG TALLSØKING: MAN KAN ALDRI VÆRE SIKKER.

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 $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, ... \dots,$ 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 $11, 38, 11, 38, 11, 38, \dots,$ 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,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M). \tag{36}$$

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

$$N_l = (N_{l-i}) \oplus (N_{l-j}) \quad (37)$$

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 with N points we have that the average $\langle x^k \rangle$ is

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i),$$

and taking the limit $N \rightarrow \infty$ 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

Random Numbers

Since our random numbers, which are typically generated via a linear congruential algorithm, are never fully independent, we can then define an important test which measures the degree of correlation, namely the so-called auto-correlation function C_k

$$C_k = \frac{\langle x_{i+k}x_i\rangle - \langle x_i\rangle^2}{\langle x_i^2\rangle - \langle x_i\rangle^2},$$

with $C_0 = 1$. Recall that $\sigma^2 = \langle x_i^2\rangle - \langle x_i\rangle^2$. The non-vanishing of C_k for $k \neq 0$ means that the random numbers are not independent. The independence of the random numbers is crucial in the evaluation of other expectation values. The expectation values which enter the definition of C_k are given by

$$\langle x_{i+k}x_i\rangle = \frac{1}{N-k}\sum_{i=1}^{N-k} x_i x_{i+k}.$$

The correlation function is related to the covariance.

Week 40, 1-5 October

Random Walks, Markov Chain and Metropolis algorithm

- Monday: Repetition from last week
- Central limit theorem
- Random walks and project 3
- Discussion of Markov chains and link to diffusion equation
- Wednesday:
- Entropy of a random walk
- Ergodic assumption and detailed balance
- Metropolis algorithm

The Lab is open during the midterm week (8-12 october). No lectures next week.

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.
- The transition probability W is also not known.
- How can we establish that we have reached a steady state? 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.
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. We establish this link first.
- 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. This is the topic for project 4.

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

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

We may be interested in the transition with one single spinflip to a new state j with energy E_j

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

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

$$j(x,t) = -D\frac{\partial w(x,t)}{\partial x}, \tag{38}$$

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

$$\frac{\partial j(x,t)}{\partial x} = -\frac{\partial w(x,t)}{\partial t}, \tag{39}$$

which leads to

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2}, \tag{40}$$

which is the diffusion equation in one dimension. Solved as a partial differential equation in chapter 15, project 5.

Diffusion from Markov Chain

With the probability distribution function $w(x,t)dx$ we can compute expectation values such as the mean distance

$$\langle x(t)\rangle = \int_{-\infty}^{\infty} xw(x,t)dx, \tag{41}$$

or

$$\langle x^2(t)\rangle = \int_{-\infty}^{\infty} x^2w(x,t)dx, \tag{42}$$

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

$$\langle f(x,t)\rangle = \int_{-\infty}^{\infty} f(x,t)w(x,t)dx. \tag{43}$$

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

$$\int_{-\infty}^{\infty} w(x,t)dx = 1 \tag{44}$$

imposes significant constraints on $w(x,t)$. These are

$$w(x=\pm\infty,t) = 0 \qquad \frac{\partial^n w(x,t)}{\partial x^n}|_{x=\pm\infty} = 0, \tag{45}$$

implying that when we study the time-derivative $\partial\langle x(t)\rangle/\partial t$, we obtain after integration by parts and using Eq. (40)

$$\frac{\partial\langle x\rangle}{\partial t} = \int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial t}dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 w(x,t)}{\partial x^2}dx, \tag{46}$$

leading to

$$\frac{\partial\langle x\rangle}{\partial t} = Dx\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - D\int_{-\infty}^{\infty}\frac{\partial w(x,t)}{\partial x}dx, \tag{47}$$

implying that

$$\frac{\partial\langle x\rangle}{\partial t} = 0. \tag{48}$$

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

$$\frac{\partial\langle x^2\rangle}{\partial t} = Dx^2\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - 2D\int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial x}dx, \tag{49}$$

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

$$\frac{\partial\langle x^2\rangle}{\partial t} = -Dxw(x,t)|_{x=\pm\infty} + 2D\int_{-\infty}^{\infty} w(x,t)dx = 2D, \tag{50}$$

leading to

$$\langle x^2\rangle = 2Dt, \tag{51}$$

and the variance as

$$\langle x^2\rangle - \langle x\rangle^2 = 2Dt. \tag{52}$$

The root mean square displacement after a time t is then

$$\sqrt{\langle x^2\rangle - \langle x\rangle^2} = \sqrt{2Dt}. \tag{53}$$

Random walks

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.

I project 3 this is what you will need to code.

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

$$D = \frac{l^2}{\Delta t}.$$

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

$$W_{ij}(\epsilon) = W(il-jl,\epsilon) = \left\{\begin{array}{cc}\frac{1}{2} & |i-j| = 1\\ 0 & \mathrm{else}\end{array}\right.$$

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

$$\sum_j W(j \rightarrow i) = 1.$$

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 $||\hat{\mathbf{w}}(t+1) - \hat{\mathbf{w}}(t)|| \rightarrow 0$, 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 $\hat{W}$

$$\hat{W} = \left(\begin{array}{ccc} 1/4 & 1/8 & 2/3 \\ 3/4 & 5/8 & 0 \\ 0 & 1/4 & 1/3 \end{array} \right),$$

and we choose our initial state as

$$\hat{w}(t=0) = \left(\begin{array}{c} 1 \\ 0 \\ 0 \end{array} \right).$$

The first iteration is

$$w_i(t=\epsilon) = W(j \rightarrow i)w_j(t=0),$$

resulting in

$$\hat{w}(t=\epsilon) = \left(\begin{array}{c} 1/4 \\ 3/4 \\ 0 \end{array} \right).$$

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

$$\hat{w}(t=2\epsilon) = \begin{pmatrix} 5/23 \\ 21/32 \\ 6/32 \end{pmatrix}.$$

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

$$\begin{aligned} W_{11}w_1(t=\infty)+W_{12}w_2(t=\infty)+W_{13}w_3(t=\infty) = & \quad w_1(t=\infty) \\ W_{21}w_1(t=\infty)+W_{22}w_2(t=\infty)+W_{23}w_3(t=\infty) = & \quad w_2(t=\infty) \\ W_{31}w_1(t=\infty)+W_{32}w_2(t=\infty)+W_{33}w_3(t=\infty) = & \quad w_3(t=\infty) \end{aligned} \tag{54}$$

with the constraint that

$$\sum_i w_i(t=\infty)=1,$$

yielding as solution

$$\hat{w}(t=\infty)=\left(\begin{array}{c} 4/15 \\ 8/15 \\ 3/15 \end{array}\right).$$

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 i^{th} 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 $t \rightarrow \infty$, $\hat{\mathbf{w}}(t)$ 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 tomorrow's lecture on 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. Form 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 **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 i^{th} 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 $t \rightarrow \infty$ 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 to be discussed tomorrow.

