# Slides from FYS3150 Lectures

Morten Hjorth-Jensen

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

Fall 2008

- 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: 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 (to be determined later) and five selected topics, see the syllabus link on the webpage. Dates to be settled, most likely start, 11 or 12 December with end 18 or 19 december.

- The computer lab (room FV329)consists of 16 Linux PCs. C/C++ is the default programming language, but Fortran95 and Python are also used. All source codes discussed during the lectures can be found at the webpage of the course. We recommend either C/C++, Fortran95 or Python as languages.

| day | teacher |
| --- | --- |
| Thursday 9am-1 pm | Arnt Inge Vistnes |
| Thursday 1pm-5pm | Arnt Inge Vistnes |
| Friday 9am-1pm | Marius Lysebo |
| Friday 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.

## Place and duration

- Oral examination, typical start week 50 or week 51 (11 or 12 december to 18 or 19 december)
- 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.
- Please communicate your wishes for oral examination day as soon as possible.

## How to prepare your Talk

- You can use slides, ps, pdf, ppt etc files (projector and slide projector at room FV329). Choose among disegnated projects.
- 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.

# Topics covered in this course

- Numerical precision and intro to C++ programming

- Numerical derivation and integration

- Random numbers and Monte Carlo integration

- Monte Carlo methods in statistical physics

- Quantum Monte Carlo methods

- Linear algebra and eigenvalue problems

- Non-linear equations and roots of polynomials

- Ordinary differential equations

- Partial differential equations

- Parallelization of codes

### Linear algebra and eigenvalue problems, chapters 4 and 12

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

### Linear algebra and eigenvalue problems, chapters 4 and 12

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

# Syllabus FYS3150

## Numerical integration, standard methods and Monte Carlo methods (chapters 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.
- Brute force Monte Carlo integration
- Random numbers (simplest algo, ran0) and probability distribution functions, expectation values
- Improved Monte Carlo integration and importance sampling.

## Monte Carlo methods in physics (chapters 9, 10, and 11)

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

# Syllabus FYS3150

## Ordinary differential equations (chapters 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).

## Partial differential equations, chapter 15

- Set up diffusion, Poisson and wave equations up to 2 spatial dimensions and time

- Set up the mathematical model and algorithms for these equations, with boundary and initial conditions. Their stability conditions.

- Explicit, implicit and Crank-Nicolson schemes, and how to solve them. Remember that they result in triangular matrices.

- How to compute the Laplacian in Poisson's equation.

- How to solve the wave equation in one and two dimensions.

## Overarching aims of this course

- Develop a critical approach to all steps in a project, which methods are most relevant, which natural laws and physical processes are important. Sort out initial conditions and boundary conditions etc.

- This means to teach you structured scientific computing, learn to structure a project.

- A critical understanding of central mathematical algorithms and methods from numerical analysis. In particular their limits and stability criteria.

- Always try to find good checks of your codes (like solutions on closed form)

- To enable you to develop a critical view on the mathematical model and the physics.

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

# Selected Texts and lectures on C/C++

📕 J. J. Barton and L. R. Nackman, *Scientific and Engineering C++*, Addison Wesley, 3rd edition 2000.

📕 B. Stoustrup, *The C++ programming language*, Pearson, 1997.

📕 H. P. Langtangen INF-VERK3830
   `http://heim.ifi.uio.no/~hpl/INF-VERK4830/`

📕 D. Yang, *C++ and Object-oriented Numeric Computing for Scientists and Engineers*, Springer 2000.

📕 More books reviewed at `http::://www.accu.org/` and
   `http://www.comeaucomputing.com/booklist/`

# Other courses in Computational Science at UiO

## Bachelor/Master/PhD Courses

- INF-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

# 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.

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

# Hello World

## Dissection I

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

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

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

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

# Hello World

## Dissection II

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

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

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

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

## Hello World

### Now in C++

Here we present first the C++ version.

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

# C++ Hello World

### Dissection I

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

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

### 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.

## Integer and Real Numbers

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

# Limits

## C++ and Fortran declarations

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

# From decimal to binary representation

## How to do it

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

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

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

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=26 | remainder 0 | coefficient of $2^3$ is 0 |
| 26/2=13 | remainder 1 | coefficient of $2^4$ is 0 |
| 13/2= 6 | remainder 1 | coefficient of $2^5$ is 1 |
| 6/2= 3 | remainder 0 | coefficient of $2^6$ is 0 |
| 3/2= 1 | remainder 1 | coefficient of $2^7$ is 1 |
| 1/2= 0 | remainder 1 | coefficient of $2^8$ is 1 |

# From decimal to binary representation

## Integer numbers

```cpp
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

```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

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

## Loss of Precision

### Machine Numbers

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

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

and a real non-zero number could be generalized as

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

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

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

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

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

# Loss of Precision

## Machine Numbers

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

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

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

### 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 explicitly that a binary number with 23 bits for the mantissa reads

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

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

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

# Loss of Precision

## Machine Numbers

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

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

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

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

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

# Loss of Precision

## Machine Numbers

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

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

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

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

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

# Loss of Precision

## Machine Numbers

In the machine a number is represented as

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

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

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

or

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

meaning that

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

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

# Loss of Precision

## Machine Numbers

Define the absolute error as

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

whereas the relative error is

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

The above subraction is thus

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

yielding

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

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

## Loss of numerical precision

Suppose we wish to evaluate the function

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

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

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

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

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

and

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

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

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

while the second expression results in

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

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

# Loss of numerical precision

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

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

# Loss of Precision, bad thing

### Real Numbers

- **Overflow :** When the positive exponent exceeds the max value, e.g., 308 for DOUBLE PRECISION (64 bits). Under such circumstances the program will terminate and some compilers may give you the warning 'OVERFLOW'.

- **Underflow :** When the negative exponent becomes smaller than the min value, e.g., -308 for DOUBLE PRECISION. Normally, the variable is then set to zero and the program continues. Other compilers (or compiler options) may warn you with the 'UNDERFLOW' message and the program terminates.

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

  $$x = 1.234567891112131468 = 0.1234567891112131468 \times 10^1 \qquad (15)$$

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

# More on Loss of Precision

## Real Numbers

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

# A problematic Case

## Three ways of computing $e^{-x}$

**1** Brute force

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

**2** recursion relation for

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

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

**3**

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

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

# Program to compute exp $(-x)$

### Brute Force

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

# Program to compute exp $(-x)$

## Still Brute Force

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

# Program to compute exp $(-x)$

## Oh, it never ends!

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

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

# Results $\exp(-x)$

## What is going on?

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

## Program to compute $\exp(-x)$

```cpp
// 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 |

- 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.

Message: Revised lecture notes (chapters 2-15) will be put on webpage september 1 2008.

# 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

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

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

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

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

## Error Analysis

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

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

we reach fairly quickly a limit for where loss of precision due to the subtraction of two nearly equal numbers becomes crucial.

If $(f_{\pm h} - f_0)$ are very close, we have $(f_{\pm h} - f_0) \approx \epsilon_M$, where $|\epsilon_M| \leq 10^{-7}$ for single and $|\epsilon_M| \leq 10^{-15}$ for double precision, respectively.

We have then

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

# Error Analysis

Our total error becomes

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

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

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

With double precision and $x = 10$ we obtain

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

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

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 | $0.0000000000000000 \times 10^{0}$ |

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

# Pointer example I

## Discussion

| Line | Comments |
|------|----------|
| 4 | • Defines an integer variable var. |
| 5 | • Define an integer pointer – reserves space in memory. |
| 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

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

# Pointer example II

## Discussion

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

# Pointer example II

## Discussion

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

```
Address of the matrix element matr[1]: 0xbffffef70
Value of the  matrix element  matr[1]; 321
Address of the matrix element matr[2]: 0xbffffef74
Value of the matrix element  matr[2]: 322
Value of the pointer: 0xbffffef70
The value pointer points at: 321
The value that (pointer+1) points at:  322
Address of the pointer variable : 0xbffffef6c
```

```
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);
}
```

```
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

```cpp
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
```

```cpp
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
```

```
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;
```

```
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()
```

```
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().

- 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.

- 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().

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.

## Linear Algebra

- Monday: Repetition from last week
- Discussion of Project 1, deadline 15 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
- 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.

# 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

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

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 **A** the following properties are all equivalent

1. If the inverse of **A** exists, **A** is nonsingular.

2. The equation $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.

3. The rows of **A** form a basis of $R^N$.

4. The columns of **A** form a basis of $R^N$.

5. **A** is a product of elementary matrices.

6. 0 is not eigenvalue of **A**.

# Important Mathematical Operations

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

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

scalar-matrix multiplication

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

vector-matrix multiplication

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

matrix-matrix multiplication

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

and transposition

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

# Important Mathematical Operations

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

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

scalar-vector multiplication

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

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{yz} \implies x_i = y_i z_i, \tag{23}$$

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

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

and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{yz}^T \implies a_{ij} = y_i z_j, \tag{25}$$

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

## Static

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

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

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

# Matrix Handling in C/C++

## Row Major Order, Addition

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

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

In C/C++ this would be coded like

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

# Matrix Handling in C/C++

## Row Major Order, Multiplication

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

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

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

```fortran
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

```fortran
a=b+c
```

Multiplication

```fortran
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;
```

# Gaussian Elimination

We start with the linear set of equations

$$\mathbf{Ax} = \mathbf{w}.$$

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

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

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

# Gaussian Elimination

Our actual $4 \times 4$ example reads after the first operation

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

or

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

$$(27)$$

## Gaussian Elimination

The new coefficients are

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

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

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

with a new right-hand side given by

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

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

# Gaussian Elimination

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

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

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

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

This set of $n-1$ eliminations 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 $10^7$.

Suppose we want to solve the following boundary value equation

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

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

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

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

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

which we rewrite as

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

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

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

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

$$\mathbf{A} = \frac{1}{h^2} \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \cdots & \cdots & \cdots & \cdots & \cdots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}$$

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

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

We start with the linear set of equations

$$\mathbf{Au} = \mathbf{f},$$

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

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

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

We can rewrite as

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

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

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

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

# Gaussian Elimination and Tridiagonal matrices, project 1

In case your system of equations leads to a tridiagonal matrix, it is clearly an overkill to employ Gaussian elimination or the standard LU decomposition. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter 15.

Our algorithm starts with forward substitution with a loop over of the elements *i* and can be expressed via the following piece of code

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

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

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

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

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

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

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

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

and finally

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

# Project 1, hints

We notice the sub-blocks which get repeated

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

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

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

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

# LU Decomposition

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

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

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

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

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

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

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

# Linear Algebra Methods

- Gaussian elimination, $O(2/3n^3)$ flops, general matrix

- LU decomposition, upper triangular and lower tridiagonal matrices, $O(2/3n^3)$ flops, general matrix. Get easily the inverse, determinant and can solve linear equations with back-substitution only, $O(n^2)$ flops

- Cholesky decomposition $A = LL^T$. Real symmetric or hermitian positive definite matrix, $O(1/3n^3)$ flops.

- Tridiagonal linear systems, important for differential equations. Normally positive definite and non-singular. $O(8n)$ flops for symmetric. $A = LDL^T$ with $D$ the diagonal. Special case of banded matrices.

- Singular value decomposition

- the QR method will be discussed in chapter 12 in connection with eigenvalue systems. $O(4/3n^3)$ flops.

# How to use the Library functions

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

```
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]);
    }
  }
```

```
// 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:

```fortran
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) )
```

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.

As a result, untill very recently, the adoption of C++ for scientific computing has been slow. This has changed quite a lot in the last years and modern C++ compilers with numerical libraries have improved the situation considerably. Recent benchmarks show C++ encroaching steadily on Fortran's high-performance monopoly, and for some benchmarks, C++ is even faster than Fortran! These results are being obtained not through better optimizing compilers, preprocessors, or language extensions, but through the use of template techniques. By using templates cleverly, optimizations such as loop fusion, unrolling, tiling, and algorithm specialization can be performed automatically at compile time.

The features of Blitz++ which are useful for our studies are the dynamical allocation of vectors and matrices and algebraic operations on these objects. In particular, if you access the Blitz++ webpage at `http://www.oonumerics.org/blitz/`, we recommend that you study chapters two and three.

A simple makefile

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

blitz_test:             blitz_test.o


clean:
                rm -f *.o
```

## Blitz++ example

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

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

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

```
// 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);
```

```
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   ]
```

## Linear Algebra and differential equations

- Monday: Repetition from last week
- Determinant and inverse of matrices, chapter 4.
- Qubic spline, chapter 6
- Singular value decomposition, chapter 4
- Wednesday: Differential equations
- Improved Euler methods and introduction to differential equations
- Presentation of project 2.
- Computer-Lab: thursday and friday 9am-7pm, Project 1.

- Ordinary differential equations, Runge-Kutta method,chapter 13
- Ordinary differential equations with boundary conditions: one-variable equations to be solved by shooting and Green's function methods, 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.

Till middle of october. Projects 2-4 cover these topics.

Qubic spline interpolation is among one of the mostly used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called qubic spline method to be described below.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.

Assume we have at our disposal $n + 1$ points $x_0, x_1, \ldots x_n$ arranged so that $x_0 < x_1 < x_2 < \ldots x_{n-1} < x_n$ (such points are called knots). A spline function $s$ of degree $k$ with $n + 1$ knots is defined as follows

- On every subinterval $[x_{i-1}, x_i]$ $s$ is a polynomial of degree $\leq k$.

- $s$ has $k - 1$ continuous derivatives in the whole interval $[x_0, x_n]$.

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

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

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

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

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

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

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

yielding $4n$ coefficients to determine.

Every subinterval provides in addition the $2n$ conditions

$$y_i = s(x_i),$$

and

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

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

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

yields $n - 1$ conditions. Similarly,

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

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

## Splines

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

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

and

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

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

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

and integrating twice one obtains

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

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

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

## Splines

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

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

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

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

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

Gaussian elimination

$$
\left[
\begin{array}{ccccccc}
u_1 & h_1 & 0 & \ldots & & & \\
h_1 & u_2 & h_2 & 0 & \ldots & & \\
0 & h_2 & u_3 & h_3 & 0 & \ldots & \\
\ldots & & \ldots & \ldots & \ldots & \ldots & \ldots \\
& \ldots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\
& & & & & 0 & h_{n-2} & u_{n-1}
\end{array}
\right]
\left[
\begin{array}{c}
f_1 \\
f_2 \\
f_3 \\
\ldots \\
f_{n-2} \\
f_{n-1}
\end{array}
\right]
=
\left[
\begin{array}{c}
v_1 \\
v_2 \\
v_3 \\
\ldots \\
v_{n-2} \\
v_{n-1}
\end{array}
\right].
$$

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

## Splines

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

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

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

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

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

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

## Differential Equations

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

$$\frac{dy}{dt} = f(t, y). \tag{34}$$

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

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{35}$$

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

$$m\frac{d^2x}{dt^2} = -kx, \tag{36}$$

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

## Differential Equations

partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x},t), \qquad (37)$$

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.

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

$$\frac{dy}{dt} = g^3(t)y(t), \tag{38}$$

is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{39}$$

is a non-linear ODE.

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 $\rho$

$$\frac{dm}{dr} = 4\pi r^2 \rho(r)/c^2,$$

and

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

where $\rho$ 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. (36). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{40}$$

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

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{41}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{42}$$

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

$$y_0 = y(t = t_0). \tag{43}$$

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

$$h = \frac{b - a}{N}. \tag{44}$$

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

$$y_1 = y(t_1 = t_0 + h), \tag{45}$$

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 $y_i$

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{46}$$

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{47}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{48}$$

# Differential Equations

We define

$$y'(t_i) = f(t_i, y_i) \tag{49}$$

and if we truncate $\Delta$ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{50}$$

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. (41) and (42). We define $y_0 = y^{(1)}(t=0)$ an $v_0 = y^{(2)}(t=0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{51}$$

and

$$y_1^{(2)} = v_0 - hy_0 k/m + O(h^2). \tag{52}$$

# 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)

$$y_{n+1}^{(1)} = y_n^{(1)} + h y_{n+1}^{(2)} + O(h^2) \tag{53}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + h a_n + O(h^2). \tag{54}$$

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

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2} \frac{df(t_i, y_i)}{dt} + O(h^3). \tag{55}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f \tag{56}$$

and we can rewrite Eq. (47) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) + O(h^3), \tag{57}$$

which has a local approximation error $O(h^3)$ and a global error $O(h^2)$.

# Differential Equations

These approximations can be generalized by using the derivative $f$ to arbitrary order so that we have

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \ldots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}). \quad (58)$$

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.

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

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{59}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{60}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{61}$$

implying that the local truncation error in the position is now $O(h^3)$, whereas Euler's or Euler-Cromer's methods have a local error of $O(h^2)$.

Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.

One method that avoids this is the so-called half-step method. Here we define

$$y_{n+1/2}^{(2)} = y_{n-1/2}^{(2)} + ha_n + O(h^2), \tag{62}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{63}$$

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

$$y_{1/2}^{(2)} = y_0^{(2)} + ha_0 + O(h^2). \tag{64}$$

As this method is numerically stable, it is often used instead of Euler's method.

Another method which one may encounter is the Euler-Richardson method with

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{65}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{66}$$

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

### Ordinary Differential Equations (ODEs)

- Monday
- Ordinary Differential equations and Runge-Kutta methods, one-step methods and adaptive methods
- Discussion of project 2
- Wednesday:
- Ordinary Differential Equations: Pendulum, code, classes, driven nonlinear oscillations and the route to chaos.

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

$$\frac{dy}{dt} = f(t, y), \tag{67}$$

and

$$y(t) = \int f(t, y) dt, \tag{68}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y) dt. \tag{69}$$

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

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx h f(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{70}$$

This means in turn that we have

$$y_{i+1} = y_i + h f(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{71}$$

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

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{72}$$

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

$$k_1 = hf(t_i, y_i), \tag{73}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{74}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{75}$$

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

$$k_1 = hf(t_i, y_i), \tag{76}$$

$$k_2 = hf(t_i + h/2, y_i + k_1/2), \tag{77}$$

$$k_3 = hf(t_i + h/2, y_i + k_2/2) \tag{78}$$

$$k_4 = hf(t_i + h, y_i + k_3) \tag{79}$$

with the final value

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{80}$$

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 $O(h^4)$.

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 $\nu$ 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.

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 $2\pi$, 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.

