Slides from FYS3150/FYS4150 Lectures

Morten Hjorth-Jensen

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

[2]Department of Physics and Astronomy, Michigan State University
East Lansing, Michigan, USA

Fall 2005

Outline

Outline

Outline

Week 34, 22-26 August

- Monday: First lecture: Presentation of the course, aims and content
- Monday: Second Lecture: Introduction to C++ programming and numerical precision.
- Wednesday: Numerical precision and C++ programming, continued
- Computer-Lab: thursday and friday 9am-7pm. Presentation of hardware and software at room FV329.

Outline

Lectures and ComputerLab

- Lectures: monday (12.15am-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
- Weekly plans and all other information are on the official webpage.

Course Format

- Several computer exercises, 6 compulsory projects. Electronic reports only. Classfronter as course organizer.
- Oral examination based on the reports and five selected topics, see the syllabus link on the webpage. Dates to be settled, most likely week 50, 12-16 December.
- The computer lab consists of 16 Linux PCs. C/C++ is the default programming language, but F90/95 is also used. All source codes discussed during the lectures can be found at the webpage of the course. Alternatively you can also use Java, or compiled languages like Matlab and Maple. Furthermore, you could also use Python. Beware that we cannot guide you in Java.

ComputerLab

day	teacher
Thursday 9am-1 pm	Morten Hjorth-Jensen
Thursday 1pm-5pm	MHJ/Maxim Kartamychev
Friday 9am-1pm	MK
Fredag 1pm-5pm	MK

The lab is open till 7pm, but from 5pm till 7pm there will only be student assistants.
Set up your preferred lab time.

Topics covered

- Numerical precision and intro to C++ programming
- Numerical derivation and integration
- Random numbers and Monte Carlo integration
- Monte Carlo methods in statistical physics
- Quantum Monte Carlo methods
- Linear algebra and eigenvalue problems
- Non-linear equations and roots of polynomials
- Ordinary differential equations
- Partial differential equations
- Interpolation and extrapolation
- Fitting of data

Outline

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.

Outline

Loss of numerical precision

Suppose we wish to evaluate the function

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

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

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

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

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

and

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

Loss of numerical precision

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

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

while the second expression results in

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

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

Loss of numerical precision

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

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

Outline

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* .

Makefiles and simple scripts

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

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

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

Hello world

The C encounter

Here we present first the C version.

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

Hello World

Dissection I

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

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

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

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

Hello World

Dissection II

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

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

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

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

Hello World

Now in C++

Here we present first the C++ version.

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

C++ Hello World

Dissection I

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

Limits

C++ and Fortran declarations

type in C/C++ and Fortran 90/95	bits	range
char/CHARACTER	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
int/INTEGER (2)	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
int/long int/INTEGER(4)	32	-2147483648 to 2147483647
signed long int	32	-2147483648 to 2147483647
float/REAL(4)	32	$3.4e^{-38}$ to $3.4e^{+38}$
double/REAL(8)	64	$1.7e^{-308}$ to $1.7e^{+308}$
long double	64	$1.7e^{-308}$ to $1.7e^{+308}$

From decimal to binary representation

How to do it

$$a_n2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_02^0.$$

In binary notation we have thus $(417)_{10} = (110100001)_2$ since we have

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

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

417/2=208	remainder 1	coefficient of 2^0 is 1
208/2=104	remainder 0	coefficient of 2^1 is 0
104/2=52	remainder 1	coefficient of 2^2 is 0
52/2=27	remainder 1	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 1	coefficient of 2^6 is 0
3/2= 1	remainder 1	coefficient of 2^7 is 1
1/2= 0	remainder 1	coefficient of 2^8 is 1

From decimal to binary representation

Integer numbers

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

From decimal to binary representation

Integer numbers, Fortran

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

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

END PROGRAM binary_integer
```

Integer Numbers

Possible Overflow for Integers

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

Loss of Precision, bad thing

Real Numbers

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

Loss of precision, real numbers

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

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

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

More on Loss of Precision

Real Numbers

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

Another simple 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(
         sum += term;
         n++;
     }  // end of while() loop
```

Program to compute $\exp(-x)$

Oh, it never ends!