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.

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

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i,n+1)-w(i,n)}{\Delta t}, \tag{55}$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1,n)+w(i-1,n)-2w(i,n)}{(\Delta x)^2}, \tag{56}$$

resulting in the discretized diffusion equation

$$\frac{w(i,n+1)-w(i,n)}{\Delta t} = D\frac{w(i+1,n)+w(i-1,n)-2w(i,n)}{(\Delta x)^2}, \tag{57}$$

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 studied in the previous section, 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 indenpendent, 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

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \begin{cases} \frac{1}{2} & |i-j| = 1 \\ 0 & \text{else} \end{cases} \tag{58}$$

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

$$w_i(t = \epsilon) = W(j \rightarrow i) w_j(t = 0). \tag{59}$$

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

$$\sum_i w_i(t) = 1, \tag{60}$$

and

$$\sum_j W(j \to i) = 1. \tag{61}$$

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

$$w_i(t_n) = \sum_j W_{ij}(t_n)w_j(0), \tag{62}$$

and defining

$$W(il - jl, n\epsilon) = (W^n(\epsilon))_{ij} \tag{63}$$

we obtain

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0), \tag{64}$$

or in matrix form

$$w\hat{(}n\epsilon) = \hat{W}^n(\epsilon)\hat{w}(0). \tag{65}$$

Diffusion from Markov Chain

The matrix $\hat{W}$ can be written in terms of two matrices

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{66}$$

where $\hat{L}$ and $\hat{R}$ represent the transition probabilities for a jump to the left or the right, respectively. For a 4×4 case we could write these matrices as

$$\hat{R} = \left(\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right), \tag{67}$$

and

$$\hat{L} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right). \tag{68}$$

Diffusion from Markov Chain

However, in principle these are infinite dimensional matrices since the number of time steps are very large or infinite. For the infinite case we can write these matrices $R_{ij} = \delta_{i,(j+1)}$ and $L_{ij} = \delta_{(i+1),j}$, implying that

$$\hat{L}\hat{R} = \hat{R}\hat{L} = 1, \tag{69}$$

and

$$\hat{L} = \hat{R}^{-1} \tag{70}$$

To see that $\hat{L}\hat{R} = \hat{R}\hat{L} = 1$, perform e.g., the matrix multiplication

$$\hat{L}\hat{R} = \sum_k \hat{L}_{ik}\hat{R}_{kj} = \sum_k \delta_{(i+1),k}\delta_{k,(j+1)} = \delta_{i+1,j+1} = \delta_{i,j}, \tag{71}$$

and only the diagonal matrix elements are different from zero.

Diffusion from Markov Chain

For the first time step we have thus

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{72}$$

and using the properties in Eqs. (69) and (70) we have after two time steps

$$\hat{W}^2(2\epsilon) = \frac{1}{4}\left(\hat{L}^2 + \hat{R}^2 + 2\hat{R}\hat{L}\right), \tag{73}$$

and similarly after three time steps

$$\hat{W}^3(3\epsilon) = \frac{1}{8}\left(\hat{L}^3 + \hat{R}^3 + 3\hat{R}\hat{L}^2 + 3\hat{R}^2\hat{L}\right). \tag{74}$$

Diffusion from Markov Chain

Using the binomial formula

$$\sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{a}^k \hat{b}^{n-k} = (a+b)^n, \tag{75}$$

we have that the transition matrix after *n* time steps can be written as

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{R}^k \hat{L}^{n-k}, \tag{76}$$

or

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{L}^{n-2k} = \frac{1}{2^n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{R}^{2k-n}, \tag{77}$$

Diffusion from Markov Chain

and using $R^m_{ij} = \delta_{i,(j+m)}$ and $L^m_{ij} = \delta_{(i+m),j}$ we arrive at

$$W(il - jl, n\epsilon) = \begin{cases} \frac{1}{2^n} \begin{pmatrix} n \\ \frac{1}{2}(n+i-j) \end{pmatrix} & |i-j| \leq n \\ 0 & \text{else} \end{cases}, \tag{78}$$

and $n + i - j$ has to be an even number.

Diffusion from Markov Chain

We note that the transition matrix for a Markov process has three important properties:

- It depends only on the difference in space $i-j$, it is thus homogenous in space.
- It is also isotropic in space since it is unchanged when we go from (i,j) to $(-i,-j)$.
- It is homogenous in time since it depends only the difference between the initial time and final time.

If we place the walker at $x=0$ at $t=0$ we can represent the initial PDF with $w_i(0)=\delta_{i,0}$. Using Eq. (65) we have

$$w_i(n\epsilon)=\sum_j (W^n(\epsilon))_{ij}w_j(0)=\sum_j \frac{1}{2^n}\left(\begin{array}{c} n \\ \frac{1}{2}(n+i-j)\end{array}\right)\delta_{j,0}, \tag{79}$$

resulting in

$$w_i(n\epsilon)=\frac{1}{2^n}\left(\begin{array}{c} n \\ \frac{1}{2}(n+i)\end{array}\right) \qquad |i|\leq n \tag{80}$$