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

$$E_0 = \frac{1}{2} k x(t = 0)^2 = \frac{1}{2} k.$$

The total energy at any time $t$ has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0 = \frac{1}{2} k x(t)^2 + \frac{1}{2} m v(t)^2.$$

# 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

$$E_0 = \frac{1}{2} k x(t = 0)^2 = \frac{1}{2} k.$$

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;
```

```
   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
```

## Ordinary Differential Equations (ODEs) with boundary conditions

- Monday
- Discussion of project 2
- Ordinary Differential Equations: Pendulum, code, classes, driven nonlinear oscillations and the route to chaos.
- Begin of discussion of ODE with two-point boundary conditions
- Wednesday:
- ODE with two-point boundary conditions, Numerov's method, shooting and matching.
- Green's function methods

- There were two errors in the old project description (till last sunday).

$$R_0 = \left(\frac{Y_e m_e c^2}{4\pi \rho_0 G M_p}\right)^{1/2} = 7.72 \times 10^6 \, Y_e \quad \text{m}.$$

and $R_\odot = 6.95 \times 10^8$ m. The power $10^6$ was wrongly written $10^5$ and the radius of the sun carried an exponent of $10^7$ instead of $10^8$.

- In the text we write

$$x \propto n^{(1/3)} \propto V^{-3},$$

it should be

$$x \propto n^{(1/3)} \propto V^{-1/3},$$

# Why do we think electrons are responsible in white dwarfs?

The arguments here follow Landau's original work back in 1932 related to the proposition of neutron stars (after Chadwick's discovery of the neutron).

Assume we have $N$ fermions inside a star of radius $R$. We have a density of fermions $n = N/V$.

Using Heisenberg's uncertainty relation we can assume that momentum is proportional

$$p \sim \hbar n^{1/3}$$

The gravitional energy per fermion is then

$$E_G \sim -\frac{GMm_B}{R},$$

with $m_B$ the mass of the relevant baryons (neutrons and protons). The total mass is assumed to be $M = Nm_B$.

The fermions are relativistic (why is explained below) and we assume that the energy is

$$E_F \sim \hbar c n^{1/3} \sim \frac{\hbar c N^{1/3}}{R},$$

recall $\sqrt{p^2 c^2 + m^2 c^4}$.

# Why do we think electrons are responsible in white dwarfs?

The total energy is

$$E = E_G + E_F \sim \frac{\hbar c N^{1/3}}{R} - \frac{GNm_B^2}{R}.$$

It depends on $R$ and $N$. Rest is given by fundamental constants. Stability when $E = 0$.

- Assume $E > 0$
    1. This means $N$ is small for fixed $R$.
    2. Reduce $E$ by increasing $R$
    3. Electrons become non-relativistic $E_F \propto 1/R^2$ and as $R \to \infty$, $E$ becomes negative, collapse. Only stable for finite $R$.

- Assume $E < 0$
    1. Can reduce $E$ with no bounds by decreasing $R$, that is $R \to 0$.
    2. No stable star.

# Why do we think electrons are responsible in white dwarfs?

The total energy is

$$E = E_G + E_F \sim \frac{\hbar c N^{1/3}}{R} - \frac{G N m_B^2}{R}.$$

Stability when $E = 0$.
Maximum baryon number when $E = 0$,

$$N_{\max} \sim \left( \frac{\hbar c}{G m_B^2} \right)^{3/2} \sim 2 \times 10^{57}.$$

Maximum mass of the star

$$M_{\max} \sim N_{\max} m_B \sim 1.5 \times M_\odot$$

This analysis involves only fundamental constants. The radius at equilibrium when $M \to M_{\max}$ is connected with a relativistic equation of state, that is $E_F \geq mc^2$ ($E = 0$) resulting in a radius

$$R \leq \frac{\hbar}{mc} \left( \frac{\hbar c}{G m_B^2} \right)^{1/2}.$$

# Why do we think electrons are responsible in white dwarfs?

Inserting the mass of the neutron or the mass of the electron in

$$R \leq \frac{\hbar}{mc} \left( \frac{\hbar c}{G m_B^2} \right)^{1/2}.$$

we obtain radii for either a neutron star or a white dwarf. With $m = m_n$ we get $R \sim 3 \times 10^3$ m, or just some few kilometers. Alternatively, with $m = m_e$ we get $R \sim 5 \times 10^6$ m, or some thousand kilometers. We have only use fundamental constants in this analysis.

Radius of white dwarf as function of central density.
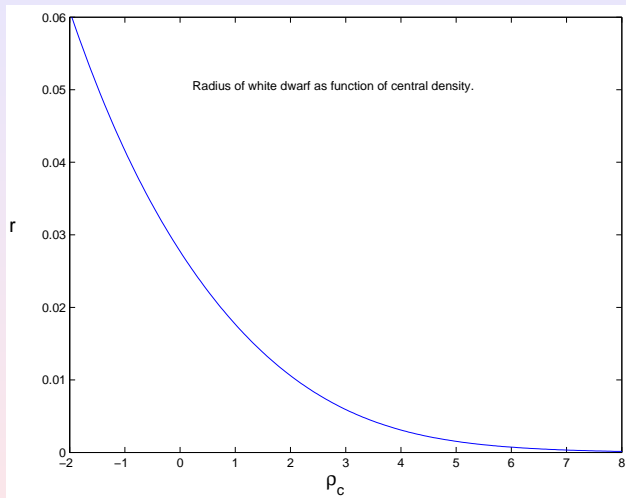
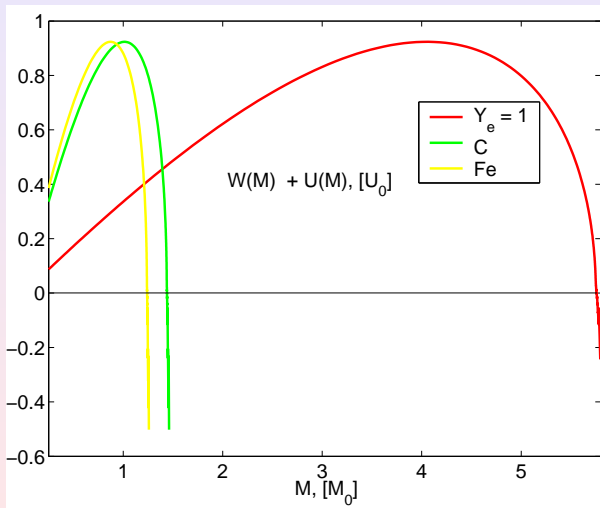# Mass-radius relationships for different values of $Y_e$
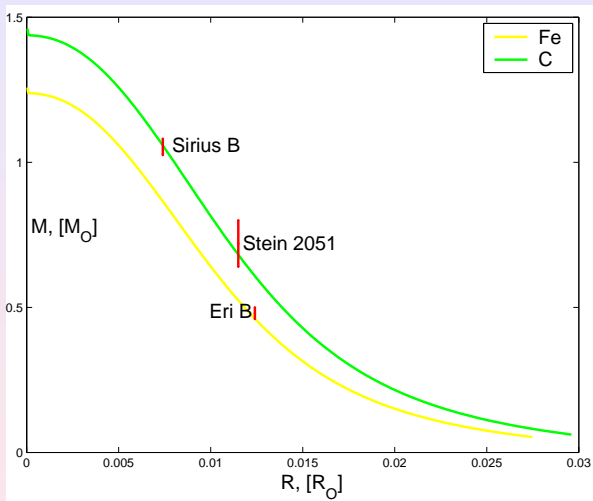
As an interesting digression, we compute the total kinetic and rest energy of the electrons $U(R)$ and the gravitational energy $W(R)$ given by

$$U(R) = \frac{3}{8} 4\pi n_0 m_e c^2 \int_0^R (x(1+2x^2)\sqrt{1+x^2} - ln(x+\sqrt{1+x^2}))\, r^2 dr,$$

where $x = (\bar{\rho}(\bar{r}))^{\frac{1}{3}}$ and

$$W(R) = -4\pi G \int_0^R m(r)\, \rho(r) r dr.$$

# Gravitational and kinetic energies for different values of $Y_e$

The mass value at which sum of $U + W$ is equal to zero corresponds to the maximum mass a white dwarf of a given composition can assume. At this point the degenerate electron gas pressure cannot balance the gravitational pull of a star's mass. This leads eventually to the star's collapse. One easily notices that the maximum mass of a white dwarf composed primarily of [12] is roughly $M \approx 1.44 M_{\odot}$, which is equal to the famous Chandrasekhar's limit.

# The classical pendulum

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

$$ml\frac{d^2\theta}{dt^2} + mg\sin(\theta) = 0, \tag{81}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{82}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{83}$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{84}$$

where $\nu$ 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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \tag{85}$$

with $A$ and $\omega$ two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

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.

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
 };
```

```
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

```fortran
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
```

Suppose we want to solve the following boundary value equation

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

with $x \in (a, b)$ and with for example boundary conditions $u(a) = u(b) = 0$. We assume that $f$ is a continuous function in the domain $x \in (a, b)$.

We have already seen this in project 1 ($g(x) = 0$).

Another example of an equation with similar boundary conditions is

$$\frac{d^2 u(x)}{dx^2} + g(x)u(x) = \lambda u(x),$$

which is often called an eigenvalue equation.

How do we solve such equations?

## Ordinary Differential Equations (ODEs) with boundary conditions and Eigenvalue problems

- Monday
- Discussion of project 3
- ODE with two-point boundary conditions, shooting and matching method.
- Differential equations as eigenvalue problems, Jacobi's method
- Wednesday:
- Eigenvalue problems, Jacobi's method and Householder's method

# Boundary Value Problems

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

In our case we are interested in attractive potentials

$$V(r) = -V_0 f(r),$$

where $V_0 > 0$ and analyze bound states where $E < 0$.

## Boundary Value Problems

The final equation can be written as

$$\frac{d^2}{d\rho^2} u(\rho) + k(\rho) u(\rho) = 0,$$

where

$$
\begin{aligned}
k(\rho) &= \gamma \left( f(\rho) - \frac{1}{\gamma} \frac{l(l+1)}{\rho^2} - \epsilon \right) \\
\gamma &= \frac{2m\alpha^2 V_0}{\hbar^2} \\
\epsilon &= \frac{|E|}{V_0}
\end{aligned}
$$

# Boundary Value Problems

$$f(r) = \begin{cases} 1 & \text{for} \quad r \leq a \\ 0 & \quad r > a \end{cases}$$

and choose $\alpha = a$. Then

$$k(\rho) = \gamma \begin{cases} 1 - \epsilon - \frac{1}{\gamma} \frac{l(l+1)}{\rho^2} & \text{for} \quad r \leq a \\ -\epsilon - -\frac{1}{\gamma} \frac{l(l+1)}{\rho^2} & \quad r > a \end{cases}$$

For small $\rho$ 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 $\rho$ 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 $\rho$ 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 $\rho$, usually zero, and up to some large value of $\rho$. 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 $u(\rho) = e^{+\gamma \epsilon \rho}$.

The final solution will then become unstable. Therefore, it is better to generate two solutions, with one starting from small values of $\rho$ 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 $\rho$ where the potential is of no importance, and inwards to the same matching point $\rho_m$. Due to the quantum mechanical requirements the logarithmic derivative at the matching point $\rho_m$ should be well defined.

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.

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 $E \in [E_{\min}, E_{\max}]$ using e.g., the bisection method.

```
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];
```

```
// 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]);
```

## 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 $\lambda$ and in general we will have $n$ distinct zeros.

## Eigenvalue Solvers

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

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

or

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

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

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

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

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

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

# Abel-Ruffini Impossibility Theorem

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

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

# Abel-Ruffini Impossibility Theorem

The Abel-Ruffini theorem says that there are some fifth-degree equations whose solution cannot be so expressed. The equation $x^5 - x + 1 = 0$ is an example. Some other fifth degree equations can be solved by radicals, for example $x^5 - x^4 - x + 1 = 0$. The precise criterion that distinguishes between those equations that can be solved by radicals and those that cannot was given by 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 $\lambda_1 \ldots \lambda_n$ (distinct or not). Let $\mathbf{D}$ be the diagonal matrix with the eigenvalues on the diagonal

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

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

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

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

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.

To prove this we start with the eigenvalue problem and a similarity transformed matrix **B**.

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

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

$$(\mathbf{S^TAS})(\mathbf{S^Tx}) = \lambda\mathbf{S^Tx}, \tag{86}$$

which is the same as

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

The variable $\lambda$ is an eigenvalue of **B** as well, but with eigenvector $\mathbf{S^Tx}$.

# Eigenvalue Solvers

The basic philosophy is to

- either apply subsequent similarity transformations so that

$$\mathbf{S_N^T} \ldots \mathbf{S_1^T A S_1} \ldots \mathbf{S_N} = \mathbf{D}, \tag{87}$$

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

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

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

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

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

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

A similarity transformation

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

results in

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

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

The main idea is thus to reduce systematically the norm of the off-diagonal matrix elements of a matrix **A**

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

To demonstrate the algorithm, we consider the simple $2 \times 2$ 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 $2 \times 2$ case we have

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

which leads to

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

since

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

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

# Eigenvalue Solvers, Jacobi

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

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

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

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

resulting in

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

and $c$ and $s$ are easily obtained via

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

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

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

- Choose a tolerance $\epsilon$, 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

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

- 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 $\text{off}(\mathbf{B}) \leq \epsilon$

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.

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

$$\mathbf{B} = \left( \begin{array}{ccc} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{array} \right) \left( \begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array} \right) \left( \begin{array}{ccc} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{array} \right).$$

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

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

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

We have

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

or

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

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

We get then a new matrix

$$\mathbf{B} = \left( \begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & 0 \\ b_{13} & 0 & a_{33} \end{array} \right).$$

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

$$\mathbf{C} = \left( \begin{array}{ccc} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{array} \right) \left( \begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & 0 \\ b_{13} & 0 & b_{33} \end{array} \right) \left( \begin{array}{ccc} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{array} \right).$$

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

The more general expression for the new matrix elements are

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

This is what we will need to code.

## Jacobi code example

### Main part

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

## Jacobi code example

Finding the max nondiagonal element

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

## Jacobi code example

Finding the new matrix elements

```
//
void    jacobirotate(double **A, double **R, int p, int q, int n)
{
  ....
  double c = 1.0; double s = 0.0;
  if ( A[p][q] /= 0.0 ) {
  double tau = (A[p][p]-A[q][q])/(2.0*A[p][q];
  if ( tau >= 0.0) { tan = 1/(tau +sqrt(1+tau*tau));}
  if ( tau < 0.0) { tan = -1/(-tau +sqrt(1+tau*tau));}
  double c = 1.0/sqrt(1.0+tan*tan);   double s = tan*c;

  A[p][p] += tan*A[p][q];
  A[q][q] -= tan*A[p][q];
  for ( int k = 0; k < n; ++k)
  {
    if ( k != p && k != q) {
       A[k][p] = c*A[k][p]-s*A[k][q];
       A[k][q] = c*A[k][q]+s*A[k][p];
    }
  }
.....
}
...
```

## Partial Differential Equations (PDEs)

- Monday: Repetition from last week and discussion of project 3
- Householder's algorithm
- Parabolic PDEs: The diffusion equation, implicit and explicit finite difference schemes
- Wednesday:
- More on the diffusion equation
- Laplace's equation, Jacobi and Gauss-Seidel iterations

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

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

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,

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

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

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

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

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

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

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

and off-diagonal matrix elements

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

The resulting matrix reads

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

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

$$\mathbf{S}_1 = \left( \begin{array}{cc} 1 & \mathbf{0}^{\mathbf{T}} \\ \mathbf{0} & \mathbf{P} \end{array} \right),$$

with $\mathbf{0}^{\mathbf{T}}$ being a zero row vector, $\mathbf{0}^{\mathbf{T}} = \{0, 0, \cdots\}$ 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).

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

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

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

$$\mathbf{Pv} = \mathbf{v} - 2\mathbf{u}(\mathbf{u}^T \mathbf{v}) = k\mathbf{e}, \tag{88}$$

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

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 $k = \pm v$.

Now we can rewrite Eq. (88) 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

$$2(\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), \tag{89}$$

which finally determines

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

In solving Eq. (89) 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.

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

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

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 $n$th 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.

Special to a $4 \times 4$ 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.

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 $\theta$ to give the element $a_{21}' = e' = 0$ then we have the first eigenvalue $= a_{11}' = d_1'$.

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.

The programs which performs these transformations are
matrix **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) |

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 $\alpha$ 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);
```

```
// 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()
```

# 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

$$\nabla^2 U(\mathbf{x}) = -4\pi\rho(\mathbf{x}).$$

Diffusion equation

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

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

## Explicit Scheme for the Diffusion Equation

In one dimension we have thus the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t}, \tag{90}$$

or

$$u_{xx} = u_t, \tag{91}$$

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

$$u(x,0) = g(x) \qquad 0 \le x \le L \tag{92}$$

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

$$u(0,t) = a(t) \qquad t \ge 0, \tag{93}$$

and

$$u(L,t) = b(t) \qquad t \ge 0, \tag{94}$$

where $a(t)$ and $b(t)$ are two functions which depend on time only, while $g(x)$ depends only on the position $x$.

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}, \tag{95}$$

and

$$u_{xx} \approx \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{96}$$

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

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{97}$$

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

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha) u_{i,j} + \alpha u_{i+1,j}. \tag{98}$$

$$V_{j+1} = AV_j$$

with

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

yielding

$$V_{j+1} = AV_j = \cdots = A^j V_0$$

The explicit scheme, although being rather simple to implement has a very weak stability condition given by

$$\Delta t/\Delta x^2 \leq 1/2 \tag{99}$$

# 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

```
!   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
```

- 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

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

## Partial Differential Equations (PDEs)

- Monday: Repetition from last week
- Laplace's equation, Jacobi and Gauss-Seidel iterations
- Begin of wave function and discussion of project 4
- Wednesday:
- Wave equation in two dimensions and project 4

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{100}$$

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

$$u_{xx} \approx \frac{u(x + h, y) - 2u(x, y) + u(x - h, y)}{h^2}, \tag{101}$$

and

$$u_{yy} \approx \frac{u(x, y + h) - 2u(x, y) + u(x, y - h)}{h^2}, \tag{102}$$

## Laplace's and Poisson's equations

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{103}$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{104}$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4} \left[ u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} \right]. \tag{105}$$

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

$$u_{i,j} = \frac{1}{4} \left[ u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} \right] + \rho_{i,j}. \tag{106}$$

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. (105) or Eq. (106) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (105). 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

```fortran
!  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
```

Consider first the two-dimensional wave equation for a vibrating square membrane given by the following initial and boundary conditions

$$
\begin{cases}
\lambda \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial^2 u}{\partial t^2} & x, y \in [0, 1], t \geq 0 \\
u(x, y, 0) = sin(\pi x) sin(2\pi y) & x, y \in (0, 1) \\
u = 0 \quad \text{boundary} & t \geq 0 \\
\partial u / \partial t |_{t=0} = 0 & x, y \in (0, 1)
\end{cases}
.
$$

The boundary is defined by $x = 0$, $x = 1$, $y = 0$ and $y = 1$. Here we set $\lambda = 1$.

# Two-dimensional wave equation

Our equations depend on three variables whose discretized versions are now

$$\begin{cases} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{cases}, \tag{107}$$

and we will let $\Delta x = \Delta y = h$ and $n_x = n_y$ for the sake of simplicity. We have now the following discretized partial derivatives

$$u_{xx} \approx \frac{u_{i+1,j}^l - 2u_{i,j}^l + u_{i-1,j}^l}{h^2}, \tag{108}$$

and

$$u_{yy} \approx \frac{u_{i,j+1}^l - 2u_{i,j}^l + u_{i,j-1}^l}{h^2}, \tag{109}$$

and

$$u_{tt} \approx \frac{u_{i,j}^{l+1} - 2u_{i,j}^l + u_{i,j}^{l-1}}{\Delta t^2}. \tag{110}$$

# Two-dimensional wave equation

We merge this into the discretized $2 + 1$-dimensional wave equation as

$$u_{i,j}^{l+1} = 2u_{i,j}^l - u_{i,j}^{l-1} + \frac{\Delta t^2}{h^2}\left(u_{i+1,j}^l - 4u_{i,j}^l + u_{i-1,j}^l + u_{i,j+1}^l + u_{i,j-1}^l\right), \qquad (111)$$

where again we have an explicit scheme with $u_{i,j}^{l+1}$ as the only unknown quantity. It is easy to account for different step lengths for $x$ and $y$. The partial derivative is treated in much the same way as for the one-dimensional case, except that we now have an additional index due to the extra spatial dimension, viz., we need to compute $u_{i,j}^{-1}$ through

$$u_{i,j}^{-1} = u_{i,j}^0 + \frac{\Delta t}{2h^2}\left(u_{i+1,j}^0 - 4u_{i,j}^0 + u_{i-1,j}^0 + u_{i,j+1}^0 + u_{i,j-1}^0\right), \qquad (112)$$

in our setup of the initial conditions.

# Two-dimensional wave equation

Check your results as function of the number of mesh points and in particular against the stability condition

$$\Delta t \leq \frac{1}{\sqrt{\lambda}} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$$

where $\Delta t$, $\Delta x$ and $\Delta y$ are the chosen step lengths. In our case $\Delta x = \Delta y = h$. How do we find this condition? In one dimension we can proceed as we did for the diffusion equation.