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

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

Results $\exp{(-x)}$

What is going on?

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

Program to compute $\exp(-x)$

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

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

Program to compute $\exp(-x)$

Last statements

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

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

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

Results $\exp(-x)$

More Problems

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

Week 35, 29 August - 2 September

- Numerical differentiation and loss of numerical precision (chapter 3 lecture notes)
- Monday: Repetition from last week
- C/C++ programming details, pointers, read/write to/from file
- Wednesday: Intro to linear Algebra
- 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 3.1 and presentation of Blitz++

Repetition

Machine Numbers

In the machine a number is represented as

$$fl(x) = x(1+\epsilon) \tag{2}$$

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

$$fl(a) = fl(b) - fl(c) = a(1+\epsilon_a), \tag{3}$$

or

$$fl(a) = b(1+\epsilon_b) - c(1+\epsilon_c), \tag{4}$$

meaning that

$$fl(a)/a = 1 + \epsilon_b\frac{b}{a} - \epsilon_c\frac{c}{a}, \tag{5}$$

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

Repetition

Machine Numbers

Define the absolute error as

$$|fl(a) - a|, \tag{6}$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a. \tag{7}$$

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - f(c) - (b - c)|}{a}, \tag{8}$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}. \tag{9}$$

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.

Repetition

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

$$\epsilon_{ro} \approx \sqrt{N}\epsilon_M, \tag{10}$$

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

$$\epsilon_{approx} = \frac{\alpha}{N^\beta}, \tag{11}$$

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

$$\epsilon_{tot} = \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_M \tag{12}$$

Pointer example I

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

Pointer example I

Discussion

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

Pointer example II

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

Pointer example II

Discussion

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

Pointer example II

Discussion

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

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

File handling, C-way

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

File handling, C way contn

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

File handling, C++-way

You must first declare input and output files

```
#include <fstream>


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

File handling, C++-way

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

File handling, C++-way

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

File handling, C++-way

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

File handling, C++-way

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

      Read something from the file:

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

      Can test on success of reading:

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

Call by value and reference

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

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

Call by value and reference

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

Call by value and reference

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

  The value of a: a. The address of a: &a
  The value of b: *b. The address of b: &b.
- Line 3: The value of a is now 10.
- Line 4: Memory to store 10 integers is reserved. The address to the first location is stored in b. Address to element number 6 is given by the expression (b + 6).
- Line 5: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;

Call by value and reference

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

Call by value and reference

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

Call by value and reference

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

Call by value and reference

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

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

whereas in C++ we would write

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

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

Call by value and reference, F90/95

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

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

This hinders unwanted changes and increases readability.

Most used formula for derivatives

3 point formulae

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

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

Second derivative

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

Error Analysis

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

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

For the computed second derivative we have

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

and the truncation or approximation error goes like

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

Error Analysis

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

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

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

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

Error Analysis

Our total error becomes

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

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

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

With double precision and $x = 10$ we obtain

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

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

Error Analysis

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

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

as

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

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

Error Analysis

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

Error Analysis

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

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

Week 36, 5-9 September

Linear Algebra

- Monday: Repetition from last week
- Presentation of Project 1, deadline 19 september (midnight).
- 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.
- LU decomposition and linear equations
- Wednesday: Further discussion of linear algebra methods, numerical stability
- Computer-Lab: thursday and friday 9am-7pm, Project 1.

Basic Matrix Features

Matrix Properties Reminder

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

The inverse of a matrix is defined by

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

Basic Matrix Features

Matrix Properties Reminder

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

Basic Matrix Features

Some Equivalent Statements

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

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

Matrix Handling in C/C++

Row Major Order, Addition

We have $N \times N$ matrices A, B and C and we wish to evaluate $A = B + C$. 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 $A = BC$. In C/C++ this would be coded like

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

Matrix Handling in Fortran 90/95

Column Major Order

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

Fortran 90 writes the above statements in a much simpler way