Diffusion from Markov Chain

Using the recursion relation for the binomials

$$\left(\begin{array}{c} n+1 \\ \frac{1}{2}(n+1+i)) \end{array} \right) = \left(\begin{array}{c} n \\ \frac{1}{2}(n+i+1) \end{array} \right) + \left(\begin{array}{c} n \\ \frac{1}{2}(n+i)-1 \end{array} \right) \tag{81}$$

we obtain, defining $x = il$, $t = n\epsilon$ and setting

$$w(x,t) = w(il, n\epsilon) = w_i(n\epsilon), \tag{82}$$

$$w(x, t+\epsilon) = \frac{1}{2}w(x+l, t) + \frac{1}{2}w(x-l, t), \tag{83}$$

and adding and subtracting $w(x,t)$ and multiplying both sides with l^2/ϵ we have

Diffusion from Markov Chain

$$\frac{w(x,t+\epsilon)-w(x,t)}{\epsilon} = \frac{l^2}{2\epsilon}\frac{w(x+l,t)-2w(x,t)+w(x-l,t)}{l^2}, \qquad (84)$$

and identifying $D = l^2/2\epsilon$ and letting $l = \Delta x$ and $\epsilon = \Delta t$ we see that this is nothing but the discretized version of the diffusion equation. Taking the limits $\Delta x \rightarrow 0$ and $\Delta t \rightarrow 0$ we recover

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2},$$

the diffusion equation.

Week 42, 15-19 October

Statistical Physics, Spins Systems and Phase Transitions

- Monday: Repetition, Markov chains, detailed balance, ergodicity and entropy
- Discussion of the Metropolis algorithm and parts of statistical physics
- How to simulate the Boltzmann probability
- Review of quantum mechanics
- Wednesday:
- Variational Monte Carlo and studies of quantum mechanical systems

Entropy and Equilibrium, section 9.4

The definition of the entropy S (as a dimensionless quantity here) is

$$S = -\sum_i w_i ln(w_i), \tag{85}$$

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. 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/chapter9/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;
}
      }
      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);
    }
  }
    cout << setiosflags(ios::showpoint | ios::uppercase);
    cout << setw(6) << time_steps;
    cout << setw(15) << setprecision(8) << entropy << endl;
}  // end of function output
```

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. 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

$$\sum_j W(j \to i) = 1$$

  Note that the probability for remaining at the same place is not necessarily equal zero.

- PDF w_i at time $t = n\epsilon$

$$w_i(t) = \sum_j W(j \to i)^n w_j(t = 0)$$

- $$\sum_i w_i(t) = 1$$

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 simply.

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

$$\frac{W(j \rightarrow i)}{W(i \rightarrow j)} = \frac{w_i}{w_j}$$

- 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

$$A(j \rightarrow i) = \begin{cases} \exp\left(-\beta(E_i - E_j)\right) & E_i - E_j > 0 \\ 1 & else \end{cases}$$

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

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{86}$$

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

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{87}$$

with mass $m = 1$.

Test of the Metropolis Algorithm

Want to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{88}$$

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

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{89}$$

with mass $m = 1$.

Test of the Metropolis Algorithm, Analytic 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);
}
```