# Two-dimensional wave equation

The analytic solution of the wave equation in $2 + 1$ dimensions has a characteristic wave component which reads

$$u(x, y, t) = A \exp\left(i(k_x x + k_y y - \omega t)\right)$$

Then from

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

we get, with $u_i = \exp ikx_i$

$$u_{xx} \approx \frac{u_i}{\Delta x^2} \left(\exp ik\Delta x - 2 + \exp\left(-ik\Delta x\right)\right),$$

or

$$u_{xx} \approx 2\frac{u_i}{\Delta x^2} \left(cos(k\Delta x) - 1\right) = -4\frac{u_i}{\Delta x^2} sin^2(k\Delta x/2)$$

We get similar results for $t$ and $y$.

We have

$$\lambda \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \frac{\partial^2 u}{\partial t^2},$$

resulting in

$$\lambda \left( -4 \frac{u_{ij}^l}{\Delta x^2} \sin^2 \left( k_x \Delta x/2 \right) - 4 \frac{u_{ij}^l}{\Delta y^2} \sin^2 \left( k_y \Delta y/2 \right) \right) = -4 \frac{u_{ij}^l}{\Delta t^2} \sin^2 \left( \omega \Delta t/2 \right),$$

resulting in

$$\sin \left( \omega \Delta t/2 \right) = \pm \sqrt{\lambda} \Delta t \left( \frac{1}{\Delta x^2} \sin^2 \left( k_x \Delta x/2 \right) + \frac{1}{\Delta y^2} \sin^2 \left( k_y \Delta y/2 \right) \right)^{1/2}.$$

The squared sine functions can at most be unity. The frequency $\omega$ is real and our wave is neither damped nor amplified.

# Two-dimensional wave equation

We have

$$\sin\left(\omega\Delta t/2\right) = \pm\sqrt{\lambda}\Delta t\left(\frac{1}{\Delta x^2}\sin^2\left(k_x\Delta x/2\right) + \frac{1}{\Delta y^2}\sin^2\left(k_y\Delta y/2\right)\right)^{1/2}.$$

The squared sine functions can at most be unity. $\omega$ is real and our wave is neither damped nor amplified. The numerical $\omega$ must also be real which is the case when $\sin\left(\omega\Delta t/2\right)$ is less than or equal to unity, meaning that

$$\Delta t \leq \frac{1}{\sqrt{\lambda}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2}.$$

# Two-dimensional wave equation

We modify now the wave equation in order to consider a $2 + 1$ dimensional wave equation with a position dependent velocity, given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (\lambda(x, y)\nabla u).$$

If $\lambda$ is constant, we obtain the standard wave equation discussed in the two previous points. The solution $u(x, y, t)$ could represent a model for water waves. It represents then the surface elevation from still water. We will model $\lambda$ as

$$\lambda = gH(x, y),$$

with $g$ being the acceleration of gravity and $H(x, y)$ is the still water depth.

The function $H(x, y)$ simulates the water depth using for example measurements of still water depths in say a fjord or the north sea. The boundary conditions are then determined by the coast lines as discussed in point d) below. We have assumed that the vertical motion is negligible and that we deal with long wavelenghts $\tilde{\lambda}$ compared with the depth of the sea $H$, that is $\tilde{\lambda}/H \gg 1$. We will also neglect Coriolis effects.

You can discretize

$$\nabla \cdot (\lambda(x,y)\nabla u) = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right),$$

as follows using again a quadratic domain for $x$ and $y$:

$$\frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) \approx \frac{1}{\Delta x}\left(\lambda_{i+1/2,j}\left[\frac{u^l_{i+1,j}-u^l_{i,j}}{\Delta x}\right] - \lambda_{i-1/2,j}\left[\frac{u^l_{i,j}-u^l_{i-1,j}}{\Delta x}\right]\right),$$

and

$$\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right) \approx \frac{1}{\Delta y}\left(\lambda_{i,j+1/2}\left[\frac{u^l_{i,j+1}-u^l_{i,j}}{\Delta y}\right] - \lambda_{i,j-1/2}\left[\frac{u^l_{i,j}-u^l_{i,j-1}}{\Delta y}\right]\right).$$

How did we do that? Look at the derivative wrt $x$ only:
First we compute the derivative

$$\frac{d}{dx}\left(\lambda(x)\frac{du}{dx}\right)|_{x=x_i} \approx \frac{1}{\Delta x}\left(\lambda\frac{du}{dx}|_{x=x_{i+1/2}} - \lambda\frac{du}{dx}|_{x=x_{i-1/2}}\right),$$

where we approximated it at the midpoint by going half a step to the right and half a step to the left. Then we approximate

$$\lambda\frac{du}{dx}|_{x=x_{i+1/2}} \approx \lambda_{x_{i+1/2}}\frac{u_{i+1} - u_i}{\Delta x},$$

and similarly for $x = x_i - 1/2$.

We assume that we can approximate the coastline with a quadratic grid. As boundary condition at the coastline we will employ

$$\frac{\partial u}{\partial n} = \nabla u \cdot \mathbf{n} = 0,$$

where $\partial u / \partial n$ is the derivative in the direction normal to the boundary.

Here you must pay particular attention to the endpoints.

## Two-dimensional wave equation

We are going to model the impact of an earthquake on sea water. This is normally modelled via an elevation of the sea bottom. We will assume that the movement of the sea bottom is very rapid compared with the period of the propagating waves. This means that we can approximate the bottom elevation with an initial surface elevation. The initial conditions are then given by (with $L$ the length of the grid)

$$u(x, y, 0) = f(x, y) \qquad x, y \in (0, L),$$

and

$$\partial u / \partial t|_{t=0} = 0 \qquad x, y \in (0, L).$$

We will approximate the initial elevation with the function

$$f(x, y) = A_0 \exp\left(-\left[\frac{x - x_c}{\sigma_x}\right]^2 - \left[\frac{y - y_c}{\sigma_y}\right]^2\right),$$

where $A_0$ is the elevation of the surface and is typically $1 - 2$ m. The variables $\sigma_x$ and $\sigma_y$ represent the extensions of the surface elevation. In this project we will let $\sigma_x = 80$ km and $\sigma_y = 200$ km. The 2004 tsunami had extensions of approximately 200 and 1000 km, respectively.

The variables $x_c$ and $y_c$ represent the epicentre of the earthquake.

We need also to model the sea bottom and the function $\lambda(x, y) = gH(x, y)$. We assume that we can model the sea bottom as depicted in the following figure, with a water depth of 5000 m and a surface elevation of 2 m.

We assume the sea bottom depends only on the variable $x$ and has depth 5000 m before it starts increasing towards the coastline. We fix the angle $\theta = 1$ degree. From the figure you will be asked below to model the $x$ dependence of $H(x)$.

### Numerical integration and intro to Monte Carlo methods

- Monday: Repetition from last week and discussion of project 4
- Numerical integration, from the trapezoidal rule to Gaussian quadrature
- (Parallel computation postponed to project 5)
- Wednesday:
- Gaussian quadrature and introduction to Monte Carlo methods (chap 8.1-8.2)

# 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.

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

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 $\omega$ and $x$ are the weights and the chosen mesh points, respectively. Simpson's rule gives

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

for the weights, while the trapezoidal rule resulted in

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

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 $\omega_n$ can be chosen to satisfy $N$ linear equations

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

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

In the Lagrange representation this interpolation polynomial is given by

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

with the Lagrange factors

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

Example: $n = 1$

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

which we recognize as the equation for a straight line.

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

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

with

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

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

The local error for the trapezoidal rule is

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

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

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

and the global error

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

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

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 $\omega$ 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

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

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

# Gaussian Quadrature

## Weight Functions

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

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

# 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

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

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

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

and the recursion relation

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

# Laguerre

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

These polynomials arise from the solution of the differential equation

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

where $l$ is an integer $l \geq 0$ and $\lambda$ 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

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

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.

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.

## Plan for Monte Carlo Lectures

- This week: intro, random numbers, MC integration and probability distribution functions (PDFs)
- Next week: PDFs, MC integration and random walks.
- Third week: random walks and statistical physics, presentation of project 5.
- Fourth week: Statistical physics
- Fifth and sixth week: Statistical physics and quantum Monte Carlo

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 | $\{x_1, x_2, x_3, \ldots, x_N\}$ | $[a, b]$ |
| probability | $p(x_i)$ | $p(x)dx$ |
| Cumulative | $P_i = \sum_{l=1}^{i} p(x_l)$ | $P(x) = \int_a^x p(t)dt$ |
| Positivity | $0 \leq p(x_i) \leq 1$ | $p(x) \geq 0$ |
| Positivity | $0 \leq P_i \leq 1$ | $0 \leq P(x) \leq 1$ |
| Monotonuous | $P_i \geq P_j$ if $x_i \geq x_j$ | $P(x_i) \geq P(x_j)$ if $x_i \geq x_j$ |
| Normalization | $P_N = 1$ | $P(b) = 1$ |

## Expectation Values

- Discrete PDF

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

  provided that the sums (or integrals) $\sum_{i=1}^{N} p(x_i)$ converge absolutely (viz , $\sum_{i=1}^{N} |p(x_i)|$ converges)

- Continuous PDF

$$E[x^k] = \langle x^k \rangle = \int_a^b x^k p(x) dx,$$

- Function $f(x)$

$$E[f^k] = \langle f^k \rangle = \int_a^b f^k p(x) dx,$$

- Variance

$$\sigma_f^2 = E[f^2] - (E[f])^2 = \langle f^2 \rangle - \langle f \rangle^2$$

# Important PDFs

- uniform distribution

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$

  which gives for $a = 0, b = 1$ $p(x) = 1$ for $x \in [0, 1]$ and zero else.

- exponential distribution

$$p(x) = \alpha e^{-\alpha x},$$

  with probability different from zero in $[0, \infty]$

- normal distribution (Gaussian)

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

  with probability different from zero in $[-\infty, \infty]$

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

$$\langle H \rangle =$$

$$\frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \ldots d\mathbf{R}_N \Psi^*(\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N) H(\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N) \Psi(\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \ldots d\mathbf{R}_N \Psi^*(\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N) \Psi(\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_N)},$$

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).

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.

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 $\sigma^2$ and the standard deviation $\sigma$. 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 $\sigma_N$ as small as possible after $N$ samples.* The results from one sample represents, since we are using concepts from statistics, a 'measurement'.

# But what do we gain by Monte Carlo Integration?

- We saw that the trapezoidal rule carries a truncation error $O(h^2)$, with $h$ the step length.

- Quadrature rules such as Newton-Cotes have a truncation error which goes like $\sim O(h^k)$, with $k \geq 1$. Recalling that the step size is defined as $h = (b-a)/N$, we have an error which goes like $\sim N^{-k}$.

- Monte Carlo integration is more efficient in higher dimensions. Assume that our integration volume is a hypercube with side $L$ and dimension $d$. This cube contains hence $N = (L/h)^d$ points and therefore the error in the result scales as $N^{-k/d}$ for the traditional methods.

- The error in the Monte carlo integration is however independent of $d$ and scales as $\sigma \sim 1/\sqrt{N}$, always!

- Comparing this with traditional methods, shows that Monte Carlo integration is more efficient than an order-k algorithm when $d > 2k$

## 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

```cpp
// 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;
```

| $N$ | $I$ | $\sigma_N$ |
|---|---|---|
| 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. This is the topic of importance sampling.

# 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.

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 $\Delta t$ 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 $\Delta t$ 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 $\Delta t$.

## 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 $\omega$, 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 $\tau$ 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

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) + \omega_X N_X(t).$$

Probability for a decay of a particle during a time step $\Delta t$ is

$$\frac{\Delta N(t)}{N(t)\Delta t} = -\lambda$$

$\lambda$ 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 $\lambda$ 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)

```
   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.
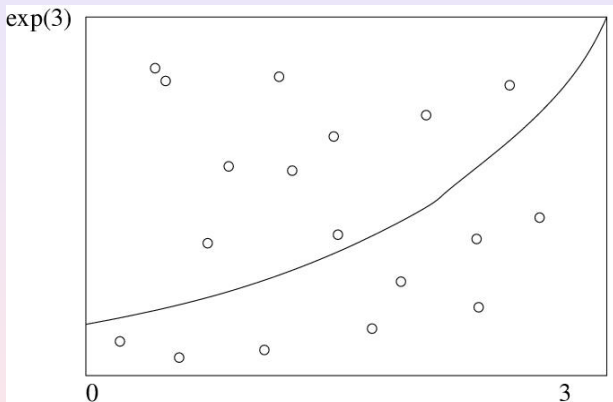
# Acceptance-Rejection Method

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
```

Monte Carlo integration, random numbers, random walks and Markov chains

- Monday: Repetition from last week
- Monte Carlo integration, importance sampling and transformation of variables
- Random numbers and covariance
- Wednesday: (Marius Lysebo)
- Discussion of Markov chains and link to diffusion equation
- Random walks

Till now we have not paid much attention to speed and possible optimization possibilities inherent in the various compilers. We have compiled and linkes as

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

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

```
man c++  >  out_to_file
```

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

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

This is the recommended option. **But you must check that you get the same results as previously**.

## Optimization and profiling

It is also useful to profile your program under the development stage. You would then compile with (Mac and unix/linux)

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

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

```
gprof mycode.exe >  out_to_profile
```

When you have profiled properly your code, you must take this option as it increases your CPU expenditure.

## Optimization and profiling

Other hints

- avoid if test or call to functions inside loops, if possible.
- avoid multiplication with constants inside loop is possible

Bad code

```
for i = 1:n
    a(i) = b(i) +c*d
    e = g(k)
end
```

Better code

```
temp = c*d
for i = 1:n
    a(i) = b(i) + temp
end
e = g(k)
```

The starting point is always the uniform distribution

$$p(x)dx = \begin{cases} dx & 0 \le x \le 1 \\ 0 & else \end{cases}$$

with $p(x) = 1$ and satisfying

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

All random number generators provided in the program library generate numbers in this domain.

When we attempt a transformation to a new variable $x \rightarrow y$ 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

$$p(y)dy = \left\{ \begin{array}{cc} \frac{dy}{b-a} & a \leq y \leq b \\ 0 & else \end{array} \right.$$

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!

Assume that

$$p(y) = e^{-y},$$

which is the exponential distribution, important for the analysis of e.g., radioactive decay. Again, $p(x)$ is given by the uniform distribution with $x \in [0, 1]$, and with the assumption that the probability is conserved we have

$$p(y)dy = e^{-y}dy = dx,$$

which yields after integration

$$x(y) = P(y) = \int_0^y \exp(-y')dy' = 1 - \exp(-y),$$

or

$$y(x) = -ln(1 - x).$$

This gives us the new random variable $y$ in the domain $y \in [0, \infty)$ determined through the random variable $x \in [0, 1]$ generated by our favorite random generator.

# Example 2, from Uniform to Exponential

This means that if we can factor out $\exp(-y)$ from an integrand we may have

$$I = \int_0^\infty F(y)dy = \int_0^\infty \exp(-y)G(y)dy$$

which we rewrite as

$$\int_0^\infty \exp(-y)G(y)dy = \int_0^\infty \frac{dx}{dy}G(y)dy \approx \frac{1}{N}\sum_{i=1}^N G(y(x_i)),$$

where $x_i$ is a random number in the interval [0,1].

Note that in practical implementations, our random number generators for the uniform distribution never return exactly 0 or 1, but we we may come very close. We should thus in principle set $x \in (0,1)$.

# Example 2, from Uniform to Exponential

The algorithm is rather simple. In the function which sets up the integral, we simply need the random number generator for the uniform distribution in order to obtain numbers in the interval [0,1]. We obtain $y$ by the taking the logarithm of $(1 - x)$. Our calling function which sets up the new random variable $y$ may then include statements like

```
.....
idum=-1;
x=ran0(&idum);
y=-log(1.-x);
.....
```

## Example 3

Another function which provides an example for a PDF is

$$p(y)dy = \frac{dy}{(a+by)^n},$$

with $n > 1$. It is normalizable, positive definite, analytically integrable and the integral is invertible, allowing thereby the expression of a new variable in terms of the old one. The integral

$$\int_0^\infty \frac{dy}{(a+by)^n} = \frac{1}{(n-1)ba^{n-1}},$$

gives

$$p(y)dy = \frac{(n-1)ba^{n-1}}{(a+by)^n}dy,$$

which in turn gives the cumulative function

$$x(y) = P(y) = \int_0^y \frac{(n-1)ba^{n-1}}{(a+bx)^n}dy' =,$$

resulting in

$$y = \frac{a}{b}\left((1-x)^{-1/(n-1)} - 1\right).$$

## Example 4, from Uniform to Normal

For the normal distribution, expressed here as

$$g(x, y) = \exp\left(-(x^2 + y^2)/2\right) dx dy.$$

it is rather difficult to find an inverse since the cumulative distribution is given by the error function $erf(x)$.

If we however switch to polar coordinates, we have for $x$ and $y$

$$r = \left(x^2 + y^2\right)^{1/2} \qquad \theta = tan^{-1}\frac{x}{y},$$

resulting in

$$g(r, \theta) = r \exp\left(-r^2/2\right) dr d\theta,$$

where the angle $\theta$ 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 $2\pi$.