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
```

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 FYS4410, 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 $A(\vec{r},\vec{p})$ which depends on position $\vec{r}$ and momentum $\vec{p}$ has a corresponding quantum mechanical operator by replacing $\vec{p}$ $-i\hbar\vec{\bigtriangledown}$, yielding the quantum mechanical operator

$$\widehat{\mathbf{A}} = A(\vec{r},-i\hbar\vec{\bigtriangledown}).$$

Quantity	Classical definition	QM operator
Position	$\vec{r}$	$\widehat{\tilde{\mathbf{r}}} = \vec{r}$
Momentum	$\vec{p}$	$\widehat{\tilde{\mathbf{p}}} = -i\hbar\vec{\bigtriangledown}$
Orbital momentum	$\vec{L} = \vec{r} \times \vec{p}$	$\widehat{\tilde{\mathbf{L}}} = \vec{r} \times (-i\hbar\vec{\bigtriangledown})$
Kinetic energy	$T = (\vec{p})^2/2m$	$\widehat{\mathbf{T}} = -(\hbar^2/2m)(\vec{\bigtriangledown})^2$
Total energy	$H = (p^2/2m) + V(\vec{r})$	$\widehat{\mathbf{H}} = -(\hbar^2/2m)(\vec{\bigtriangledown})^2 + V(\vec{r})$

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 $A(\vec{r},\vec{p})$ is given by

$$\langle A\rangle = \int (\Phi)^* \widehat{\mathbf{A}}(\vec{r}, -i\hbar\vec{\bigtriangledown})\Phi d\tau.$$

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

$$2^A \times \left(\begin{array}{c} A \\ Z \end{array} \right)$$

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.

Week 43, 22-26 October

Monte Carlo and Quantum Mechanics

- Monday:
- Repetition from last week and presentation of project 4.
- How to simulate the Hydrogen atom and the Helium atom
- Wednesday:
- Further discussion of variational Monte Carlo, quantum mechanical systems and project 4
- Introduction to statistical mechanics

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

$$\langle H \rangle = \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})H(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})},$$

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

$$\frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones.

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.

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 $\alpha = (\alpha_\mathbf{1}, \dots, \alpha_\mathbf{N})$.
- Then we evaluate the expectation value of the hamiltonian H

$$\langle H \rangle = \frac{\int d\mathbf{R}\Psi^*_{T_\alpha}(\mathbf{R})H(\mathbf{R})\Psi_{T_\alpha}(\mathbf{R})}{\int d\mathbf{R}\Psi^*_{T_\alpha}(\mathbf{R})\Psi_{T_\alpha}(\mathbf{R})}. \tag{90}$$

- Thereafter we vary α according to some minimization algorithm and return to the first step.

Quantum Monte Carlo

Choose a trial wave function $\psi_T(\mathbf{R})$.

$$P(\mathbf{R}) = \frac{\left|\psi_T(\mathbf{R})\right|^2}{\int \left|\psi_T(\mathbf{R})\right|^2 d\mathbf{R}}.$$

This is our new probability distribution function (PDF).

$$\langle E \rangle = \frac{\int d\mathbf{R}\Psi^*(\mathbf{R})H(\mathbf{R})\Psi(\mathbf{R})}{\int d\mathbf{R}\Psi^*(\mathbf{R})\Psi(\mathbf{R})},$$

where Ψ is the exact eigenfunction.

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H\psi_T(\mathbf{R}),$$

the local energy, which, together with our trial PDF yields

$$\langle H \rangle = \int P(\mathbf{R})E_L(\mathbf{R})d\mathbf{R}.$$

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.

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 hamiltionan 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

$$\langle H^n \rangle = \frac{\int d\mathbf{R}\Psi^{*}_T(\mathbf{R})H^n(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^{*}_T(\mathbf{R})\Psi_T(\mathbf{R})} = \mathrm{constant} \times \frac{\int d\mathbf{R}\Psi^{*}_T(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^{*}_T(\mathbf{R})\Psi_T(\mathbf{R})} = \mathrm{constant}.$$

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 $r_{12} \to 0$ we can write a possible trial wave function as

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}e^{r_{12}/2}.$$

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.

Week 44, 29 October-2 November

Quantum Monte Carlo, Ordinary Differential Equations

- Monday: Repetition from last week, Variational Monte Carlo
- Discussion of project 4
- Introduction to ordinary Differential Equations
- Wednesday:
- Ordinary Differential Equations, standard methods with emphasis on fourth order Runge-Kutta
- The Classical pendulum

Rest of semester, November, last five weeks

- Ordinary differential equations, Runge-Kutta method,chapter 13
- Ordinary differential equations with boundary conditions: one-variable equations to be solved by shooting method, chapter 14
- We can solve such equations by a finite difference scheme as well, turning the equation into an eigenvalue problem. Still one variable.
- Eigenvalue problems, chapter 12
- If we have more than one variable, we need to solve partial differential equations, which form the last part of this course. Chapter 15
- Fourier transforms and Fast Fourier transforms, chapter 15.

Project 5 starts november 12. Revised chapter 12-15 available from monday november 5.

Differential Equations

The order of the ODE refers to the order of the derivative on the left-hand side in the equation

$$\frac{dy}{dt} = f(t, y). \tag{91}$$

This equation is of first order and f is an arbitrary function. A second-order equation goes typically like

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{92}$$

A well-known second-order equation is Newton's second law

$$m\frac{d^2x}{dt^2} = -kx, \tag{93}$$

where k is the force constant. ODE depend only on one variable

Differential Equations

partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x},t), \qquad (94)$$

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 15. Involve boundary conditions in addition to initial conditions.

Differential Equations

We distinguish also between linear and non-linear differential equation where e.g.,

$$\frac{dy}{dt} = g^3(t)y(t), \tag{95}$$

is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{96}$$

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. (93). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{97}$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{98}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{99}$$

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

$$y_0 = y(t = t_0). \tag{100}$$

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

$$h = \frac{b-a}{N}. \tag{101}$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h), \tag{102}$$

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{103}$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{104}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{105}$$

Differential Equations

We define

$$y'(t_i) = f(t_i, y_i) \tag{106}$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{107}$$

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. (98) and (99). We define $y_0 = y^{(1)}(t=0)$ an $v_0 = y^{(2)}(t=0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{108}$$

and

$$y_1^{(2)} = v_0 - hy_0k/m + O(h^2). \tag{109}$$

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)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{110}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{111}$$

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

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \tag{112}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f \tag{113}$$

and we can rewrite Eq. (104) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f\right) + O(h^3), \tag{114}$$

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \dots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}). \quad (115)$$

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

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{116}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{117}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{118}$$

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 $O(h^2)$.

Differential Equations

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

$$y^{(2)}_{n+1/2} = y^{(2)}_{n-1/2} + ha_n + O(h^2), \tag{119}$$

and

$$y^{(1)}_{n+1} = y^{(1)}_n + hy^{(2)}_{n+1/2} + O(h^2). \tag{120}$$

Note that this method needs the calculation of $y^{(2)}_{1/2}$. This is done using e.g., Euler's method

$$y^{(2)}_{1/2} = y^{(2)}_0 + ha_0 + O(h^2). \tag{121}$$

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

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{122}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{123}$$

The program program2.cpp includes all of the above methods.

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

$$\frac{dy}{dt} = f(t, y), \tag{124}$$

and

$$y(t) = \int f(t, y)dt, \tag{125}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{126}$$

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

$$\int_{t_i}^{t_{i+1}} f(t,y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{127}$$

This means in turn that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{128}$$

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

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{129}$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i), \tag{130}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{131}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{132}$$

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

$$k_1 = hf(t_i, y_i), \tag{133}$$

$$k_2 = hf(t_i + h/2, y_i + k_1/2), \tag{134}$$

$$k_3 = hf(t_i + h/2, y_i + k_2/2) \tag{135}$$

$$k_4 = hf(t_i + h, y_i + k_3) \tag{136}$$

with the final value

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{137}$$

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

$$x(t) = A cos(\omega_0 t + \nu),$$

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
```

Week 45, 5 - 9 November

Ordinary Differential Equations (ODEs)

- Ordinary Differential Equations: driven nonlinear oscillations and the route to chaos. Discussion of the Pendulum and driven nonlinear oscillations.
- Ordinary Differential equations with boundary conditions
- Wednesday:
- Ordinary Differential equations with boundary conditions
- Begin partial differential equations

The classical pendulum

The angular equation of motion of the pendulum is given by Newton's equation and with no external force it reads

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{138}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{139}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{140}$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{141}$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \tag{142}$$

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.

Pendulum Code

1. Task: Write then a code which solves

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0,$$

using the fourth-order Runge Kutta method. Perform calculations for at least ten periods with $N = 100$, $N = 1000$ and $N = 10000$ mesh points and values of $\nu = 1$, $\nu = 5$ and $\nu = 10$. Set $l = 1.0$ m, $g = 1$ m/s^2 and $m = 1$ kg. Choose as initial conditions $\theta(0) = 0.2$ (radians) and $v(0) = 0$ (radians/s). Make plots of θ (in radians) as function of time and phase space plots of θ versus the velocity v. Check the stability of your results as functions of time and number of mesh points. Which case corresponds to damped, underdamped and overdamped oscillatory motion? Comment your results.

Classes for ODE methods

In this course we have not emphasized the use of classes. However for the ordinary differential equations 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 chapter13 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
```

More on the Pendulum

1. Next task: in

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t),$$

   we set set $l = g = 1$, $m = 1$, $\nu = 1/2$ and $\omega = 2/3$. Choose as initial conditions $\theta(0) = 0.2$ and $v(0) = 0$ and $A = 0.5$ and $A = 1.2$. Make plots of θ (in radians) as function of time for at least 300 periods and phase space plots of θ versus the velocity v. Choose an appropriate time step. Comment and explain the results for the different values of A.

This driving force will pump energy into the system and the externally imposed frequency ω will compete with the natural frequency $\omega_0 = \sqrt{g/l}$. There is an interesting situation when the driving frequency matches the naturalfrequency of the pendulum. This yields a resonance and the amplitude of θ can become very large especially if the friction is small.

More on the Pendulum

You will see that the first few oscillations are affected by the decay of an initial transient as is the case with no driving force. That is, the initial displacement of the pendulum leads to a component of the motion that decays with time and has an angular frequency $\sim \omega_0$. Afer this motion is damped away the pendulum settles in a steady motion drivenby ω.

For large A (F_D in the slides) the vertical jumps in θ are due to our resetting of the angle to keep it in the range $-\pi$ to π and thus corresponds to the pendulum swinging 'over the top'. For this value there is no settlement towards a steady behavior. Also, the behavior would be different with other initial conditions. Does not settle into a stable attractor in phase-space. Cannot predict the fate of the pendulum. At askance with the determinism of Newton's equations.

More on the Pendulum, Period Doubling

1. Next task: Keep now the constants from the previous exercise fixed but set now $A = 1.35$, $A = 1.44$ and $A = 1.465$. Plot θ (in radians) as function of time for at least 300 periods for these values of A and comment your results.

When a nonlinear system is excited or driven by a single frequency, the response is in general not limited to the driving frequency. We can have multiples $n\omega$ with $n = 1, 2, 3, \dots$. Well-known from Fourier's theorem and wave theory. This means that the periods of these harmonics will be smaller than the drive period ($T = 2\pi/\omega$). We see from the figure however that the middle figure with $F_D = 1.44$ exhibits a response $\omega/2$, a lower frequency! For the middle curve the bumps alternate in amplitude. We have a period which is twice as large as the $F_D = 1.35$ case. For $F_D = 1.465$ it is four times as large. If we increase F_D we will se further period doublings. Bifurcation plot is very useful. In the figure we waited 300 periods and plotted θ at times in phase with the driving period $2n\pi = T$ up to 400 periods. Below 1.44 there is only value although that value is plotted many times.Called period-1. At 1.44 it alternates between two values. Period-2. Then at 1.465 we have period-4.

More on the Pendulum

1. Final task: We want to analyse further these results by making phase space plots of θ versus the velocity v using only the points where we have $\omega t = 2n\pi$ where n is an integer. These are normally called the drive periods. This is an example of what is called a Poincare section and is a very useful way to plot and analyze the behavior of a dynamical system. Comment your results.

If you look at the non-chaotic case you will get only one point in the Poincare plot.

More on the Pendulum

Petit summary

1. Need a driving force and frequency
2. Strong dependence on the amplitude
3. In the chaotic regime we do not get a normal attractor in phase space. Cannot predict a path.
4. The path in the chaotic regime changes from initial condition to initial condition.

Can one extract universal behaviors for the chaotic pendulum? Period doubling is one route towards case. Another are the Lyapunov coefficients

$$\Delta\theta = e^{\lambda t}$$

Negative value is regular, $\lambda \geq 0$ is chaotic.

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

$$V(r)=-V_0f(r),$$

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

$$f(r) = \left\{ \begin{array}{c} 1 \\ -0 \end{array} \quad \text{for} \quad \begin{array}{c} r \leq a \\ r > a \end{array} \right.$$

and choose $\alpha = a$. Then

$$k(\rho) = \gamma \left\{ \begin{array}{c} 1 - \epsilon - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} \\ -\epsilon - -\frac{1}{\gamma}\frac{l(l+1)}{\rho^2} \end{array} \quad \text{for} \quad \begin{array}{c} r \leq a \\ r > a \end{array} \right.$$

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_{min} and E_{max}, the maximum number of iterations $\mathrm{max\_iter}$ and the desired numerical precision.
- Search then for the roots of the function $f(E)$, where the root(s) is(are) in the interval $E \in [E_{\mathrm{min}}, E_{\mathrm{max}}]$ 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

  $$f = u^{<}(\rho_m - h) - u^{>}(\rho_m + h) \leq \text{test}$$

  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, 12- 16 November

Partial Differential Equations (PDEs)

- Monday: Repetition from last week
- Parabolic PDEs: The diffusion equation, implicit and explicit finite difference schemes
- Discussion of project 5
- Wednesday:
- Discussion of project 5
- More on the diffusion equation
- Laplace's equation.
- The wave equation in one and two dimensions.

Partial Differential Equations

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