## 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. (355) 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);
```

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.

| $N$ | $I_{cr}$ | $\sigma_{cr}$ | $I_{is}$ | $\sigma_{is}$ |
|---|---|---|---|---|
| 10000 | 3.13395E+00 | 4.22881E-01 | 3.14163E+00 | 6.49921E-03 |
| 100000 | 3.14195E+00 | 4.11195E-01 | 3.14163E+00 | 6.36837E-03 |
| 1000000 | 3.14003E+00 | 4.14114E-01 | 3.14128E+00 | 6.39217E-03 |
| 10000000 | 3.14213E+00 | 4.13838E-01 | 3.14160E+00 | 6.40784E-03 |

However, it is unfair to study one-dimensional integrals with MC methods!

# Multidimensional Integrals

When we deal with multidimensional integrals of the form

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \ldots \int_0^1 dx_d g(x_1, \ldots, x_d),$$

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 $g(x_1, \ldots, x_d)$ to

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

The task of this project is to integrate in a brute force manner a six-dimensional integral which is used to determine the ground state correlation energy between two electrons in a helium atom. We will employ both Gauss-Legendre quadrature and Monte-Carlo integration. We assume that the wave function of each electron can be modelled like the single-particle wave function of an electron in the hydrogen atom. The single-particle wave function for an electron $i$ in the $1s$ state is given in terms of a dimensionless variable (the wave function is not properly normalized)

$$\mathbf{r}_i = x_i \mathbf{e}_x + y_i \mathbf{e}_y + z_i \mathbf{e}_z,$$

as

$$\psi_{1s}(\mathbf{r}_i) = e^{-\alpha r_i},$$

where $\alpha$ is a parameter and

$$r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}.$$

We will fix $\alpha = 2$, which should correspond to the charge of the helium atom $Z = 2$.

# Switch to spherical coordinates

Useful to change to spherical coordinates

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

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

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

The ansatz for the wave function for two electrons is then given by the product of two $1s$ wave functions as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1 + r_2)}.$$

Note that it is not possible to find an analytic solution to Schrödinger's equation for two interacting electrons in the helium atom.

The integral we need to solve is the quantum mechanical expectation value of the correlation energy between two electrons, namely

$$\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1 + r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} = \frac{5\pi^2}{16^2} = 0.192765711. \quad (113)$$

Note that our wave function is not normalized. There is a normalization factor missing, but for this project we don't need to worry about that.

a) Use Gauss-Legendre quadrature and compute the integral by integrating for each variable $x_1$, $y_1$, $z_1$, $x_2$, $y_2$, $z_2$ from $-\infty$ to $\infty$. How many mesh points do you need before the results converges at the level of the fourth leading digit? Hint: the single-particle wave function $e^{-\alpha r_i}$ is more or less zero at $r_i \approx 10 - 15$. You can therefore replace the integration limits $-\infty$ and $\infty$ with $-10$ and $10$, respectively. You need to check that this approximation is satisfactory.

b) Compute the same integral but now with brute force Monte Carlo and compare your results with those from the previous point. Discuss the differences. With bruce force we mean that you should use the uniform distribution.

c) Improve your brute force Monte Carlo calculation by using importance sampling. Hint: use the exponential distribution. Does the variance decrease? Does the CPU time used compared with the brute force Monte Carlo decrease in order to achieve the same accuracy? Comment your results.

# Example, six-dimensional integral, Gauss-Legendre

```
      double *x = new double [N];
      double *w = new double [N];
//    set up the mesh points and weights
      gauleg(a,b,x,w, N);

//    evaluate the integral with the Gauss-Legendre method
//    Note that we initialize the sum
      double int_gauss = 0.;
      for (int i=0;i<N;i++){
      for (int j = 0;j<N;j++){
      for (int k = 0;k<N;k++){
      for (int l = 0;l<N;l++){
      for (int m = 0;m<N;m++){
      for (int n = 0;n<N;n++){
         int_gauss+=w[i]*w[j]*w[k]*w[l]*w[m]*w[n]
        *int_function(x[i],x[j],x[k],x[l],x[m],x[n]);
       }}}}}
      }
```

```
//  this function defines the function to integrate
double int_function(double x1, double y1, double z1, double x2, double
{
    double alpha = 2.;
// evaluate the different terms of the exponential
    double exp1=-2*alpha*sqrt(x1*x1+y1*y1+z1*z1);
    double exp2=-2*alpha*sqrt(x2*x2+y2*y2+z2*z2);
    double deno=sqrt(pow((x1-x2),2)+pow((y1-y2),2)+pow((z1-z2),2));
  if(deno <pow(10.,-6.)) { return 0;}
  else return exp(exp1+exp2)/deno;
} // end of function to evaluate
```

# Example, six-dimensional integral, brute force MC

```
double int_mc = 0.;  double variance = 0.;
double sum_sigma= 0. ; long idum=-1 ;
double length=1.5; // we fix the max size of the box to L=3
double volume=pow((2*length),6.);

// evaluate the integral with importance sampling
for ( int i = 1;  i <= n; i++){
// x[] contains the random numbers for all dimensions
    for (int j = 0; j< 6; j++) {
        x[j]=-length+2*length*ran0(&idum); // Maps U[0,1] to U[-L,L
    }
    fx=brute_force_MC(x);
    int_mc += fx;
    sum_sigma += fx*fx;
}
int_mc = int_mc/((double) n );
sum_sigma = sum_sigma/((double) n );
variance=sum_sigma-int_mc*int_mc;
....
```

# Example, six-dimensional integral, brute force MC

```
double brute_force_MC(double *x)
{
   double alpha = 2.;
// evaluate the different terms of the exponential
   double exp1=-2*alpha*sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
   double exp2=-2*alpha*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5]);
   double deno=sqrt(pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[
   double value=exp(exp1+exp2)/deno;
return value;
} // end function for the integrand
```

# Example, six-dimensional integral, importance sampling

```
double int_mc = 0.;  double variance = 0.;
double sum_sigma= 0. ; long idum=-1 ;
// The 'volume' contains 4 jacobideterminants(pi,pi,2pi,2pi) and a sca
double volume=4*pow(acos(-1.),4.)*1./16;
//    evaluate the integral with importance sampling
for ( int i = 1;  i <= n; i++){
   for (int j = 0; j < 2; j++) {
y=ran0(&idum);
x[j]=-0.25*log(1.-y);
     }
   for (int j = 2; j < 4; j++) {
x[j] = 2*acos(-1.)*ran0(&idum);
   }
   for (int j = 4; j < 6; j++) {
 x[j] = acos(-1.)*ran0(&idum);
     }
fx=gaussian_MC(x);
       ....
```

# Example, six-dimensional integral, importance sampling

```
// this function defines the integrand to integrate

double  gaussian_MC(double *x)
{
double num=x[0]*x[0]*x[1]*x[1]*sin(x[4])*sin(x[5]);
double deno=sqrt(x[0]*x[0]+x[1]*x[1]-2*x[0]*x[1]*
      (sin(x[4])*sin(x[5])*cos(x[2]-x[3])+cos(x[4])*cos(x[5])));
return num/deno;
} // end function for the integrand
```

# Test Runs and Comparison with Brute Force and Gauss-Legendre

The suffix *br* stands for the brute force approach while *is* stands for the use of importance sampling.

| $N$ | $I_{br}$ | $\sigma_{br}$ | time(s) | $I_{is}$ | $\sigma_{is}$ | time(s) |
|---|---|---|---|---|---|---|
| 1E6 | 0.19238 | 3.85124E-4 | 0.6 | 0.19176 | 1.01515E-4 | 1.4 |
| 10E6 | 0.18607 | 1.18053E-4 | 6 | 0.192254 | 1.22430E-4 | 14 |
| 100E6 | 0.18846 | 4.37163E-4 | 57 | 0.192720 | 1.03346E-4 | 138 |
| 1000E6 | 0.18843 | 1.35879E-4 | 581 | 0.192789 | 3.28795E-5 | 1372 |

Gauss-Legendre results:

| $N$ | time(s) | $I_n$ | $|I_n - I|$ |
|---|---|---|---|
| 20 | 31 | 0.18047 | 1.123E-2 |
| 30 | 354 | 0.18501 | 7.76E-3 |
| 40 | 1999 | 0.18653 | 6.24E-3 |
| 50 | 7578 | 0.18722 | 5.54E-3 |

# Example: 6-dimensional Integral

As a further example, consider the following six-dimensional integral

$$\int_{-\infty}^{\infty} \mathbf{dx dy} g(\mathbf{x}, \mathbf{y}),$$

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

$$\frac{1}{\sqrt{\pi}} \exp\left(-x^2\right),$$

and through

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

we can rewrite our integral as

$$\int f(x_1, \ldots, x_d) F(x_1, \ldots, x_d) \prod_{i=1}^{6} dx_i,$$

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;
......
```

```
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);
```

```
..........
//    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;
.............
```

```
// 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
```

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 |
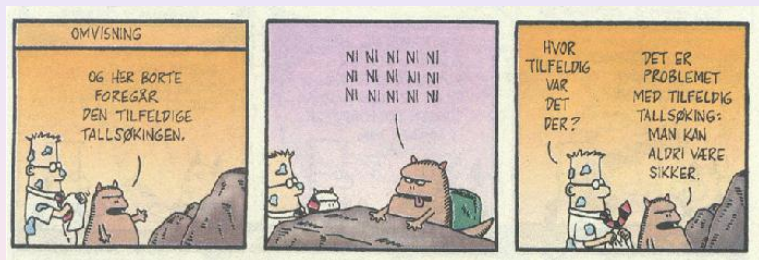
# Random Numbers

Most used are so-called 'Linear congruential'

$$N_i = (aN_{i-1} + c)MOD(M),$$

and to find a number in $x \in [0, 1]$

$$x_i = N_i/M$$

$M$ is called the period and should be as big as possible. The start value is $N_0$ and is called the seed.

- The random variables should result in the uniform distribution

- No correlations between numbers (zero covariance)

- As big as possible period $M$

- Fast algo

# Random Numbers

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most $M$. If however the parameters $a$ and $c$ are badly chosen, the period may be even shorter.
Consider the following example

$$N_i = (6N_{i-1} + 7)\mathrm{MOD}(5),$$

with a seed $N_0 = 2$. This generator produces the sequence
$4, 1, 3, 0, 2, 4, 1, 3, 0, 2, \ldots \ldots$, i.e., a sequence with period 5. However, increasing $M$ may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\mathrm{MOD}(54),$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \ldots$, a period of just 2.

# Random Numbers

Typical periods for the random generators provided in the program library are of the order of $\sim 10^9$ or larger. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could make a shift register generator whose $l$th number is the sum of the $l-i$th and $l-j$th values with modulo $M$,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M). \tag{114}$$

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 $2^{94}$.

Using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j}) \tag{115}$$

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 $\text{IEOR}(m, n)$

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 $r < q$.

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

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq+r))\mathrm{MOD}(M),$$

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 $\sigma$ 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 $N \to \infty$ 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 $\mu$ 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 |
| $\mu$ | 0.4997 | 0.5018 | 0.4992 | 0.4990 |
| $\sigma$ | 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.

# 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

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

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

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

This change from one microstate $i$ (or spin configuration) to another microstate $j$ is the **configuration space** analogue to a random walk on a lattice. Instead of jumping from one place to another in space, we 'jump' from one microstate to another.

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

$$j(x, t) = -D\frac{\partial w(x, t)}{\partial x}, \tag{116}$$

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

$$\frac{\partial j(x, t)}{\partial x} = -\frac{\partial w(x, t)}{\partial t}, \tag{117}$$

which leads to

$$\frac{\partial w(x, t)}{\partial t} = D\frac{\partial^2 w(x, t)}{\partial x^2}, \tag{118}$$

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

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

$$\langle x(t) \rangle = \int_{-\infty}^{\infty} x w(x,t)dx, \qquad (119)$$

or

$$\langle x^2(t) \rangle = \int_{-\infty}^{\infty} x^2 w(x,t)dx, \qquad (120)$$

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

$$\langle f(x,t) \rangle = \int_{-\infty}^{\infty} f(x,t)w(x,t)dx. \qquad (121)$$

# 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

$$\int_{-\infty}^{\infty} w(x, t)dx = 1 \tag{122}$$

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

$$w(x = \pm\infty, t) = 0 \qquad \frac{\partial^n w(x, t)}{\partial x^n}|_{x=\pm\infty} = 0, \tag{123}$$

implying that when we study the time-derivative $\partial\langle x(t)\rangle/\partial t$, we obtain after integration by parts and using Eq. (118)

$$\frac{\partial\langle x\rangle}{\partial t} = \int_{-\infty}^{\infty} x\frac{\partial w(x, t)}{\partial t}dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 w(x, t)}{\partial x^2}dx, \tag{124}$$

leading to

$$\frac{\partial\langle x\rangle}{\partial t} = Dx\frac{\partial w(x, t)}{\partial x}|_{x=\pm\infty} - D\int_{-\infty}^{\infty} \frac{\partial w(x, t)}{\partial x}dx, \tag{125}$$

implying that

$$\frac{\partial\langle x\rangle}{\partial t} = 0. \tag{126}$$

# 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

$$\frac{\partial \langle x^2 \rangle}{\partial t} = Dx^2 \frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - 2D \int_{-\infty}^{\infty} x \frac{\partial w(x,t)}{\partial x} dx, \tag{127}$$

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

$$\frac{\partial \langle x^2 \rangle}{\partial t} = -Dxw(x,t)|_{x=\pm\infty} + 2D \int_{-\infty}^{\infty} w(x,t) dx = 2D, \tag{128}$$

leading to

$$\langle x^2 \rangle = 2Dt, \tag{129}$$

and the variance as

$$\langle x^2 \rangle - \langle x \rangle^2 = 2Dt. \tag{130}$$

The root mean square displacement after a time $t$ is then

$$\sqrt{\langle x^2 \rangle - \langle x \rangle^2} = \sqrt{2Dt}. \tag{131}$$

# 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

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

# 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

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i, n+1) - w(i, n)}{\Delta t}, \tag{132}$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1, n) + w(i-1, n) - 2w(i, n)}{(\Delta x)^2}, \tag{133}$$

resulting in the discretized diffusion equation

$$\frac{w(i, n+1) - w(i, n)}{\Delta t} = D\frac{w(i+1, n) + w(i-1, n) - 2w(i, n)}{(\Delta x)^2}, \tag{134}$$

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

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \begin{cases} \frac{1}{2} & |i - j| = 1 \\ 0 & \text{else} \end{cases} \tag{135}$$

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

$$w_i(t = \epsilon) = W(j \rightarrow i)w_j(t = 0). \tag{136}$$

# 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

$$\sum_i w_i(t) = 1, \tag{137}$$

and

$$\sum_j W(j \to i) = 1. \tag{138}$$

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.

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

$$w_i(t_n) = \sum_j W_{ij}(t_n)w_j(0), \tag{139}$$

and defining

$$W(il - jl, n\epsilon) = (W^n(\epsilon))_{ij} \tag{140}$$

we obtain

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij}w_j(0), \tag{141}$$

or in matrix form

$$\hat{w(n\epsilon)} = \hat{W}^n(\epsilon)\hat{w}(0). \tag{142}$$

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

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{143}$$

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

$$\hat{R} = \left(\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right), \tag{144}$$

and

$$\hat{L} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right). \tag{145}$$

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

$$\hat{L}\hat{R} = \hat{R}\hat{L} = 1, \tag{146}$$

and

$$\hat{L} = \hat{R}^{-1} \tag{147}$$

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

$$\hat{L}\hat{R} = \sum_k \hat{L}_{ik}\hat{R}_{kj} = \sum_k \delta_{(i+1),k}\delta_{k,(j+1)} = \delta_{i+1,j+1} = \delta_{i,j}, \tag{148}$$

and only the diagonal matrix elements are different from zero.

For the first time step we have thus

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{149}$$

and using the properties in Eqs. (146) and (147) we have after two time steps

$$\hat{W}^2(2\epsilon) = \frac{1}{4}\left(\hat{L}^2 + \hat{R}^2 + 2\hat{R}\hat{L}\right), \tag{150}$$

and similarly after three time steps

$$\hat{W}^3(3\epsilon) = \frac{1}{8}\left(\hat{L}^3 + \hat{R}^3 + 3\hat{R}\hat{L}^2 + 3\hat{R}^2\hat{L}\right). \tag{151}$$

Using the binomial formula

$$\sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) \hat{a}^{k} \hat{b}^{n-k} = (a+b)^{n}, \tag{152}$$

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

$$\hat{W}^{n}(n\epsilon)) = \frac{1}{2^{n}} \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) \hat{R}^{k} \hat{L}^{n-k}, \tag{153}$$

or

$$\hat{W}^{n}(n\epsilon)) = \frac{1}{2^{n}} \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) \hat{L}^{n-2k} = \frac{1}{2^{n}} \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) \hat{R}^{2k-n}, \tag{154}$$

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

$$W(il - jl, n\epsilon) = \left\{ \begin{array}{cc} \frac{1}{2^n} \left( \begin{array}{c} n \\ \frac{1}{2}(n+i-j) \end{array} \right) & |i - j| \leq n \\ 0 & \text{else} \end{array} \right. , \qquad (155)$$

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. (142) we have

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0) = \sum_j \frac{1}{2^n} \left( \begin{array}{c} n \\ \frac{1}{2}(n + i - j) \end{array} \right) \delta_{j,0}, \tag{156}$$

resulting in

$$w_i(n\epsilon) = \frac{1}{2^n} \left( \begin{array}{c} n \\ \frac{1}{2}(n + i) \end{array} \right) \qquad |i| \leq n \tag{157}$$

Using the recursion relation for the binomials

$$\left( \begin{array}{c} n+1 \\ \frac{1}{2}(n+1+i)) \end{array} \right) = \left( \begin{array}{c} n \\ \frac{1}{2}(n+i+1) \end{array} \right) + \left( \begin{array}{c} n \\ \frac{1}{2}(n+i) - 1 \end{array} \right) \qquad (158)$$

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

$$w(x, t) = w(il, n\epsilon) = w_i(n\epsilon), \qquad (159)$$

$$w(x, t + \epsilon) = \frac{1}{2}w(x + l, t) + \frac{1}{2}w(x - l, t), \qquad (160)$$

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

$$\frac{w(x, t+\epsilon) - w(x, t)}{\epsilon} = \frac{l^2}{2\epsilon} \frac{w(x+l, t) - 2w(x, t) + w(x-l, t)}{l^2}, \qquad (161)$$

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.

## Markov chains, Statistical Physics, Spins Systems and Phase Transitions

- Monday: Repetition from last week, Markov chains, detailed balance, ergodicity and entropy
- Discussion of the Metropolis algorithm and how to simulate the Boltzmann probability
- Wednesday:
- Presentation of project 5
- How to simulate the Boltzmann probability and statistical physics (chap 10)

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

$$w_i(t = \epsilon) = W(j \to i)w_j(t = 0).$$

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

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

We call $W_{ij}$ for the transition probability and we represent it as a matrix.

Both *W* and *w* represent probabilities and they have to be normalized, meaning that that at each time step we have

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

and

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

Further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$. We can thus write the action of *W* as

$$w_i(t+1) = \sum_j W_{ij} w_j(t),$$

or as vector-matrix relation

$$\hat{\mathbf{w}}(t+1) = \hat{\mathbf{W}}\hat{\mathbf{w}}(t),$$

and if we have that $||\hat{\mathbf{w}}(t+1) - \hat{\mathbf{w}}(t)|| \to 0$, we say that we have reached the most likely state of the system, the so-called steady state or equilibrium state. Another way of phrasing this is

$$\mathbf{w}(t = \infty) = \mathbf{W}\mathbf{w}(t = \infty).$$

# Brownian Motion and Markov Processes, a simple Example

Consider the simple $3 \times 3$ matrix $\hat{W}$

$$\hat{W} = \begin{pmatrix} 1/4 & 1/8 & 2/3 \\ 3/4 & 5/8 & 0 \\ 0 & 1/4 & 1/3 \end{pmatrix},$$

and we choose our initial state as

$$\hat{w}(t = 0) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

The first iteration is

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

resulting in

$$\hat{w}(t = \epsilon) = \begin{pmatrix} 1/4 \\ 3/4 \\ 0 \end{pmatrix}.$$

The next iteration results in

$$w_i(t = 2\epsilon) = W(j \rightarrow i)w_j(t = \epsilon),$$

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

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

(162)

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 $W$!**

We have after $t$-steps

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t \hat{\mathbf{w}}(0),$$

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 $\lambda_i$ the $i^{\text{th}}$ eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

# Brownian Motion and Markov Processes, what is happening?

If we assume that $\lambda_0$ 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 later statistical physics and quantum physics applications, we will relate these properties to correlation functions such as the time-correlation function.

That will allow us to define the so-called *equilibration time*, viz the time needed for the system to reach its most likely state. 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 $\mu$.
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 $i^{\text{th}}$ 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 $\lambda_0$, 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 $\tau_i$ are the correlation times for the system. They control also the time-correlation functions to be discussed in connection with project 5 and chapter 10 on statistical physics.

The longest correlation time is obviously given by the second largest eigenvalue $\tau_1$, 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 $\lambda_1$ and we simulate long enough, $\tau_1$ may well define the correlation time. In other cases we may not be able to extract a reliable result for $\tau_1$.

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

$$S = - \sum_i w_i ln(w_i), \qquad (163)$$

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 $x = L$.

```
// 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

```cpp
// 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 simple.

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

$$\frac{w_i}{w_j} = \exp\left(-\beta(E_i - E_j)\right)$$

- In general

$$W(i \rightarrow j) = g(i \rightarrow j)A(i \rightarrow j)$$

  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

$$\frac{g(j \rightarrow i)A(j \rightarrow i)}{g(i \rightarrow j)A(i \rightarrow j)} = \exp\left(-\beta(E_i - E_j)\right)$$

For a system which follows the Boltzmann distribution the Metropolis algorithm reads

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

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.

Want to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{164}$$

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

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{165}$$

with mass $m = 1$.

# Test of the Metropolis Algorithm

Want to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{166}$$

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

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{167}$$

with mass $m = 1$.