$$\frac{\kappa}{C\rho}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}$$

$$\frac{\kappa}{C\rho(\mathbf{x},t)}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}$$

Project 5

For this project you will have to build your own program. The project is more numerically directed, with tests of algorithms for solving partial differential equations using finite difference schemes.
The physical problem can be that of the temperature gradient in a rod of length $L = 1$ or that of channel flow between two flat and infinite plates at $x = 0$ and $x = 1$, where the fluid is initially at rest and the plate $x = 1$ is given a sudden initial movement. We are looking at a one-dimensional problem

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, t > 0, x \in [0, L]$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = 0 \qquad 0 < x < L$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = 0 \qquad t > 0 \qquad\qquad u(L,t) = 1 \qquad t > 0.$$

Project 5, Methods to study

1. The explicit forward Euler algorithm with discretized versions of time given by a forward formula and a centered difference in space resulting in
$$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j)}{\Delta t}$$
and
$$u_{xx} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2},$$
2. The implicit Backward Euler with
$$u_t \approx \frac{u(x,t)-u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j)-u(x_i,t_j-\Delta t)}{\Delta t}$$
and
$$u_{xx} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2},$$
3. Finally we use the implicit Crank-Nicolson scheme with a time-centered scheme at $(x, t+\Delta t/2)$
$$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j)}{\Delta t}.$$
The corresponding spatial second-order derivative reads
$$u_{xx} \approx \frac{1}{2}\left(\frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}+\right.$$

Project 5

a) Write down the algorithms for these three methods and the equations you need to implement. For the implicit schemes show that the equations lead to a tridiagonal matrix system for the new values.

b) Find the truncation errors of these three schemes and investigate their stability properties. Find also the analytic solution to the continuous problem.

c) Implement the three algorithms in the same code and perform tests of the solution for these three approaches for $\Delta x = 1/10$, $h = 1/100$ using Δt as dictated by the stability limit of the explicit scheme. Study the solutions at two time points t_1 and t_2 where $u(x, t_1)$ is smooth but still significantly curved and $u(x, t_2)$ is almost linear, close to the stationary state. Remember that forsolving the tridiagonal equations you can use results from project 1.

d) Compare the solutions at t_1 and t_2 with the analytic result for the continuous problem. Which of the schemes would you classify as the best?

Explicit Scheme for the Diffusion Equation

In one dimension we have thus the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t}, \tag{143}$$

or

$$u_{xx} = u_t, \tag{144}$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = g(x) \qquad 0 \leq x \leq L \tag{145}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = a(t) \qquad t \geq 0, \tag{146}$$

and

$$u(L,t) = b(t) \qquad t \geq 0, \tag{147}$$

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

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}, \tag{148}$$

and

$$u_{xx} \approx \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{149}$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{150}$$

Defining $\alpha = \Delta t / \Delta x^2$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha) u_{i,j} + \alpha u_{i+1,j}. \tag{151}$$

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

$$\Delta t/\Delta x^2 \leq 1/2 \tag{152}$$

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

$$A = \begin{pmatrix} 1+2\alpha & -\alpha & 0 & 0\dots \\ -\alpha & 1+2\alpha & -\alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & -\alpha & 1+2\alpha \end{pmatrix}$$

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.

$$u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t}$$

- 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

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & 0\dots \\ -1 & 2 & -1 & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & & 2 \end{pmatrix}$$

Tip for the analytic Solution of Project 5

Define a new function

$$v(x,t) = u(x,t) - x$$

and expand $v(x,t)$ as a Fourier Sine series

$$v(x,t) = \sum_k a_k(t) sin \pi kx$$

Week 47, 19- 23 November

Partial Differential Equations and Eigenvalue Problems

- Monday: Repetition from last week
- Discussion of project 5
- Laplace's equation and Poisson's equation
- The wave equation in one and two dimensions
- Wednesday:
- Eigenvalues: Jacobi's method and QR algorithm, Householder's

Laplace's and Poisson's equations

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{153}$$

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

$$u_{xx} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}, \tag{154}$$

and

$$u_{yy} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2}, \tag{155}$$

Laplace's and Poisson's equations

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{156}$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{157}$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \tag{158}$$

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

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \rho_{i,j}. \tag{159}$$

Solution Approach

The way we solve these equations is based on an iterative scheme called the relaxation method. Its 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. (158) or Eq. (159) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (158). 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.

Iterative Scheme