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 $\sigma_E$ 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
   }
```

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 |
| $\sigma_E$ | 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
```

# Most Common Ensembles in Statistical Physics

|                                          | Microcanonical | Canonical | Grand can. | Pressure can. |
|------------------------------------------|----------------|-----------|------------|---------------|
| Exchange of heat with the environment    | no             | yes       | yes        | yes           |
| Exchange of particles with the environemt | no            | no        | yes        | no            |
| Thermodynamical parameters               | $V, \mathcal{M}, \mathcal{D}$ <br> $E$ <br> $N$ | $V, \mathcal{M}, \mathcal{D}$ <br> $T$ <br> $N$ | $V, \mathcal{M}, \mathcal{D}$ <br> $T$ <br> $\mu$ | $P, \mathcal{H}, \mathcal{E}$ <br> $T$ <br> $N$ |
| Potential                                | Entropy        | Helmholtz | $PV$       | Gibbs         |
| Energy                                   | Internal       | Internal  | Internal   | Enthalpy      |

# Microcanonical Ensemble

Entropy

$$S = k_B ln\Omega \tag{168}$$

$$dS = \frac{1}{T}dE + \frac{p}{T}dV - \frac{\mu}{T}dN \tag{169}$$

Temperature

$$\frac{1}{k_B T} = \left( \frac{\partial ln\Omega}{\partial E} \right)_{N,V} \tag{170}$$

Pressure

$$\frac{p}{k_B T} = \left( \frac{\partial ln\Omega}{\partial V} \right)_{N,E} \tag{171}$$

Chemical potential

$$\frac{\mu}{k_B T} = - \left( \frac{\partial ln\Omega}{\partial N} \right)_{V,E} \tag{172}$$

# Canonical Ensemble

Helmholtz Free Energy

$$F = -k_B T \ln Z \tag{173}$$

$$dF = -SdT - pdV + \mu dN \tag{174}$$

Entropy

$$S = k_B \ln Z + k_B T \left( \frac{\partial \ln Z}{\partial T} \right)_{N,V} \tag{175}$$

Pressure

$$p = k_B T \left( \frac{\partial \ln Z}{\partial V} \right)_{N,T} \tag{176}$$

Chemical Potential

$$\mu = -k_B T \left( \frac{\partial \ln Z}{\partial N} \right)_{V,T} \tag{177}$$

Energy (internal only)

$$E = k_B T^2 \left( \frac{\partial \ln Z}{\partial T} \right)_{V,N} \tag{178}$$

Potential

$$pV = k_B T ln \Xi \tag{179}$$

$$d(pV) = SdT + Nd\mu + pdV \tag{180}$$

Entropy

$$S = k_B ln \Xi + k_B T \left( \frac{\partial ln \Xi}{\partial T} \right)_{V,\mu} \tag{181}$$

Particles

$$N = k_B T \left( \frac{\partial ln \Xi}{\partial \mu} \right)_{V,T} \tag{182}$$

Pressure

$$p = k_B T \left( \frac{\partial ln \Xi}{\partial V} \right)_{\mu,T} \tag{183}$$

Gibbs Free Energy

$$G = -k_B T ln\Delta \tag{184}$$

$$dG = -SdT + Vdp + \mu dN \tag{185}$$

Entropy

$$S = k_B ln\Delta + k_B T \left(\frac{\partial ln\Delta}{\partial T}\right)_{p,N} \tag{186}$$

Volume

$$V = -k_B T \left(\frac{\partial ln\Delta}{\partial p}\right)_{N,T} \tag{187}$$

Chemical potential

$$\mu = -k_B T \left(\frac{\partial ln\Delta}{\partial N}\right)_{p,T} \tag{188}$$

At a given temperature we have the probability distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{189}$$

with $\beta = 1/kT$ being the inverse temperature, $k$ the Boltzmann constant, $E_i$ is the energy of a state $i$ while $Z$ is the partition function for the canonical ensemble defined as

$$Z = \sum_{i=1}^{M} e^{-\beta E_i}, \tag{190}$$

where the sum extends over all states $M$. $P_i$ expresses the probability of finding the system in a given configuration $i$.

## Expectation Values

For a system described by the canonical ensemble, the energy is an expectation value since we allow energy to be exchanged with the surroundings (a heat bath with temperature $T$). This expectation value, the mean energy, can be calculated using the probability distribution $P_i$ as

$$\langle E \rangle = \sum_{i=1}^{M} E_i P_i(\beta) = \frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i}, \tag{191}$$

with a corresponding variance defined as

$$\sigma_E^2 = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{Z} \sum_{i=1}^{M} E_i^2 e^{-\beta E_i} - \left( \frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i} \right)^2. \tag{192}$$

If we divide the latter quantity with $kT^2$ we obtain the specific heat at constant volume

$$C_V = \frac{1}{kT^2} \left( \langle E^2 \rangle - \langle E \rangle^2 \right). \tag{193}$$

We can also write

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta}. \tag{194}$$

The specific heat is

$$C_V = \frac{1}{kT^2} \frac{\partial^2 lnZ}{\partial \beta^2} \tag{195}$$

These expressions link a physical quantity (in thermodynamics) with the microphysics given by the partition function. Statistical physics is the field where one relates microscopic quantities to observables at finite temperature.

$$\langle \mathcal{M} \rangle = \sum_i^M \mathcal{M}_i P_i(\beta) = \frac{1}{Z} \sum_i^M \mathcal{M}_i e^{-\beta E_i}, \tag{196}$$

and the corresponding variance

$$\sigma^2_{\mathcal{M}} = \langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 = \frac{1}{Z} \sum_{i=1}^M \mathcal{M}_i^2 e^{-\beta E_i} - \left( \frac{1}{Z} \sum_{i=1}^M \mathcal{M}_i e^{-\beta E_i} \right)^2. \tag{197}$$

This quantity defines also the susceptibility $\chi$

$$\chi = \frac{1}{kT} \left( \langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 \right). \tag{198}$$

## Phase Transitions

NOTE: Helmholtz free energy and canonical ensemble

$$F = \langle E \rangle - TS = -kTlnZ$$

meaning $lnZ = -F/kT = -F\beta$ and

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta} = \frac{\partial(\beta F)}{\partial \beta}.$$

and

$$C_V = -\frac{1}{kT^2}\frac{\partial^2(\beta F)}{\partial \beta^2}.$$

We can relate observables to various derivatives of the partition function and the free energy. When a given derivative of the free energy or the partition function is discontinuous or diverges (logarithmic divergence for the heat capacity from the Ising model) we talk of a phase transition of order of the derivative.

# Phase Transitions

- An important quantity is the correlation length ($\xi$, to be discussed next week). The correlation length defines the length scale at which the overall properties of a material start to differ from its bulk properties. It is the distance over which the fluctuations of the microscopic degrees of freedom (for example the position of atoms) are significantly correlated with each other. Usually it is of the order of few interatomic spacings for a solid.

- The correlation length $\xi$ depends however on external conditions such as pressure and temperature.

- A phase transition is marked by abrupt macroscopic changes as external parameters are changed, such as an increase of temperature.

- The point where a phase transition takes place is called a critical point.

**1** First order/discontinuous phase transitions: Two or more states on either side of the critical point also coexist exactly at the critical point. As we pass through the critical point we observe a discontinuous behavior of thermodynamical functions. The correlation length is normally finite at the critical point. Phenomena such as hysteris occur, viz. there is a continuation of state below the critical point into one above the critical point. This continuation is metastable so that the system may take a macroscopically long time to readjust. Classical example, melting of ice.

**2** Second order or continuous transitions: The correlation length diverges at the critical point, fluctuations are correlated over all distance scales, which forces the system to be in a unique critical phase. The two phases on either side of the critical point become identical. Smooth behavior of first derivatives of the partition function, while second derivatives diverge. Strong correlations make a perturbative treatment impossible. Renormalization group theory (FYS4410).

# Examples of Phase Transitions

| System | Transition | Order Parameter |
|--------|-----------|-----------------|
| Liquid-gas | Condensation/evaporation | Density difference $\Delta\rho = \rho_{liquid} - \rho_{gas}$ |
| Binary liquid | mixture/Unmixing | Composition difference |
| Quantum liquid | Normal fluid/superfluid | $<\phi>$, $\psi$ = wavefunction |
| Liquid-solid | Melting/crystallisation | Reciprocal lattice vector |
| Magnetic solid | Ferromagnetic | Spontaneous magnetisation $M$ |
| | Antiferromagnetic | Sublattice magnetisation $M$ |
| Dielectric solid | Ferroelectric | Polarization $P$ |
| | Antiferroelectric | Sublattice polarisation $P$ |

The model we will employ in our studies of phase transitions at finite temperature for magnetic systems is the so-called Ising model. In its simplest form the energy is expressed as

$$E = -J \sum_{<kl>}^{N} s_k s_l - B \sum_{k}^{N} s_k, \tag{199}$$

with $s_k = \pm 1$, $N$ is the total number of spins, $J$ is a coupling constant expressing the strength of the interaction between neighboring spins and $B$ is an external magnetic field interacting with the magnetic moment set up by the spins. The symbol $<kl>$ indicates that we sum over nearest neighbors only.

# Ising Model

Notice that for $J > 0$ it is energetically favorable for neighboring spins to be aligned. This feature leads to, at low enough temperatures, to a cooperative phenomenon called spontaneous magnetization. That is, through interactions between nearest neighbors, a given magnetic moment can influence the alignment of spins that are separated from the given spin by a macroscopic distance. These long range correlations between spins are associated with a long-range order in which the lattice has a net magnetization in the absence of a magnetic field. This phase is normally called the ferromagnetic phase. With $J < 0$, we have a so-called antiferromagnetic case. At a critical temperature we have a phase transition to a disordered phase, a so-called paramagnetic phase.

With two spins, since each spin takes two values only, it means that in total we have $2^2 = 4$ possible arrangements of the two spins. These four possibilities are

$$1 = \uparrow\uparrow \qquad 2 = \uparrow\downarrow \qquad 3 = \downarrow\uparrow \qquad 4 = \downarrow\downarrow$$

What is the energy of each of these configurations?

For small systems, the way we treat the ends matters. In the first case we employ what is called free ends. For the one-dimensional case, the energy is then written as a sum over a single index

$$E_i = -J \sum_{j=1}^{N-1} s_j s_{j+1},$$

If we label the first spin as $s_1$ and the second as $s_2$ we obtain the following expression for the energy

$$E = -J s_1 s_2.$$

## Treatment of Boundaries

The calculation of the energy for the one-dimensional lattice with free ends for one specific spin-configuration can easily be implemented in the following lines

```
for ( j=1; j < N; j++) {
    energy += spin[j]*spin[j+1];
}
```

where the vector *spin*[] contains the spin value $s_k = \pm 1$. For the specific state $E_1$, we have chosen all spins up. The energy of this configuration becomes then

$$E_1 = E_{\uparrow\uparrow} = -J.$$

The other configurations give

$$E_2 = E_{\uparrow\downarrow} = +J,$$
$$E_3 = E_{\downarrow\uparrow} = +J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -J.$$

We can also choose so-called periodic boundary conditions. This means that if $i = N$, we set the spin number to $i = 1$. In this case the energy for the one-dimensional lattice reads

$$E_i = -J \sum_{j=1}^{N} s_j s_{j+1},$$

and we obtain the following expression for the two-spin case

$$E = -J(s_1 s_2 + s_2 s_1).$$

If we choose to use periodic boundary conditions we can code the above expression as

```
jm=N;
for ( j=1; j <=N ; j++) {
    energy += spin[j]*spin[jm];
    jm = j ;
}
```

| State | Energy (FE) | Energy (PBC) | Magnetization |
|-------|-------------|--------------|---------------|
| 1 $=\uparrow\uparrow$ | $-J$ | $-2J$ | 2 |
| 2 $=\uparrow\downarrow$ | $J$ | $2J$ | 0 |
| 3 $=\downarrow\uparrow$ | $J$ | $2J$ | 0 |
| 4 $=\downarrow\downarrow$ | $-J$ | $-2J$ | -2 |

| Number spins up | Degeneracy | Energy (FE) | Energy (PBC) | Magnetization |
|-----------------|-----------|-------------|--------------|---------------|
| 2 | 1 | $-J$ | $-2J$ | 2 |
| 1 | 2 | $J$ | $2J$ | 0 |
| 0 | 1 | $-J$ | $-2J$ | -2 |

It is worth noting that for small dimensions of the lattice, the energy differs depending on whether we use periodic boundary conditions or free ends. This means also that the partition functions will be different, as discussed below. In the thermodynamic limit however, $N \rightarrow \infty$, the final results do not depend on the kind of boundary conditions we choose. The magnetization is however the same, defined as

$$\mathcal{M}_i = \sum_{j=1}^{N} s_j,$$

where we sum over all spins for a given configuration $i$.

In a similar way, we could enumerate the number of states for a two-dimensional system consisting of two spins, i.e., a $2 \times 2$ Ising model on a square lattice with *periodic boundary conditions*. In this case we have a total of $2^4 = 16$ states. Some examples of configurations with their respective energies are listed here

$$E = -8J \qquad \begin{matrix} \uparrow & \uparrow \\ \uparrow & \uparrow \end{matrix} \qquad E = 0 \qquad \begin{matrix} \uparrow & \uparrow \\ \uparrow & \downarrow \end{matrix} \qquad E = 0 \qquad \begin{matrix} \downarrow & \downarrow \\ \uparrow & \downarrow \end{matrix}$$

We can group these configurations according to their total energy and magnetization.

| Number spins up | Degeneracy | Energy | Magnetization |
|---|---|---|---|
| 4 | 1 | $-8J$ | 4 |
| 3 | 4 | 0 | 2 |
| 2 | 4 | 0 | 0 |
| 2 | 2 | $8J$ | 0 |
| 1 | 4 | 0 | -2 |
| 0 | 1 | $-8J$ | -4 |

### Ising model, phase transitions and project 5

- Monday: Repetition from last week
- Discussion of the code for project 5, chapter 10.4
- Discussion of project 5 and chapters 10.5 and 10.6. Scaling laws.
- Wednesday:
- Discussion of results and the time-correlation function, parts of chapters 10.5 and 10.6
- Parallelization of codes, chapter 7.7 of lecture notes.

## Modelling the Ising Model

The code uses periodic boundary conditions with energy

$$E_i = -J \sum_{j=1}^{N} s_j s_{j+1},$$

In our case we have as the Monte Carlo sampling function the probability for finding the system in a state $s$ given by

$$P_s = \frac{e^{-(\beta E_s)}}{Z},$$

with energy $E_s$, $\beta = 1/kT$ and $Z$ is a normalization constant which defines the partition function in the canonical ensemble

$$Z(\beta) = \sum_s e^{-(\beta E_s)}$$

This is difficult to compute since we need all states. In a calculation of the Ising model in two dimensions, the number of configurations is given by $2^N$ with $N = L \times L$ the number of spins for a lattice of length $L$. Fortunately, the Metropolis algorithm considers only ratios between probabilities and we do not need to compute the partition function at all.

# Metropolis Algorithm

1. Establish an initial state with energy $E_b$ by positioning yourself at a random position in the lattice

2. Change the initial configuration by flipping e.g., one spin only. Compute the energy of this trial state $E_t$.

3. Calculate $\Delta E = E_t - E_b$. The number of values $\Delta E$ is limited to five for the Ising model in two dimensions, see the discussion below.

4. If $\Delta E \leq 0$ we accept the new configuration, meaning that the energy is lowered and we are hopefully moving towards the energy minimum at a given temperature. Go to step 7.

5. If $\Delta E > 0$, calculate $w = e^{-(\beta \Delta E)}$.

6. Compare $w$ with a random number $r$. If

$$r \leq w,$$

then accept the new configuration, else we keep the old configuration and its values.

7. The next step is to update various expectations values.

8. The steps (2)-(7) are then repeated in order to obtain a sufficently good representation of states.

In the calculation of the energy difference from one spin configuration to the other, we will limit the change to the flipping of one spin only. For the Ising model in two dimensions it means that there will only be a limited set of values for $\Delta E$. Actually, there are only five possible values. To see this, select first a random spin position $x, y$ and assume that this spin and its nearest neighbors are all pointing up. The energy for this configuration is $E = -4J$. Now we flip this spin as shown below. The energy of the new configuration is $E = 4J$, yielding $\Delta E = 8J$.

$$E = -4J \qquad\qquad \begin{array}{c} \uparrow \\ \uparrow \;\; \uparrow \;\; \uparrow \\ \uparrow \end{array} \qquad \Longrightarrow \qquad E = 4J \qquad\qquad \begin{array}{c} \uparrow \\ \uparrow \;\; \downarrow \;\; \uparrow \\ \uparrow \end{array}$$

The four other possibilities are as follows

$$E = -2J \qquad \downarrow \quad \overset{\uparrow}{\underset{\uparrow}{\uparrow}} \quad \uparrow \qquad \Longrightarrow \qquad E = 2J \qquad \downarrow \quad \overset{\uparrow}{\underset{\uparrow}{\downarrow}} \quad \uparrow$$

with $\Delta E = 4J$,

$$E = 0 \qquad \downarrow \quad \overset{\uparrow}{\underset{\downarrow}{\uparrow}} \quad \uparrow \qquad \Longrightarrow \qquad E = 0 \qquad \downarrow \quad \overset{\uparrow}{\underset{\downarrow}{\downarrow}} \quad \uparrow$$

with $\Delta E = 0$

$$E = 2J \qquad \downarrow \quad \begin{matrix} \downarrow \\ \uparrow \\ \downarrow \end{matrix} \quad \uparrow \qquad \Longrightarrow \qquad E = -2J \qquad \downarrow \quad \begin{matrix} \downarrow \\ \downarrow \\ \downarrow \end{matrix} \quad \uparrow$$

with $\Delta E = -4J$ and finally

$$E = 4J \qquad \downarrow \quad \begin{matrix} \downarrow \\ \uparrow \\ \downarrow \end{matrix} \quad \downarrow \qquad \Longrightarrow \qquad E = -4J \qquad \downarrow \quad \begin{matrix} \downarrow \\ \downarrow \\ \downarrow \end{matrix} \quad \downarrow$$

with $\Delta E = -8J$. This means in turn that we could construct an array which contains all values of $e^{\beta \Delta E}$ before doing the Metropolis sampling. Else, we would have to evaluate the exponential at each Monte Carlo sampling.

```
for ( double temp = initial_temp; temp <= final_temp; temp+=temp_ste
  //    initialise energy and magnetization
  E = M = 0.;
  // setup array for possible energy changes
  for( int de =-8; de <= 8; de++) w[de+8] = 0;
  for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temp);
  // initialise array for expectation values
  for( int i = 0; i < 5; i++) average[i] = 0.;
  initialize(n_spins, temp, spin_matrix, E, M);
  // start Monte Carlo computation
  for (int cycles = 1; cycles <= mcs; cycles++){
    Metropolis(n_spins, idum, spin_matrix, E, M, w);
    // update expectation values
    average[0] += E;     average[1] += E*E;
    average[2] += M;     average[3] += M*M; average[4] += fabs(M);
  }
  // print results
  output(n_spins, mcs, temp, average);
}
```

## The Initialise function

```
void initialize(int n_spins, double temp, int **spin_matrix,
double& E, double& M)
{
  // setup spin matrix and intial magnetization
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      spin_matrix[y][x] = 1; // spin orientation for the ground state
      M +=  (double) spin_matrix[y][x];
    }
  }
  // setup initial energy
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      E -=  (double) spin_matrix[y][x]*
(spin_matrix[periodic(y,n_spins,-1)][x] +
 spin_matrix[y][periodic(x,n_spins,-1)]);
    }
  }
}// end function initialise
```

# The periodic function

A compact way of dealing with periodic boundary conditions is given as follows:

```
// inline function for periodic boundary conditions
inline int periodic(int i, int limit, int add) {
  return (i+limit+add) % (limit);
```

with the following example from the function initialise

```
      E -=  (double) spin_matrix[y][x]*
(spin_matrix[periodic(y,n_spins,-1)][x] +
 spin_matrix[y][periodic(x,n_spins,-1)]);
```

# Alternative way for periodic boundary conditions

A more pedagogical way is given by the (here Fortran as example) program

```fortran
DO y = 1,lattice_y
    DO x = 1,lattice_x
        right = x+1 ; IF(x == lattice_x ) right = 1
        left = x-1 ; IF(x == 1 ) left = lattice_x
        up = y+1 ; IF(y == lattice_y ) up = 1
        down = y-1 ; IF(y == 1 ) down = lattice_y
        energy=energy - spin_matrix(x,y)*(spin_matrix(right,y)+&
            spin_matrix(left,y)+spin_matrix(x,up)+ &
            spin_matrix(x,down) )
        magnetization = magnetization + spin_matrix(x,y)
    ENDDO
ENDDO
energy = energy*0.5
```

## Computing $\Delta E$ and $\Delta M$

The energy difference between a state $E_1$ and a state $E_2$ with zero magnetic field is

$$\Delta E = E_2 - E_1 = J \sum_{<kl>}^{N} s_k^1 s_l^1 - J \sum_{<kl>}^{N} s_k^2 s_l^2,$$

which we can rewrite as

$$\Delta E = -J \sum_{<kl>}^{N} s_k^2 (s_l^2 - s_l^1),$$

where the sum now runs only over the nearest neighbors $k$ of the spin. Since the spin to be flipped takes only two values, $s_l^1 = \pm 1$ and $s_l^2 = \pm 1$, it means that if $s_l^1 = 1$, then $s_l^2 = -1$ and if $s_l^1 = -1$, then $s_l^2 = 1$. The other spins keep their values, meaning that $s_k^1 = s_k^2$. If $s_l^1 = 1$ we must have $s_l^1 - s_l^2 = 2$, and if $s_l^1 = -1$ we must have $s_l^1 - s_l^2 = -2$. From these results we see that the energy difference can be coded efficiently as

$$\Delta E = 2 J s_l^1 \sum_{<k>}^{N} s_k,$$

where the sum runs only over the nearest neighbors $k$ of spin $l$
The difference in magnetisation is given by the difference $s_l^1 - s_l^2 = \pm 2$, or in a more compact way as

$$M_2 = M_1 + 2 s_l^2,$$

where $M_1$ and $M_2$ are the magnetizations before and after the spin flip, respectively.

## The Metropolis function

```
  // loop over all spins
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
       int ix = (int) (ran1(&idum)*(double)n_spins);   // RANDOM SPIN
       int iy = (int) (ran1(&idum)*(double)n_spins);  // RANDOM SPIN
       int deltaE =  2*spin_matrix[iy][ix]*
(spin_matrix[iy][periodic(ix,n_spins,-1)]+
 spin_matrix[periodic(iy,n_spins,-1)][ix] +
 spin_matrix[iy][periodic(ix,n_spins,1)] +
 spin_matrix[periodic(iy,n_spins,1)][ix]);
       if ( ran1(&idum) <= w[deltaE+8] ) {
spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin confi
         M += (double) 2*spin_matrix[iy][ix];
         E += (double) deltaE;
       }
     }
   }
```

# Expectation Values

```
double norm = 1/((double) (mcs));// divided by total number of cycle
double Eaverage = average[0]*norm;
double E2average = average[1]*norm;
double Maverage = average[2]*norm;
double M2average = average[3]*norm;
double Mabsaverage = average[4]*norm;
// all expectation values are per spin, divide by 1/n_spins/n_spins
double Evariance = (E2average- Eaverage*Eaverage)/n_spins/n_spins;
double Mvariance = (M2average - Mabsaverage*Mabsaverage)/n_spins/n_s
ofile << setiosflags(ios::showpoint | ios::uppercase);
ofile << setw(15) << setprecision(8) << temp;
ofile << setw(15) << setprecision(8) << Eaverage/n_spins/n_spins;
ofile << setw(15) << setprecision(8) << Evariance/temp/temp;
ofile << setw(15) << setprecision(8) << Maverage/n_spins/n_spins;
ofile << setw(15) << setprecision(8) << Mvariance/temp;
ofile << setw(15) << setprecision(8) << Mabsaverage/n_spins/n_spins
```

For the one-dimensional Ising model we can compute rather easily the exact partition function for a system of $N$ spins. Let us consider first the case with free ends. The energy reads

$$E = -J \sum_{j=1}^{N-1} s_j s_{j+1}.$$

The partition function for $N$ spins is given by

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp\left(\beta J \sum_{j=1}^{N-1} s_j s_{j+1}\right), \tag{200}$$

and since the last spin occurs only once in the last sum in the exponential, we can single out the last spin as follows

$$\sum_{s_N=\pm 1} \exp\left(\beta J s_{N-1} s_N\right) = 2 cosh(\beta J). \tag{201}$$

The partition function consists then of a part from the last spin and one from the remaining spins resulting in

$$Z_N = Z_{N-1} 2 cosh(\beta J). \tag{202}$$

We can repeat this process and obtain

$$Z_N = (2cosh(\beta J))^{N-2} Z_2, \tag{203}$$

with $Z_2$ given by

$$Z_2 = \sum_{s_1 = \pm 1} \sum_{s_2 = \pm 1} \exp(\beta J s_1 s_2) = 4cosh(\beta J), \tag{204}$$

resulting in

$$Z_N = 2(2cosh(\beta J))^{N-1}. \tag{205}$$

In the thermodynamical limit where we let $N \to \infty$, the way we treat the ends does not matter. However, since our computations will always be carried out with a limited value of $N$, we need to consider other boundary conditions as well. Here we limit the attention to periodic boundary conditions.

We can then calculate the mean energy with free ends from the above formula for the partition function using

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta} = -(N-1)J tanh(\beta J). \tag{206}$$

Helmholtz's free energy is given by

$$F = -k_B T lnZ_N = -Nk_B T ln\left(2cosh(\beta J)\right). \tag{207}$$

The specific heat in one-dimension with free ends is

$$C_V = \frac{1}{kT^2} \frac{\partial^2}{\partial \beta^2} lnZ_N = (N-1)k \left(\frac{\beta J}{cosh(\beta J)}\right)^2. \tag{208}$$

Note well that this expression for the specific heat from the one-dimensional Ising model does not diverge, thus we do not have a second order phase transition.

If we use periodic boundary conditions, the partition function is given by

$$Z_N = \sum_{s_1 = \pm 1} \cdots \sum_{s_N = \pm 1} \exp\left(\beta J \sum_{j=1}^{N} s_j s_{j+1}\right), \tag{209}$$

where the sum in the exponential runs from 1 to $N$ since the energy is defined as

$$E = -J \sum_{j=1}^{N} s_j s_{j+1}.$$

We can then rewrite the partition function as

$$Z_N = \sum_{\{s_i = \pm 1\}} \prod_{i=1}^{N} \exp\left(\beta J s_i s_{i+1}\right), \tag{210}$$

where the first sum is meant to represent all lattice sites. Introducing the matrix $\hat{\mathbf{T}}$ (the so-called transfer matrix)

$$\hat{\mathbf{T}} = \begin{pmatrix} e^{\beta J} & e^{-\beta J} \\ e^{-\beta J} & e^{\beta J} \end{pmatrix}, \tag{211}$$

$$Z_N = \sum_{\{s_i = \pm 1\}} \hat{\mathbf{T}}_{s_1 s_2} \hat{\mathbf{T}}_{s_2 s_3} \ldots \hat{\mathbf{T}}_{s_N s_1} = Tr \hat{\mathbf{T}}^N. \tag{212}$$

The $2 \times 2$ matrix $\hat{\mathbf{T}}$ is easily diagonalized with eigenvalues $\lambda_1 = 2cosh(\beta J)$ and $\lambda_2 = 2sinh(\beta J)$. Similarly, the matrix $\hat{\mathbf{T}}^N$ has eigenvalues $\lambda_1^N$ and $\lambda_2^N$ and the trace of $\hat{\mathbf{T}}^N$ is just the sum over eigenvalues resulting in a partition function

$$Z_N = \lambda_1^N + \lambda_2^N = 2^N \left( [cosh(\beta J)]^N + [sinh(\beta J)]^N \right). \tag{213}$$

Helmholtz's free energy is in this case

$$F = -k_B T ln(\lambda_1^N + \lambda_2^N) = -k_B T \left\{ N ln(\lambda_1) + ln \left( 1 + (\frac{\lambda_2}{\lambda_1})^N \right) \right\} \tag{214}$$

which in the limit $N \to \infty$ results in $F = -k_B T N ln(\lambda_1)$

Hitherto we have limited ourselves to studies of systems with zero external magnetic field, viz $B = 0$. We will mostly study systems which exhibit a spontaneous magnitization. It is however instructive to extend the one-dimensional Ising model to $B \neq 0$, yielding a partition function (with periodic boundary conditions)

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp\left(\beta \sum_{j=1}^{N}(Js_j s_{j+1} + \frac{B}{2}(s_i + s_{i+1}))\right), \tag{215}$$

which yields a new transfer matrix with matrix elements $t_{11} = e^{\beta(J+B)}$, $t_{1-1} = e^{-\beta J}$, $t_{-11} = e^{\beta J}$ and $t_{-1-1} = e^{\beta(J-B)}$ with eigenvalues

$$\lambda_1 = e^{\beta J} cosh(\beta J) + \left(e^{2\beta J} sinh^2(\beta B) + \exp -(\beta 2J)\right)^{1/2}, \tag{216}$$

and

$$\lambda_2 = e^{\beta J} cosh(\beta J) - \left(e^{2\beta J} sinh^2(\beta B) + \exp -(\beta 2J)\right)^{1/2}. \tag{217}$$

It is now useful to compute the expectation value of the magnetisation per spin

$$\langle \mathcal{M}/N \rangle = \frac{1}{NZ} \sum_{i}^{M} \mathcal{M}_i e^{-\beta E_i} = -\frac{1}{N} \frac{\partial F}{\partial B}, \tag{218}$$

resulting in

$$\langle \mathcal{M}/N \rangle = \frac{sinh(\beta B}{\left(sinh^2(\beta B) + \exp{-(\beta 2J)}\right)^{1/2}}. \tag{219}$$

We see that for $B = 0$ the magnetisation is zero. This means that for a one-dimensional Ising model we cannot have a spontaneous magnetization. And there is no second order phase transition as well.

In studies of phase transitions we are interested in minimizing the free energy by varying the average magnetization, which is the order parameter (disappears at $T_C$). In mean field theory the local magnetization is a treated as a constant, all effects from fluctuations are neglected. A way to achieve this is to rewrite by adding and subtracting the mean magnetization $\langle s \rangle$

$$s_i s_j = (s_i - \langle s \rangle + \langle s \rangle)(s_i - \langle s \rangle + \langle s \rangle) \approx \langle s \rangle^2 + \langle s \rangle(s_i - \langle s \rangle) + \langle s \rangle(s_j - \langle s \rangle), \quad (220)$$

where we have ignored terms of the order $(s_i - \langle s \rangle)(s_i - \langle s \rangle)$, which leads to correlations between neighbouring spins. In mean field theory we ignore correlations.

# Mean Field Theory and the Ising Model

This means that we can rewrite the Hamiltonian

$$E = -J \sum_{<ij>}^{N} s_k s_l - B \sum_{i}^{N} s_i, \qquad (221)$$

as

$$E = -J \sum_{<ij>} \langle s \rangle^2 + \langle s \rangle (s_i - \langle s \rangle) + \langle s \rangle (s_j - \langle s \rangle) - B \sum_{i} s_i, \qquad (222)$$

resulting in

$$E = -(B + zJ\langle s \rangle) \sum_{i} s_i + zJ\langle s \rangle^2, \qquad (223)$$

with $z$ the number of nearest neighbours for a given site $i$. We can define an effective field which all spins see, namely

$$B_{\mathrm{eff}} = (B + zJ\langle s \rangle). \qquad (224)$$

# Mean Field Theory and the Ising Model

How do we get $\langle s \rangle$)?
Here we use the canonical ensemble. The partition function reads in this case

$$Z = e^{-NzJ\langle s \rangle^2/kT} \left(2cosh(B_{\text{eff}}/kT)\right)^N, \tag{225}$$

with a free energy

$$F = -kTlnZ = -NkTln(2) + NzJ\langle s \rangle^2 - NkTln\left(cosh(B_{\text{eff}}/kT)\right) \tag{226}$$

and minimizing $F$ wrt $\langle s \rangle$ we arrive at

$$\langle s \rangle = tanh(2cosh\left(B_{\text{eff}}/kT\right)). \tag{227}$$

Close to the phase transition we expect $\langle s \rangle$ to become small and eventually vanish. We can then expand $F$ in powers of $\langle s \rangle$ as

$$F = -NkTln(2) + NzJ\langle s \rangle^2 - NkT - BN\langle s \rangle + NkT \left( \frac{1}{2}\langle s \rangle^2 + \frac{1}{12}\langle s \rangle^4 + \dots \right), \quad (228)$$

and using $\langle M \rangle = N\langle s \rangle$ we can rewrite as

$$F = F_0 - B\langle M \rangle + \frac{1}{2}a\langle M \rangle^2 + \frac{1}{4}b\langle M \rangle^4 + \dots \quad (229)$$

## Connecting to Landau Theory

Let $\langle M \rangle = m$ and

$$F = F_0 + \frac{1}{2}am^2 + \frac{1}{4}bm^4 + \frac{1}{6}cm^6 \qquad (230)$$

$F$ has a minimum at equilibrium $F'(m) = 0$ and $F''(m) > 0$

$$F'(m) = 0 = m(a + bm^2 + cm^4),$$

and if we assume that $m$ is real we have two solutions

$$m = 0,$$

or

$$m^2 = \frac{b}{2c}\left(-1 \pm \sqrt{1 - 4ac/b^2}\right)$$

Can describe both first and second-order phase transitions. Here we consider the second case. Assume $b > 0$ and $a \ll 1$ small since we want to study a perturbation around $m = 0$. We reach the critical point when $a = 0$.

$$m^2 = \frac{b}{2c} \left( (-1 \pm \sqrt{1 - 4ac/b^2} \right) \approx -a/b$$

Assume that

$$a(T) = \alpha(T - T_C),$$

with $\alpha > 0$ and $T_C$ the critical temperature where the magnetization vanishes. If $a$ is negative we have two solutions

$$m = \pm\sqrt{-a/b} = \pm\sqrt{\frac{\alpha(T_C - T)}{b}}$$

$m$ evolves continuously to the critical temperature where $F = 0$ for $T \leq T_C$ (see separate graph).

## Entropy and Specific Heat

We can now compute the entropy

$$S = -\left(\frac{\partial F}{\partial T}\right)$$

For $T \geq T_C$ we have $m = 0$ and

$$S = -\left(\frac{\partial F_0}{\partial T}\right)$$

and for $T \leq T_C$

$$S = -\left(\frac{\partial F_0}{\partial T}\right) - \alpha^2(T_C - T)/2b,$$

and we see that there is a smooth crossover at $T_C$.

We can now compute the specific heat

$$C_V = T\left(\frac{\partial S}{\partial T}\right)$$

and $T_C$ we get a discontinuity of

$$\Delta C_V = -\alpha^2/2b,$$

signalling a second-order phase transition. Landau theory gives irrespective of dimension critical exponents

$$m \sim (T_C - T)^\beta,$$

and

$$C_V \sim (T_C - T)^\alpha,$$

with $\beta = 1/2$ and $\alpha = 1$. It predicts a phase transition for one dimension as well. For the Ising model there is no phase transition for $d = 1$. In two dimensions we have $\beta = 1/8$ and $\alpha = 0$.

Another quantity (given by the covariance) is the correlation function

$$G_{ij} = \langle S_i S_j \rangle - \langle S_i \rangle \langle S_j \rangle. \tag{231}$$

and the correlation length

$$\xi^{-1} = -\lim_{r \to \infty} \frac{\partial}{\partial r} ln G(r), \tag{232}$$

with $r = |i - j|$.

Near $T_C$ we can characterize the behavior of many physical quantities by a power law behavior. As an example, the mean magnetization is given by

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta}, \tag{233}$$

where $\beta$ is a so-called critical exponent. A similar relation applies to the heat capacity

$$C_V(T) \sim |T_C - T|^{-\alpha}, \tag{234}$$

the susceptibility

$$\chi(T) \sim |T_C - T|^{\gamma}. \tag{235}$$

and the correlation length

$$\xi(T) \sim |T_C - T|^{-\nu}. \tag{236}$$

$\alpha = 0$, $\beta = 1/8$, $\gamma = 7/4$ and $\nu = 1$. Below we will derive these coefficients from finite size scaling theories.

Through finite size scaling relations it is possible to relate the behavior at finite lattices with the results for an infinitely large lattice. The critical temperature scales then as

$$T_C(L) - T_C(L = \infty) \sim aL^{-1/\nu}, \tag{237}$$

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta} \rightarrow L^{-\beta/\nu}, \tag{238}$$

$$C_V(T) \sim |T_C - T|^{-\gamma} \rightarrow L^{\gamma/\nu}, \tag{239}$$

and

$$\chi(T) \sim |T_C - T|^{-\alpha} \rightarrow L^{\alpha/\nu}. \tag{240}$$

We can compute the slope of the curves for $M$, $C_V$ and $\chi$ as function of lattice sites and extract the exponent $\nu$.

The analytic expression for the Ising model in two dimensions was obtained in 1944 by the Norwegian chemist Lars Onsager (Nobel prize in chemistry). The exact partition function for $N$ spins in two dimensions and with zero magnetic field $\mathcal{H}$ is given by

$$Z_N = \left[2\cosh(\beta J)e^I\right]^N, \tag{241}$$

with

$$I = \frac{1}{2\pi}\int_0^\pi d\phi \ln\left[\frac{1}{2}\left(1 + (1 - \kappa^2 \sin^2\phi)^{1/2}\right)\right], \tag{242}$$

and

$$\kappa = 2\sinh(2\beta J)/\cosh^2(2\beta J). \tag{243}$$

# Analytic Results: two-dimensional Ising model

The resulting energy is given by

$$\langle E \rangle = -Jcoth(2\beta J)\left[1 + \frac{2}{\pi}(2tanh^2(2\beta J) - 1)K_1(q)\right], \tag{244}$$

with $q = 2sinh(2\beta J)/cosh^2(2\beta J)$ and the complete elliptic integral of the first kind

$$K_1(q) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - q^2 sin^2\phi}}. \tag{245}$$

Differentiating once more with respect to temperature we obtain the specific heat given by

$$C_V = \frac{4k_B}{\pi}(\beta J coth(2\beta J))^2$$

$$\left\{ K_1(q) - K_2(q) - (1 - tanh^2(2\beta J)) \left[ \frac{\pi}{2} + (2tanh^2(2\beta J) - 1)K_1(q) \right] \right\},$$

with

$$K_2(q) = \int_0^{\pi/2} d\phi \sqrt{1 - q^2 sin^2\phi}. \tag{246}$$

is the complete elliptic integral of the second kind. Near the critical temperature $T_C$ the specific heat behaves as

$$C_V \approx -\frac{2}{k_B\pi} \left( \frac{J}{T_C} \right)^2 ln \left| 1 - \frac{T}{T_C} \right| + \text{const.} \tag{247}$$

In theories of critical phenomena one has that

$$C_V \sim \left| 1 - \frac{T}{T_C} \right|^{-\alpha}, \tag{248}$$

and Onsager's result is a special case of this power law behavior. The limiting form of the function

$$lim_{\alpha \to 0} \frac{1}{\alpha} (Y^{-\alpha} - 1) = -lnY, \tag{249}$$

meaning that the analytic result is a special case of the power law singularity with $\alpha = 0$.