```
!  set up of initial conditions at t = 0 and boundary conditions
   ......
!  iteration algorithm starts here
       iterations = 0
       DO WHILE ( (iterations <= 20) .OR. ( diff > 0.00001) )
          u_temp = u; diff = 0.
          DO j = 2, ndim - 1
             DO l = 2, ndim -1
                u(j,l) = 0.25*(u_temp(j+1,l)+u_temp(j-1,l)+ &
                               u_temp(j,l+1)+u_temp(j,l-1))
                diff = diff + ABS(u_temp(i,j)-u(i,j))
             ENDDO
          ENDDO
          iterations = iterations + 1
          diff = diff/(ndim+1)**2
       ENDDO
!   write out results
       DO j = 1, ndim
          DO l = 1, ndim
             WRITE(6,*) j*h, l*h, u(j,l)
          ENDDO
       ENDDO
```

Laplace's Equation, Simple Case

A simple example may help in visualizing this method. We consider a condensator with parallel plates separated at a distance L resulting in e.g., the voltage differences $u(x,0) = 100sin(2\pi x/L)$ and $u(x,1) = -100sin(2\pi x/L)$. These are our boundary conditions and we ask what is the voltage u between the plates?

One-dimensional Wave Equation

The $1+1$-dimensional wave equation reads

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2}, \qquad (160)$$

with $u = u(x,t)$ and we have assumed that we operate with dimensionless variables. Possible boundary and initial conditions with $L = 1$ are

$$\left\{ \begin{array}{cc} u_{xx} = u_{tt} & x \in (0,1), t > 0 \\ u(x,0) = g(x) & x \in (0,1) \\ u(0,t) = u(1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x \in (0,1) \end{array} \right. . \qquad (161)$$

One-dimensional Wave Equation

We discretize again time and position,

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}, \quad (162)$$

and

$$u_{tt} \approx \frac{u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)}{\Delta t^2}, \quad (163)$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}, \quad (164)$$

and

$$u_{tt} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta t^2}, \quad (165)$$

resulting in

$$u_{i,j+1} = 2u_{i,j} - u_{i,j-1} + \frac{\Delta t^2}{\Delta x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right). \quad (166)$$

If we assume that all values at times $t=j$ and $t=j-1$ are known, the only unknown variable is $u_{i,j+1}$ and the last equation yields thus an explicit scheme for updating this quantity.

One-dimensional Wave Equation

The discretized version of this first derivative is given by

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t}, \tag{167}$$

and at $t = 0$ it reduces to

$$u_t \approx \frac{u_{i,+1} - u_{i,-1}}{2\Delta t} = 0, \tag{168}$$

implying that $u_{i,+1} = u_{i,-1}$. If we insert this condition in Eq. (166) we arrive at a special formula for the first time step

$$u_{i,1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right). \tag{169}$$

One-dimensional Wave Equation

We need seemingly two different equations, one for the first time step given by Eq. (169) and one for all other time-steps given by Eq. (166). However, it suffices to use Eq. (166) for all times as long as we provide $u(i,-1)$ using

$$u_{i,-1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right), \tag{170}$$

in our setup of the initial conditions.

Two-dimensional Wave Equation

$$\left\{\begin{array}{ll} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{array}\right. , \tag{171}$$

and we will let $\Delta x = \Delta y = h$ and $n_x = n_y$ for the sake of simplicity. The equation with initial and boundary conditions reads now

$$\left\{\begin{array}{cc} u_{xx} + u_{yy} = u_{tt} & x, y \in (0,1), t > 0 \\ u(x,y,0) = g(x,y) & x, y \in (0,1) \\ u(0,0,t) = u(1,1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x, y \in (0,1) \end{array}\right. . \tag{172}$$

Two-dimensional Wave Equation

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2}, \tag{173}$$

and

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}, \tag{174}$$

and

$$u_{tt} \approx \frac{u^{l+1}_{i,j} - 2u^l_{i,j} + u^{l-1}_{i,j}}{\Delta t^2}, \tag{175}$$

which we merge into the discretized $2+1$-dimensional wave equation as

$$u^{l+1}_{i,j} = 2u^l_{i,j} - u^{l-1}_{i,j} + \frac{\Delta t^2}{h^2}\left(u^l_{i+1,j} - 4u^l_{i,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right), \tag{176}$$

Two-dimensional Wave Equation

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

$$u_{i,j}^{-1} = u_{i,j}^{0} + \frac{\Delta t}{2h^2}\left(u_{i+1,j}^{0} - 4u_{i,0}^{0} + u_{i-1,j}^{0} + u_{i,j+1}^{0} + u_{i,j-1}^{0}\right), \tag{177}$$

in our setup of the initial conditions.

How do we deal with the spatial part? We need an algorithm for the Laplacian again.

Week 48, 26-30 November

Eigenvalue Problems and course summary

- Monday: Repetition from last week
- Eigenvalues: Jacobi's method and QR algorithm, Householder's algorithm.
- Wednesday:
- Course summary and exam discussion

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 $\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ then we have

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

and if we define the trace of $\mathbf{A}$ as

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

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

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 Évariste 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

$$\mathbf{D} = \left(\begin{array}{ccccccc} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n \end{array}\right).$$

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 **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 **A** and **x**. Then we get

$$(\mathbf{S}^{\mathbf{T}}\mathbf{A}\mathbf{S})(\mathbf{S}^{\mathbf{T}}\mathbf{x}) = \lambda\mathbf{S}^{\mathbf{T}}\mathbf{x}, \tag{178}$$

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 **B** as well, but with eigenvector $\mathbf{S}^{\mathbf{T}}\mathbf{x}$.

Eigenvalue Solvers

The basic philosophy is to

- either apply subsequent similarity transformations so that

$$\mathbf{S}_N^T \dots \mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 \dots \mathbf{S}_N = \mathbf{D}, \qquad (179)$$

- or apply subsequent similarity transformations so that **A** becomes tridiagonal. Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.

Eigenvalue Solvers, Jacobi

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

$$\mathbf{S} = \left(\begin{array}{cccccccc}
1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\
0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\
0 & 0 & \dots & \cos\theta & 0 & \dots & 0 & \sin\theta \\
0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\
0 & 0 & \dots & -sin\theta & 0 & \dots & 1 & \cos\theta \\
0 & 0 & \dots & 0 & \dots & \dots & 0 & 1
\end{array} \right)$$

with property $\mathbf{S^T} = \mathbf{S^{-1}}$. 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

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

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 **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

$$\left(\begin{array}{cc} b_{kk} & 0 \\ 0 & b_{ll} \end{array}\right) = \left(\begin{array}{cc} c & -s \\ s & c \end{array}\right) \left(\begin{array}{cc} a_{kk} & a_{kl} \\ a_{lk} & a_{ll} \end{array}\right) \left(\begin{array}{cc} c & s \\ -s & c \end{array}\right).$$

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

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

we obtain the quadratic equation

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

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

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

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.

Eigenvalue Solvers, Householder

The first step consists in finding an orthogonal matrix **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 **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,

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

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

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

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

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

and off-diagonal matrix elements

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

The resulting matrix reads

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

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

$$\mathbf{S_1} = \left(\begin{array}{cc} 1 & \mathbf{0^T} \\ \mathbf{0} & \mathbf{P} \end{array} \right),$$

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

$$P_{ij} = \delta_{ij} - 2u_i u_j,$$

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

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

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

$$\mathbf{P}\mathbf{v} = \mathbf{v} - 2\mathbf{u}(\mathbf{u}^T\mathbf{v}) = k\mathbf{e}, \tag{180}$$

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

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

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

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

Eigenvalue Solvers, Householder

Now we can rewrite Eq. (180) 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

$$2(\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), \tag{181}$$

which finally determines

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

In solving Eq. (181) 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

The matrix is now transformed into tridiagonal form and the last step is to transform it into a diagonal matrix giving the eigenvalues on the diagonal.
Before we discuss the algorithms, we note that the eigenvalues of a tridiagonal 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 a recursive 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

Special 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 $\mathbf{S_1}$

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

Eigenvalue Solvers, Householder

Then the similarity transformation

$$\mathbf{S_1^T A S_1} = \mathbf{A}' = \begin{pmatrix} d_1' & e_1' & 0 & 0 \\ e_1' & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e'3 \\ 0 & 0 & e_3' & d_4' \end{pmatrix}$$

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 $= a_{11}^{'} = d_1^{'}$.

Eigenvalue Solvers, Householder

This procedure can be continued on the remaining three-dimensional submatrix for the next eigenvalue. Thus after four transformations we have the wanted diagonal form.

Much more efficient than standard Jacobi.

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, Applications

Instead of solving the Schrödinger equation as a differential equation, we will solve it through diagonalization of a large matrix. However, in both cases we need to deal with a problem with boundary conditions, viz., the wave function goes to zero at the endpoints.
To solve the Schrödinger equation as a matrix diagonalization problem, let us study the radial part of the Schrödinger equation. The radial part of the wave function, $R(r)$, is a solution to

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

Eigenvalue Solvers, Applications

Then we substitute $R(r) = (1/r)u(r)$ and obtain

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

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

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

Code Example