One can also show that the mean magnetization per spin is

$$\left[ 1 - \frac{(1 - tanh^2(\beta J))^4}{16 tanh^4(\beta J)} \right]^{1/8}$$

for $T < T_C$ and 0 for $T > T_C$. The behavior is thus as $T \to T_C$ from below

$$M(T) \sim (T_C - T)^{1/8}$$

The susceptibility behaves as

$$\chi(T) \sim |T_C - T|^{-7/4}$$

In project 5 it will be useful to parallelize the code. An example of code for the Ising model can be found under project 5. **Task parallelism**: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation or integrations are examples of this. It is almost embarrassingly trivial to parallelize Monte Carlo codes. MPI is a message-passing library where all the routines have corresponding C/C++-binding

```
MPI_Command_name
```

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

```
MPI_COMMAND_NAME
```

# What is Message Passing Interface (MPI)? Yet another library!

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library.

MPI is a specification, not a particular implementation. MPI programs should be able to run on all possible machines and run all MPI implementetations without change.

An MPI computation is a collection of processes communicating with messages.

See chapter 7.7 of lecture notes for more details.

## MPI

MPI is a library specification for the message passing interface, proposed as a standard.

- independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use. Designed for high performance. Insert communication and synchronization functions where necessary.

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.

- **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 used for the distributed memory systems, but can also be used on shared memory systems.

In these lectures we consider only message-passing for writing parallel programs.

# Overhead present in parallel computing

- **Uneven load balance**: not all the processors can perform useful work at all time.
- **Overhead of synchronization.**
- **Overhead of communication**.
- Extra computation due to parallelization.

Due to the above overhead and that certain part of a sequential algorithm cannot be parallelized we may not achieve an optimal parallelization.

## Parallelizing a sequential algorithm

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This is the difficult part,
- Distribute the global work and data among $P$ processors.

## Process and processor

- We refer to process as a logical unit which executes its own code, in an MIMD style.
- The processor is a physical device on which one or several processes are executed.
- The MPI standard uses the concept process consistently throughout its documentation.
- 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 of chapter 7.

```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

```fortran
PROGRAM hello
INCLUDE "mpif.h"
INTEGER:: size, my_rank, ierr

CALL  MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
WRITE(*,*)"Hello world, I've rank ",my_rank," out o
CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

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 7.7 of lecture notes.

```
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 ();
```

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

```cpp
.....
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 datatype,
             int dest, int tag, MPI_Comm comm)}
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable **buf** is the variable we wish to send while **count** is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

## Note 4

Once you have sent a message, you must receive it on another task. The function **MPI_RECV** is similar to the send call.

```
int MPI_Recv( void *buf, int count, MPI_Datatype da
          int source,
          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.
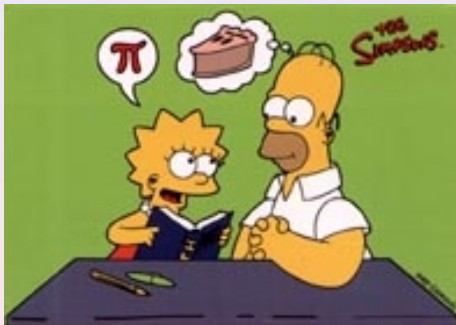
# Integrating $\pi$



### Examples

- Go to the webpage and click on the programs link
- Go to MPI and then chapter 7
- Look at program5.ccp and program6.cpp. (Fortran version also available).
- These codes compute $\pi$ using the rectangular and trapezoidal rules.

## Integration algos

The trapezoidal rule (example6.cpp)

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right).$$

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length $h$ and heigh given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x_{i-1/2})$ is the midpoint value of $f$ for a given rectangle. This is used in example5.cpp.

# Dissection of example 5

```
1   //    Reactangle rule and numerical integration
2   using namespace std;
3   #include <mpi.h>
4   #include <iostream>

5   int main (int nargs, char* args[])
6   {
7       int numprocs, my_rank, i, n = 1000;
8       double local_sum, rectangle_sum, x, h;
9       //   MPI initializations
10      MPI_Init (&nargs, &args);
11      MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
12      MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

After the standard initializations with MPI such as MPI_Init, MPI_Comm_size and MPI_Comm_rank, MPI_COMM_WORLD contains now the number of processes defined by using for example

```
mpiexec -np 10 ./prog.x
```

In line 4 we check if we have read in from screen the number of mesh points $n$. Note that in line 7 we fix $n = 1000$, however we have the possibility to run the code with a different number of mesh points as well. If my_rank equals zero, which correponds to the master node, then we read a new value of $n$ if the number of arguments is larger than two. This can be done as follows when we run the code

```
mpiexec -np 10 ./prog.x  10000
```

## Dissection of example 5

```
13      //   Read from screen a possible new vaue of n
14      if (my_rank == 0 && nargs > 1) {
15         n = atoi(args[1]);
16      }
17      h = 1.0/n;
18      //  Broadcast n and h to all processes
19      MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
20      MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
21      // Every process sets up its contribution to the integral
22      local_sum = 0.;
23      for (i = my_rank; i < n; i += numprocs) {
24         x = (i+0.5)*h;
25         local_sum += 4.0/(1.0+x*x);
26      }
27      local_sum *= h;
```

In line 17 we define also the step length *h*. In lines 19 and 20 we use the broadcast function MPI_Bcast. We use this particular function because we want data on one processor (our master node) to be shared with all other processors. The broadcast function sends data to a group of processes.

## Dissection of example 5

The MPI routine MPI_Bcast transfers data from one task to a group of others. The format for the call is in C++ given by the parameters of

```
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);.
MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

in a case of a double. The general structure of this function is

```
MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_
```

All processes call this function, both the process sending the data (with rank zero) and all the other processes in MPI_COMM_WORLD. Every process has now copies of *n* and *h*, the number of mesh points and the step length, respectively.

We transfer the addresses of *n* and *h*. The second argument represents the number of data sent. In case of a one-dimensional array, one needs to transfer the number of array elements. If you have an $n \times m$ matrix, you must transfer $n \times m$. We need also to specify whether the variable type we transfer is a non-numerical such as a logical or character variable or numerical of the integer, real or complex type.

## Dissection of example 5

```
28    if (my_rank == 0) {
29      MPI_Status status;
30      rectangle_sum = local_sum;
31      for (i=1; i < numprocs; i++) {
32        MPI_Recv(&local_sum,1,MPI_DOUBLE,MPI_ANY_SOURCE,500,
                     MPI_COMM_WORLD,&status);
33        rectangle_sum += local_sum;
34      }
35      cout << "Result: " << rectangle_sum  << endl;
36    }  else
37      MPI_Send(&local_sum,1,MPI_DOUBLE,0,500,MPI_COMM_WORLD);
38    // End MPI
39    MPI_Finalize ();
40    return 0;
41  }
```

In lines 23-27, every process sums its own part of the final sum used by the rectangle rule. The receive statement collects the sums from all other processes in case my_rank == 0, else an MPI send is performed. If we are not the master node, we send the results, else they are received and the local results are added to final sum. The above can be rewritten using the MPI_allreduce, as discussed in the next example.

The above function is not very elegant. Furthermore, the MPI instructions can be simplified by using the functions MPI_Reduce or MPI_Allreduce. The first function takes information from all processes and sends the result of the MPI operation to one process only, typically the master node. If we use MPI_Allreduce, the result is sent back to all processes, a feature which is useful when all nodes need the value of a joint operation. We limit ourselves to MPI_Reduce since it is only one process which will print out the final number of our calculation, The arguments to MPI_Allreduce are the same.

## MPI_reduce

Call as

```
MPI_reduce( void *senddata, void* resultdata, int count,
     MPI_Datatype datatype, MPI_Op, int root, MPI_Comm comm)
```

The two variables *senddata* and *resultdata* are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable *count* represents the total dimensionality, 1 in case of just one variable, while MPI_Datatype defines the type of variable which is sent and received.

The new feature is MPI_Op. It defines the type of operation we want to do. In our case, since we are summing the rectangle contributions from every process we define MPI_Op = MPI_SUM. If we have an array or matrix we can search for the largest og smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINOC. If we want the product we write MPI_PROD.

MPI_Allreduce is defined as

```
MPI_Alreduce( void *senddata, void* resultdata, int count,
          MPI_Datatype datatype, MPI_Op, MPI_Comm comm)}.
```

## Dissection of example 6

```cpp
//    Trapezoidal rule and numerical integration usign MPI, example 6
using namespace std;
#include <mpi.h>
#include <iostream>

//     Here we define various functions called by the main program

double int_function(double );
double trapezoidal_rule(double , double , int , double (*)(double));

//   Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum, local_sum;
  double  time_start, time_end, total_time;
```

```
//  MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
time_start = MPI_Wtime();
//  Fixed values for a, b and n
a = 0.0 ; b = 1.0;  n = 1000;
h = (b-a)/n;     // h is the same for all processes
local_n = n/numprocs;
// make sure n > numprocs, else integer division gives zero
// Length of each process' interval of
// integration = local_n*h.
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
```

## Dissection of example 6

```
total_sum = 0.0;
local_sum = trapezoidal_rule(local_a, local_b, local_n,
                             &int_function);
MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
time_end = MPI_Wtime();
total_time = time_end-time_start;
if ( my_rank == 0) {
  cout << "Trapezoidal rule = " <<  total_sum << endl;
  cout << "Time = " <<  total_time
       << " on number of processors: "  << numprocs  << endl;
}
// End MPI
MPI_Finalize ();
return 0;
}  // end of main program
```

We use MPI_reduce to collect data from each process. Note also the use of the
function MPI_Wtime.

```
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate
```

# Dissection of example 6

Implementation of the trapezoidal rule.

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

- Develop codes locally, run with some few processes and test your codes. Do benchmarking, timing and so forth on local nodes, for example your laptop. You can install MPICH2 on your laptop (most new laptos come with dual cores). You can test with one node at the lab.

- When you are convinced that your codes run correctly, you start your production runs on available supercomputers, in our case titan.uio.no.

## hpc@usit.uio.no

- Computational Physics requires High Performance Computing (HPC) resources
- USIT and the Research Computing Services (RCS) provides HPC resources and HPC support
- Resources: `titan.uio.no`
- Support: 14 people
- Contact: `hpc@usit.uio.no`

# Titan

## Hardware

- 546 X2200m2, 7 X4200, Magnum 3456 IB switch
- 174 Dell 1425, + 20 HP DL385 servers
- EVA8K 120 TB storage
- Dual core Intel and AMD ($\sim$ 2500 cores in total)
- Quad-core intel
- Infiniband and ethernet
- From 4GB to 64GB RAM
- Heterogenous cluster!

# Titan

## Software

- Batch system: Sun Grid Engine (SGE)
- Message Passing Interface (MPI):
  - Scampi
  - MPICH2
  - OpenMPI
- Compilers: GCC, Intel, Portland and Pathscale
- Optimized math libraries and scientific applications
- All you need may be found under `/site`
- Available software: `http://www.hpc.uio.no/index.php/Titan_software`

# Getting started

## Batch systems

- A batch system controls the use of the cluster resources
- Submits the job to the right resource
- Monitors the job while executing
- Restarts the job in case of failure
- Takes care of priorities and queues to control execution order of unrelated jobs

## Sun Grid Engine

- SGE is the batch system used on Titan
- Jobs are executed either interactively or through job scripts
- Useful commands: `showq`, `qlogin`, `qsub`
- http: //www.hpc.uio.no/index.php/FAQ#QUEUE_SYSTEM

# Getting started

## Modules

- Different compilers, MPI-versions and applications need different sets of user environment variables
- The `modules` package lets you load and remove the different variable sets
- Useful commands:
    - List available modules: `module avail`
    - Load module: `module load <environment>`
    - Unload module: `module unload <environment>`
    - Currently loaded: `module list`
- `http://www.hpc.uio.no/index.php/FAQ#MODULES`

## Example

### Interactively

```
# login to titan
$ ssh titan.uio.no
# ask for 4 cpus
$ qlogin -P fys3150 -pe mpi 4
# we want to use the scampi module
$ module load scampi
$ mkdir -p fys3150/mpiexample/
$ cd fys3150/mpiexample/
# download program5.cpp from the course pages:
$ wget http://tinyurl.com/39hrah/program5.cpp
# compile the program
$ mpic++ -O3 -o program5.x program5.cpp
# and execute it
$ mpiexec ./program5.x
Result: 3.14159
# finished, so logging out
$ exit
```

# The job script

## job.sge

```
#!/bin/sh
# 4 cpus with mpi (or other communication)
#$ -pe mpi 4
# 10 mins of walltime
#$ -l s_rt=0:10:0
# project fys3150
#$ -P fys3150
# we need 512 MB of memory per process
#$ -l s_vmem=512M
# name of job
#$ -N program5

source /site/bin/jobsetup

# load the module used when we compiled the program
module load scampi

# start program
/site/bin/mpiexec ./program5.x

#END OF SCRIPT
```

## Example

### Submitting

```
# login to titan
$ ssh titan.uio.no
# we want to use the module scampi
$ module load scampi
$ cd fys3150/mpiexample/
# compile the program
$ mpic++ -O3 -o program5.x program5.cpp
# and submit it
$ qsub job.sge
$ exit
```

## Example

### Checking execution

```
# check if job is running:
$ qstat -u mhjensen
job-ID  prior   name    user        state submit/start at    queue
                                slots ja-task-ID
----------------------------------------------------------------------------------------------
 379931 2.50073 rectangle  mhjensen      r   10/17/2008 06:02:39 normal@compute
      -13-6.local        4
# results are in rectangle.o379931
$ cat rectangle.o379931
Starting job 379931 on node compute-13-6.local at Thu Jan 17 06:02:39 CET 2008
Creating /work/379931.undefined.d
Result: 3.14159
Terminating job 379931 on compute-13-6.local at Thu Jan 17 06:02:44 CET 2008
Deleting /work/379931.undefined.d
```

# Tips and admonitions

## Tips

- Titan FAQ: `http://www.hpc.uio.no/index.php/FAQ`
- man-pages, e.g. `man qsub`
- Ask us

## Admonitions

- Remember to exit from qlogin-sessions; the resource is reserved for you untill you exit
- Don't run jobs on login-nodes; these are only for compiling and editing files

### Ising model, phase transitions, project 5 and quantum Monte Carlo

- Monday: Repetition from last week
- Discussion of project 5 and definition of correlation time and equilibration time (optional part of project 5).
- Discussion of Wolff's algorithm for calculations close to critical point (optional part of project 5).
- Wednesday:
- Quantum Monte Carlo, last part of lectures. Chapter 11.1-11.3.

Three small messages: 1) Remember to sign up for the exam. 2) Evaluation form. 3) Blitz actually works on Titan.

## Covariance

The mean value of a random variable $X$ with range $x_1, x_2, \ldots, N$ is

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

and the variance is

$$\langle \sigma^2 \rangle = \frac{1}{N} \sum_{i=1}^{N} (x_i - \langle x \rangle)^2 p(x_i) = \frac{1}{N} \sum_{i=1}^{N} \langle (x_i - \mu_i)^2 \rangle.$$

Assume now that we have two independent sets of measurements $X_1$ and $X_2$ with corresponding mean and variance $\mu_1$ and $\mu_2$ and $\langle \sigma^2 \rangle_{X_1}$ and $\langle \sigma^2 \rangle_{X_2}$.

## Covariance

It follows that if we define the new stochastic variable

$$Y = X_1 + X_2,$$

we have

$$\mu_Y = \mu_1 + \mu_2,$$

and

$$\langle \sigma^2 \rangle_Y = \langle (Y - \mu_Y)^2 \rangle = \langle (X_1 - \mu_1)^2 \rangle + \langle (X_2 - \mu_2)^2 \rangle + 2\langle (X_1 - \mu_1)(X_2 - \mu_2) \rangle.$$

It is useful to define the so-called covariance, given by

$$\langle cov(X_1, X_2) \rangle = \langle (X_1 - \mu_1)(X_2 - \mu_2) \rangle$$

where we consider the averages $\mu_1$ and $\mu_2$ as the outcome of two separate measurements. The covariance measures thus the degree of correlation between variables. We can then rewrite the variance of $Y$ as

$$\langle \sigma^2 \rangle_Y = \sum_{j=1}^{2} \langle (X_j - \mu_j)^2 \rangle + 2cov(X_1, X_2),$$

which in our notation becomes

$$\langle \sigma^2 \rangle_Y = \langle \sigma^2 \rangle_{X_1} + \langle \sigma^2 \rangle_{X_2} + 2cov(X_1, X_2).$$

If $X_1$ and $X_2$ are two independent variables we can show that the covariance is zero, but one cannot deduce from a zero covariance whether the variables are independent or not. If our random variables which we generate are truely random numbers, then the covariance should be zero.

A way to measure the correlation between two sets of stochastic variables is the so-called correlation function $\rho(X_1, X_2)$ defined as

$$\rho(X_1, X_2) = \frac{\langle cov(X_1, X_2) \rangle}{\sqrt{\langle \sigma^2 \rangle_{X_1} \langle \sigma^2 \rangle_{X_2}}}.$$

Obviously, if the covariance is zero due to the fact that the variables are independent, then the correlation is zero. This quantity is often called the correlation coefficient between $X_1$ and $X_2$. We can extend this analysis to a set of stochastic variables $Y = (X_1 + X_2 + \cdots + X_N)$. We now assume that we have $N$ different measurements of the mean and variance of a given variable. Each measurement consists again of $N$ measurements, although we could have chosen the latter to be different from $N$. As an example, every evening for $N$ days you measure $N$ throws of two dice. The mean and variance are defined as above. The total mean value is defined as

$$\langle \mu_Y \rangle = \sum_{i=1}^{N} \langle \mu_i \rangle.$$

The total variance is however now defined as

$$\langle \sigma^2 \rangle_Y = \langle (Y - \mu_Y)^2 \rangle = \sum_{j=1}^{N} \langle (X_j - \mu_j) \rangle^2 = \sum_{j=1}^{N} \langle \sigma^2 \rangle_{X_j} + 2 \sum_{j<k}^{N} \langle (X_j - \mu_j) \rangle \langle (X_k - \mu_k) \rangle,$$

or

$$\langle \sigma^2 \rangle_Y = \sum_{j=1}^{N} \langle \sigma^2 \rangle_{X_j} + 2 \sum_{j<k}^{N} cov(X_j, X_k).$$

If the variables are independent, the covariance is zero and the variance is reduced to

$$\langle \sigma^2 \rangle_Y = \sum_{j=1}^{N} \langle \sigma^2 \rangle_{X_j},$$

The so-called time-displacement autocorrelation $\phi(t)$ for the magnetization is given by

$$\phi(t) = \int dt' \left[ \mathcal{M}(t') - \langle \mathcal{M} \rangle \right] \left[ \mathcal{M}(t' + t) - \langle \mathcal{M} \rangle \right],$$

which can be rewritten as

$$\phi(t) = \int dt' \left[ \mathcal{M}(t') \mathcal{M}(t' + t) - \langle \mathcal{M} \rangle^2 \right],$$

where $\langle \mathcal{M} \rangle$ is the average value of the magnetization and $\mathcal{M}(t)$ its instantaneous value. We can discretize this function as follows, where we used our set of computed values $\mathcal{M}(t)$ for a set of discretized times (our Monte Carlo cycles corresponding to a sweep over the lattice)

$$\phi(t) = \frac{1}{t_{\max} - t} \sum_{t'=0}^{t_{\max}-t} \mathcal{M}(t') \mathcal{M}(t'+t) - \frac{1}{t_{\max} - t} \sum_{t'=0}^{t_{\max}-t} \mathcal{M}(t') \times \frac{1}{t_{\max} - t} \sum_{t'=0}^{t_{\max}-t} \mathcal{M}(t'+t).$$

# Correlation Time, how to compute

```
// define m-value at each cycle within loop over cycles
m_matrix[cycles] = Maverage/(n_spins**2)/cycles
//  Then compute phi(i)  as (in pseudocode)
    for  i = 1, mcs
         r = 1.0/(mcs-i)
         s = 0.0; v = 0.0; p= 0.0
         for j = 1, mcs-i
             p = p+ m_matrix(j)*m_matrix(j+i)
             s = s+ m_matrix(j)
             v = v+ m_matrix(j+i)
         end for
         phi(i)  = r*p-r*r*s*v
    end for
```

One should be careful with times close to $t_{\max}$, the upper limit of the sums becomes small and we end up integrating over a rather small time interval. This means that the statistical error in $\phi(t)$ due to the random nature of the fluctuations in $\mathcal{M}(t)$ can become large.

One should therefore choose $t \ll t_{\max}$.

Note also that we could replace the magnetization with the mean energy, or any other expectation values of interest.

The time-correlation function for the magnetization gives a measure of the correlation between the magnetization at a time $t'$ and a time $t' + t$. If we multiply the magnetizations at these two different times, we will get a positive contribution if the magnetizations are fluctuating in the same direction, or a negative value if they fluctuate in the opposite direction. If we then integrate over time, or use the discretized version of, the time correlation function $\phi(t)$ should take a non-zero value if the fluctuations are correlated, else it should gradually go to zero. For times a long way apart the magnetizations are most likely uncorrelated and $\phi(t)$ should be zero.

# Time Auto-correlation Function

We can derive the correlation time by observing that our Metropolis algorithm is based on a random walk in the space of all possible spin configurations. Our probability distribution function $\hat{\mathbf{w}}(t)$ after a given number of time steps $t$ could be written as

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t \hat{\mathbf{w}}(0),$$

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 of $\hat{\mathbf{v}}$ 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 $\lambda_i$ the $i^{\text{th}}$ eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

If we assume that $\lambda_0$ 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.

We can relate this property to an observable like the mean magnetization. With the probabilty $\hat{\mathbf{w}}(t)$ (which in our case is the Boltzmann distribution) we can write the mean magnetization as

$$\langle \mathcal{M}(t) \rangle = \sum_\mu \hat{\mathbf{w}}(t)_\mu \mathcal{M}_\mu,$$

or as the scalar of a vector product

$$\langle \mathcal{M}(t) \rangle = \hat{\mathbf{w}}(t)\mathbf{m},$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathcal{M}_\mu$ in its various microstates $\mu$.

## Time Auto-correlation Function

We rewrite this 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 $i^{\text{th}}$ 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 $t \to \infty$ the mean magnetization is dominated by the the largest eigenvalue $\lambda_0$, we can rewrite the last equation as

$$\langle \mathcal{M}(t) \rangle = \langle \mathcal{M}(\infty) \rangle + \sum_{i \neq 0} \lambda_i^t \alpha_i m_i.$$

We define the quantity

$$\tau_i = -\frac{1}{\log \lambda_i},$$

and rewrite the last expectation value as

$$\langle \mathcal{M}(t) \rangle = \langle \mathcal{M}(\infty) \rangle + \sum_{i \neq 0} \alpha_i m_i e^{-t/\tau_i}.$$

# Time Auto-correlation Function

The quantities $\tau_i$ are the correlation times for the system. They control also the auto-correlation function discussed above. The longest correlation time is obviously given by the second largest eigenvalue $\tau_1$, 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 $\lambda_1$ and we simulate long enough, $\tau_1$ may define the correlation time. In other cases we may not be able to extract a reliable result for $\tau_1$. Coming back to the time correlation function $\phi(t)$ we can present a more general definition in terms of the mean magnetizations $\langle \mathcal{M}(t) \rangle$. Recalling that the mean value is equal to $\langle \mathcal{M}(\infty) \rangle$ we arrive at the expectation values

$$\phi(t) = \langle \mathcal{M}(0) - \mathcal{M}(\infty) \rangle \langle \mathcal{M}(t) - \mathcal{M}(\infty) \rangle,$$

resulting in

$$\phi(t) = \sum_{i,j \neq 0} m_i \alpha_i m_j \alpha_j e^{-t/\tau_i},$$

which is appropriate for all times.

# Correlation Time

If the correlation function decays exponentially

$$\phi(t) \sim \exp\left(-t/\tau\right)$$

then the exponential correlation time can be computed as the average

$$\tau_{\exp} = -\langle \frac{t}{log|\frac{\phi(t)}{\phi(0)}|} \rangle.$$

If the decay is exponential, then

$$\int_0^\infty dt\phi(t) = \int_0^\infty dt\phi(0)\exp\left(-t/\tau\right) = \tau\phi(0),$$

which suggests another measure of correlation

$$\tau_{\mathrm{int}} = \sum_k \frac{\phi(k)}{\phi(0)},$$

called the integrated correlation time.

Here we mention that one can show, using scaling relations, that at the critical temperature the correlation time $\tau$ relates to the lattice size $L$ as

$$\tau \sim L^{d+z},$$

with $d$ the dimensionality of the system. For the Metropolis algorithm based on a single spin-flip process, Nightingale and Blöte obtained $z = 2.1665 \pm 0.0012$. This is a rather high value, meaning that our algorithm is not the best choice when studying properties of the Ising model near $T_C$.

We can understand this behavior by studying the development of the two-dimensional

Ising model as function of temperature.

# Time Auto-correlation Function

Cooling the system down to the critical temperature we observe clusters pervading larger areas of the lattice. The reason for the large correlation time (and the parameter $z$) for the single-spin flip Metropolis algorithm is the development of these large domains or clusters with all spins pointing in one direction. It is quite difficult for the algorithm to flip over one of these large domains because it has to do it spin by spin, with each move having a high probability of being rejected due to the ferromagnetic interaction between spins. Since all spins point in the same direction, the chance of performing the flip

$$E = -4J \qquad \uparrow \begin{matrix} \uparrow \\ \uparrow \\ \uparrow \end{matrix} \uparrow \qquad \Longrightarrow \qquad E = 4J \qquad \uparrow \begin{matrix} \uparrow \\ \downarrow \\ \uparrow \end{matrix} \uparrow$$

leads to an energy difference of $\Delta E = 8J$. Using the exact critical temperature $k_B T_C / J \approx 2.2.69$, we obtain a probability $\exp -(8/2.269) = 0.029429$ which is rather small. The increase in large correlation times due to increasing lattices can be diminished by using so-called cluster algorithms, such as that introduced by Ulli Wolff in 1989 and the Swendsen-Wang algorithm from 1987. The two-dimensional Ising model with the Wolff or Swendsen-Wang algorithms exhibits a much smaller correlation time, with the variable $z = 0.25 \pm 001$. Here, instead of flipping a single spin, one flips an entire cluster of spins pointing in the same direction.

# Better Algorithm needed

Monte Carlo simulations close to a phase transition are affected by critical slowing down. In the 2-D Ising system, the correlation length becomes very large, and the correlation time, which measures the number of steps between independent Monte Carlo configurations behaves like

$$\tau \sim \xi^z,$$

with $z \approx 2.1$ for the Metropolis algorithm. The exponent $z$ is called the dynamic critical exponent, The maximum possible value for $\xi$ in a finite system of $N = L \times L$ spins is $\xi \sim L$, because $\xi$ cannot be larger than the lattice size! This implies that $\tau \sim L^z \approx N$. This makes simulations difficult because the Metropolis algorithm time scales like $N$, so the time to generate independent Metropolis configurations scales like

$$N\tau \sim N^2 = L^4.$$

If the lattice size

$$L \rightarrow \sqrt{10}L \approx 3.2L$$

, the simulation time increases by a factor of 100.

# Better Algorithm needed

There is a simple physical argument which helps understand why $z = 2$, The Metropolis algorithm is a local algorithm, i.e., one spin is tested and flipped at a time. Near $T_C$ the system develops large domains of correlated spins which are difficult to break up. So the most likely change in configuration is the movement of a whole domain of spins. But one Metropolis sweep of the lattice can move a domain at most by approximately one lattice spacing in each time step. This motion is stochastic, i.e., like a random walk. The distance traveled in a random walk scales like $\sqrt{\text{time}}$, so to move a domain a distance of order $\xi$ takes $\tau \sim \xi^2$ Monte Carlo steps. This argument suggests that the way to speed up a Monte Carlo simulation near $T_C$ is to use a non-local algorithm.

The essential idea of this algorithm suggested by R.H. Swendsen and J.-S. Wang, Phys. Rev. Lett. **58**, 86 (1987), is to identify clusters of like spins and treat each cluster as a giant spin to be flipped according to a random criterion. It is necessary that the algorithm obey the detailed balance condition. Swendsen and Wang found the following algorithm based on ideas from percolation theory.

Freeze/delete bonds: The $2 - D$ square lattice, periodic boundary conditions, has $N = L \times L$ spins and $2N$ bonds between spins. Construct a bond lattice as follows:

- If the bond connects opposite spins, then delete it, i.e., temporarily uncouple the two spins. Note that opposite spins have a higher bond energy $+J$ if $J > 0$ and thus a higher effective temperature. So if $J$ is large we are effectively "melting" the bond.

- If the bond connects like spins (both up or both down), then delete the bond with probability $e^{-2J/(k_B T)}$, i.e., generate a random deviate $r$ and delete the bond if $r < e^{-2J/(k_B T)}$. Note that a like-spin pair has bond energy $-J$: so the change in energy in flipping one spin of the pair, i.e., in going from like to unlike spins is $E = 2J$. Bonds which survive this test are "frozen". The probability of this happening is $1 - e^{-2J/(kBT)}$. If $T = 0$ all like-spin bonds get frozen, while at $T = \infty$ the freezing probability is zero and all the bonds melt.

Note that constructing the bond lattice takes time of $O(N)$ because there are $2N$ bonds.

# Swendsen-Wang Algorithm, Ising Model

- After the bond lattice has been set up, the spins are decomposed into clusters. A cluster is simply a domain of spins connected to one another by frozen bonds. The lattice obviously decomposes into clusters in a unique way, and the decomposition is a deterministic problem. Cluster decomposition is potentially time consuming. A naive algorithm can take time of $O(N^2)$, so it is essential to use a decomposition algorithm that scales linearly with lattice size like Metropolis!

- Spin Up date: So far, constructing the bond lattice and identifying clusters has not changed any of the spins. The spins in each cluster are now "frozen" and the bonds between different clusters have been deleted. Each cluster is now updated by assigning a random new value $\pm 1$ to all of the spins simultaneously, i.e., generate a random deviate $r$ and flip all spins in that cluster if $r < 0.5$. Note that $T$ does not play a role in this flipping decision. The spin update step scales like the number of clusters which is $< N$. Swendsen and Wang showed that $z \approx 0.35$ for this algorithm in the $2 - D$ Ising model. Assuming that each Swendsen-Wang step scales like $N$, the running time for the simulation scales like

$$N\tau \sim N\xi^{0.35} \sim NL^{0.35} \sim N^{1.175}$$

which is much better than $O(N^2)$ with Metropolis.

Two years after Swendsen and Wang published their algorithm, U. Wolff, Phys. Rev. Lett. **62**, 361 (1989) published an even more efficient algorithm based on constructing and flipping one single cluster at a time Freeze/delete bonds: The $2 - D$ square lattice, periodic boundary conditions, has $N = L \times L$ spins and $2N$ bonds between spins. Construct a bond lattice as follows:

- Choose a random spin in the lattice

- Look at the nearest neighbors of that spin. If they point in the same direction as the seed spin, add them to the cluster with a probability $P_{add} = 1 - e^{-2J/(kBT)}$ as in the Swendsen-Wang algo.

- For each spin that is added, examine each ot its neighbors. If they point in the same direction, add each with the probability $P_{add}$. Repeat till no more neighbors to consider for inclusions.

- Flip the cluster.

Consider two states of the total lattice, $i$ a and $j$. They differ from one another by the flipping of a single cluster of similarly oriented spins. The crucial thing to notice is the way the spins are oriented around the edge of the cluster. The bonds between these spins and the ones in the cluster have to be broken when the cluster is flipped. This means that the bonds which are not broken in going from $i$ to $j$ must be broken when we flip back again from $j$ to $i$

When we move from $i$ to $j$ we break $m$ bonds in order to flip the cluster. These bonds represent pairs of similarly oriented spins which are not added to the cluster. The probability of not adding a spin is $1 - P_{\mathrm{add}}$. If we break $m$ bonds we have a selection probability $(1 - P_{\mathrm{add}})^m$.

If there are $n$ bonds to break in the reverse move, then the selection probability is $(1 - P_{\mathrm{add}})^n$.

An important condition we require that our Markov chain should satisfy is that of detailed balance. At equilibrium detailed balance gives thus

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

We model the transition probability as $W(j \rightarrow i) = g(j \rightarrow i)A(j \rightarrow i)$, where $g$ is the selection probability and $A$ is the acceptance probability. Our cluster algo gives

$$\frac{g(j \rightarrow i)A(j \rightarrow i)}{g(i \rightarrow j)A(i \rightarrow j)} = \frac{w_i}{w_j},$$

which with the Boltzmann distribution results in

$$(1 - P_{\mathrm{add}})^{m-n} \frac{A(j \rightarrow i)}{A(i \rightarrow j)} = e^{-\beta(E_i - E_j)}.$$

The change between the two states depends on the number of broken bonds. For each broken bond of the $m$ ones the change is (Ising model, you must figure out what this means for the Potts model) $+2J$. For each broken bond for the reverse process (total $n$) we an energy change $-2J$. That means that we get

$$(1 - P_{\text{add}})^{m-n} \frac{A(j \to i)}{A(i \to j)} = e^{-\beta 2J(m-n)},$$

which we rewrite as

$$\frac{A(j \to i)}{A(i \to j)} = \left( e^{\beta 2J} (1 - P_{\text{add}}) \right)^{n-m},$$

and using the fact that $P_{\text{add}} = 1 - e^{-2J/(kBT)}$ we obtain a ratio between the acceptance probabilities as

$$\frac{A(j \to i)}{A(i \to j)} = 1$$

The cluster is always flipped. We make the acceptance ratios for both forward and backward moves equal unity. Note that we need not go through the whole lattice to flip a cluster, this should be contrasted with the Swendsen-Wang also. Wolff showed that $z \approx 0.25$ for this algorithm.

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 $P(x, t)$!!

### 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 $\Psi$ 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 | $\vec{r}$ | $\widehat{\mathbf{r}} = \vec{r}$ |
| Momentum | $\vec{p}$ | $\widehat{\mathbf{p}} = -i\hbar\vec{\bigtriangledown}$ |
| Orbital momentum | $\vec{L} = \vec{r} \times \vec{p}$ | $\widehat{\mathbf{L}} = \vec{r} \times (-i\hbar\vec{\bigtriangledown})$ |
| 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 $\Phi$ is a linear combination of the eigenfunctions $\psi_\nu$ 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 $\Phi$, the probability of obtaining the value $a_\nu$ as the outcome of a measurement of the physical quantity A is given by $|c_\nu|^2$ and $\psi_\nu$ is an eigenfunction of $\widehat{\mathbf{A}}$ with eigenvalue $a_\nu$.*

As a consequence one can show that:
*when a quantal system is in the state $\Phi$, the mean value or expectation value of a physical quantity $A(\vec{r}, \vec{p})$ is given by*

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

We have assumed that $\Phi$ 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 and repetition

- Monday: Repetition from last week
- Quantum Monte Carlo, last part of lectures. Chapter 11.1-11.4.
- Wednesday: Repetition of course and discussion of oral exam.

# 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

$$\langle H \rangle =$$

$$\frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \ldots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R}_N) H(\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \ldots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R_2}, \ldots, \mathbf{R}_N)},$$

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).

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.

# Quantum Monte Carlo

Given a hamiltonian $H$ and a trial wave function $\Psi_T$, 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.

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.

# Quantum Monte Carlo

The tedious part in a VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). DMC is a way of solving exactly the many-body Schrödinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

## Quantum Monte Carlo

- Construct first a trial wave function $\psi_T^\alpha(\mathbf{R})$, for a many-body system consisting of $N$ particles located at positions $\mathbf{R} = (\mathbf{R_1}, \ldots, \mathbf{R_N})$. The trial wave function depends on $\alpha$ variational parameters $\alpha = (\alpha_1, \ldots, \alpha_N)$.

- Then we evaluate the expectation value of the hamiltonian $H$

$$\langle H \rangle = \frac{\int d\mathbf{R} \Psi_{T_\alpha}^*(\mathbf{R}) H(\mathbf{R}) \Psi_{T_\alpha}(\mathbf{R})}{\int d\mathbf{R} \Psi_{T_\alpha}^*(\mathbf{R}) \Psi_{T_\alpha}(\mathbf{R})}. \tag{250}$$

- Thereafter we vary $\alpha$ according to some minimization algorithm and return to the first step.

# Quantum Monte Carlo

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

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

This is our new probability distribution function (PDF).

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

where $\Psi$ 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 **R** and variational parameters $\alpha$ and calculate $|\psi_T^\alpha(\mathbf{R})|^2$.

- 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 $\alpha$ 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

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right).$$

| $\alpha$ | $\langle H \rangle$ | $\sigma^2$ | $\sigma/\sqrt{N}$ |
|---|---|---|---|
| 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 |

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

$$H\psi = \text{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

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

# 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

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

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 $\Psi$ at the origin.

## Quantum Monte Carlo

This results in

$$\frac{1}{\mathcal{R}_T(r_1)}\frac{d\mathcal{R}_T(r_1)}{dr_1} = -Z,$$

and

$$\mathcal{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have

$$\frac{1}{\mathcal{R}_T(r)}\frac{d\mathcal{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Similalry, studying the case $r_{12} \rightarrow 0$ we can write a possible trial wave function as

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

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.

## Place and duration

- 10/12-17/12.
- 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). Latex example on webpage. Choose among projects 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.

### Pensum/syllabus

Go to `http://www.uio.no/studier/emner/matnat/fys/FYS3150/h08/` and click on syllabus.

## Topics

- Linear algebra and eigenvalue problems. (Lecture notes chapters 4.1-4.4 and 12.1-12.7 and projects 1 and 3), Blitz is not part of the curriculum.
- Numerical integration, standard methods and Monte Carlo methods (Lecture notes chapters 7.1-7.4 and 8), MPI is not part of the curriculum
- Monte Carlo methods in physics (Lecture notes chapters 9 and 10, project 5)
- Ordinary differential equations (Lecture notes chapters 13 and 14.1-14.3 and project 2)
- Partial differential equations (Lecture notes chapter 15, project 4)

## Linear algebra and eigenvalue problems, chapters 4.1-4.4, 6.4 and 12.1-12.7

- 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 (project 3)
- Cubic spline, chapter 6.4.

### Numerical integration, standard methods and Monte Carlo methods (7.1-7.4 and 8)

- Trapezoidal, rectangle and Simpson's rules
- Gaussian quadrature, emphasis on Legendre polynomials, but you need to know about other polynomials as well.
- Brute force Monte Carlo integration
- Random numbers (simplest algo, ran0) and probability distribution functions, expectation values
- Improved Monte Carlo integration and importance sampling.
- MPI chapter 7.7 is not part of curriculum.

### Monte Carlo methods in physics (9 and 10)

- Random walks and Markov chains and relation with diffusion equation (project 5)
- Metropolis algorithm, detailed balance and ergodicity (project 5)
- Aplications to Ising model (project 5)

### Ordinary differential equations (13 and 14.1-14.3)

- Euler's method and improved Euler's method, truncation errors
- Runge Kutta methods, 2nd and 4th order, truncation errors (project 2)
- How to implement a second-order differential equation, both linear and non-linear. How to make your equations dimensionless.
- Boundary value problems, Numerov's method, shooting and matching method (chap 14.1-14.3).

## Partial differential equations

- Set up diffusion, Poisson and wave equations up to 2 spatial dimensions and time
- Set up the mathematical model and algorithms for these equations, with boundary and initial conditions. The stability conditions for the diffusion equation.
- Explicit, implicit and Crank-Nicolson schemes, and how to solve them. Remember that they result in triangular matrices.
- How to compute the Laplacian in Poisson's equation.
- How to solve the wave equation in one and two dimensions (project 5).

# 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, Monte Carlo, DFT and other many-body methods)
- 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).*