```
//   Read in data
initialise(r_min, r_max, orb_l, max_step);
// initialise constants
step    = (r_max - r_min) / max_step;
const_2 = -1.0 / (step * step);
const_1 =  - 2.0 * const_2;
orb_factor = orb_l * (orb_l + 1);
```

Code Example

```
// local memory for r and the potential w[r]
r = new double[max_step + 1];
w = new double[max_step + 1];
for(i = 0; i <= max_step; i++) {
  r[i] = r_min + i * step;
  w[i] = potential(r[i]) + orb_factor / (r[i] * r[i]);
}
// local memory for the diagonalization process
d = new double[max_step];    // diagonal elements
e = new double[max_step];    // tridiagonal off-diagonal elements
z = (double **) matrix(max_step, max_step, sizeof(double));
```

Code Example

```
for(i = 0; i < max_step; i++) {
  d[i]    = const_1 + w[i + 1];
  e[i]    = const_2;
  z[i][i] = 1.0;
  for(j = i + 1; j < max_step; j++)  {
    z[i][j] = 0.0;
  }
}
// diagonalize and obtain eigenvalues
tqli(d, e, max_step - 1, z);
// Sort eigenvalues as an ascending series
qsort(d,(UL) max_step - 1,sizeof(double),
       (int(*)(const void *,const void *))comp);
// send results to ouput file
output(r_min , r_max, max_step, d);
```

Code Example

```
/*
  The function potential()
  calculates and return the value of the
  potential for a given argument x.
  The potential here is for the 1-dim harmonic oscillator
*/

double potential(double x)
{
   return x*x;  //  or coulomb   return 1/x;

} // End: function potential()
```

Exam FYS3150/4150, place FV329, Lab

Place and duration

- 10/12-14/12.
- duration: $\sim$ 45 min
- ca 20-25 min for discussion of the project, your presentation, reproduction of results, test runs etc
- 20-25 min for questions from one of the five topics listed below.
- Check final exam list by the end of this week, see webpage. Changes must be communicated to me at mhjensen@fys.uio.no.

Exam FYS3150/4150

How to prepare your Talk

- You can use slides, ps, pdf, ppt etc files (projector and slide projector at room FV329). Choose among projects 2 and 4 and 5.
- 10 mins (3-5 slides) for your presentation, rest of 10-15 mins questions and test of code from classfronter

You should discuss (briefly)

- The mathematical model and the physics
- Your algorithm and how you implemented it, with perhaps a selected calculation
- How you dealt with eventual comments and your corrections of these
- Any improvement you can think of
- And: we welcome your critiscism of the project.

Exam FYS3150/4150

Topics

Go to http://www.uio.no/studier/emner/matnat/fys/FYS3150/h07/, and click on syllabus.

- Linear algebra and eigenvalue problems. (Lecture notes chapters 4 and 12.1-12.3 and project 1), Blitz is not part of the curriculum.
- Numerical integration, standard methods and Monte Carlo methods (Lecture notes chapters 7 and 8, project 2), MPI is not part of the curriculum
- Monte Carlo methods in physics (Lecture notes chapters 9 and 11, projects 3-4)
- Ordinary differential equations (Lecture notes chapters 13 and 14)
- Partial differential equations (Lecture notes chapter 15, project 5)

Exam FYS3150/4150

Linear algebra and eigenvalue problems, chapters 4 and 12

- 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
- Cholesky and tridiagonal matrix decomposition (project 1)
- Householder's tridiagonalization technique and finding eigenvalues based on this
- Jacobi's method for finding eigenvalues

Exam FYS3150/4150

Numerical integration, standard methods and Monte Carlo methods (7 and 8)

- Trapezoidal, rectangle and Simpson's rules
- Gaussian quadrature, emphasis on Legendre polynomials, but you need to know about other polynomials as well. (project 2)
- Brute force Monte Carlo integration (project 2)
- Random numbers (simplest algo, ran0) and probability distribution functions, expectation values
- Improved Monte Carlo integration and importance sampling. (Project 2)

Exam FYS3150/4150

Monte Carlo methods in physics (9 and 11)

- Random walks and Markov chains and relation with diffusion equation (project 3)
- Metropolis algorithm, detailed balance and ergodicity
- Variational Monte Carlo (project 4)
- How to construct trial wave functions for quantum systems (project 4)

Exam FYS3150/4150

Ordinary differential equations (13 and 14)

- 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 14).

Exam FYS3150/4150

Partial differential equations

- Set up diffusion, Poisson and wave equations up to 2 spatial dimensions and time (project 5)
- 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. (project 5)
- How to compute the Laplacian in Poisson's equation.
- How to solve the wave equation in one and two dimensions.

Other courses in Computational Science at UiO

Bachelor/Master/PhD Courses

- INF-MAT3350/4350 Numerical linear algebra
- MAT-INF3300/3310/4300/4310, PDEs and Sobolev spaces I and II
- INF-MAT3360/4360 PDEs
- INF5620/5630/5640 Numerical methods for PDEs, finite element method
- MEK4550 Finite element in solid state mechanics
- FYS4410 Computational physics II (Parallelization (MPI), object orientation, classical statistical physics, simulation of phase transitions and quantum mechanical systems with many interacting particles)
- AST5340 Numerical Simulation, Methods in numerical simulation in hydrodynamics and plasma physics applied to astrophysical problems

Moralism...

. . . and it is feared that the French public, always impatient to come to a conclusion, eager to know the connections between general principles and the immediate questions that have aroused their passions, may be disheartened because they will be unable to move on at once.
That is a disadvantage I am powerless to overcome, unless it be by forewarning and forearming those readers who zealously seek the truth. There is no royal road to science, and only those who do not dread the fatiguing climb of its steep paths have a chance of gaining its luminous summits.
Karl Marx, preface to the french edition of Capital (Progress Publishers, Moscow, 1954).

AND, GOOD LUCK TO YOU ALL!