```
a=b+c
```

Multiplication

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

Makefile for Blitz++

```
# Path where Blitz is installed
BZDIR = /site/Blitz++-0.8
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
```

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);
```

Blitz++ example

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

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

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

Blitz++ example

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

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

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

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

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

Blitz++ example

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

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

thirdIndex k;             // Placeholder for the third index

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

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

Blitz++ example

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

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

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

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

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

Blitz++ example

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

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

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

  return 0;
}
```

Week 37, 12-16 September

Numerical Integration and Monte Carlo

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

Equal Step Methods

Generalities

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

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

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

Equal Step Methods

Trapezoidal Rule

Taylor expansion

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

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

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

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

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

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

Equal Step Methods

Trapezoidal Rule

Easy to implement numerically through the following simple algorithm

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

Trapezoidal Rule

Simple Code

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

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

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

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

Equal Step Methods

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

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

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

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

for the weights, while the trapezoidal rule resulted in

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

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

Gaussian Quadrature

A greater precision for a given amount of numerical work can be achieved if we are willing to give up the requirement of equally spaced integration points. In Gaussian quadrature (hereafter GQ), both the mesh points and the weights are to be determined. The points will not be equally spaced The theory behind GQ is to obtain an arbitrary weight ω through the use of so-called orthogonal polynomials. These polynomials are orthogonal in some interval say e.g., [-1,1]. Our points x_i are chosen in some optimal sense subject only to the constraint that they should lie in this interval. Together with the weights we have then $2N$ (N 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

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

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

Gaussian Quadrature

Weight Functions

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

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

Gaussian Quadrature

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

$$f(x) \approx P_{N-1}(x),$$

with N mesh points we should be able to integrate exactly the polynomial P_{N-1}.

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

$$f(x) \approx P_{2N-1}(x),$$

and with only N mesh points these methods promise that

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

Legendre

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

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

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

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

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

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

and the recursion relation

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

Laguerre

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

These polynomials arise from the solution of the differential equation

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

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

$$\int_{-\infty}^{\infty} e^{-x} \mathcal{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathcal{L}_{n+1}(x) = (2n+1-x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x).$$

Hermite

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{\infty} e^{-x^2} g(x)dx.$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

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

They fulfil the orthorgonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

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

Why Monte Carlo?

An example from quantum mechanics: most problems of interest in e.g., atomic, molecular, nuclear and solid state physics consist of a large number of interacting electrons and ions or nucleons. The total number of particles N is usually sufficiently large that an exact solution cannot be found. Typically, the expectation value for a chosen hamiltonian for a system of N particles is

$$\langle H \rangle =$$

$$\frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) H(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)},$$

an in general intractable problem.

This integral is actually the starting point in a Variational Monte Carlo calculation.

Gaussian quadrature: Forget it! given 10 particles and 10 mesh points for each degree of freedom and an ideal 1 Tflops machine (all operations take the same time), how long will it ta ke to compute the above integral? Lifetime of the universe $T \approx 4.7 \times 10^{17}$s.

More on dimensionality

As an example from the nuclear many-body problem, we have Schrödinger's equation as a differential equation

$$\hat{H}\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A) = E\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A)$$

where

$$\mathbf{r}_1, .., \mathbf{r}_A,$$

are the coordinates and

$$\alpha_1, .., \alpha_A,$$

are sets of relevant quantum numbers such as spin and isospin for a system of A nucleons ($A = N + Z$, N being the number of neutrons and Z the number of p rotons).

Even more on dimensionality

There are

$$2^A \times \begin{pmatrix} A \\ Z \end{pmatrix}$$

coupled second-order differential equations in $3A$ dimensions.

For a nucleus like ^{10}Be this number is **215040**. This is a truely challenging many-body problem.

Plan for Monte Carlo Lectures

- This week: intro, random numbers and PDFs
- Next week:MC integration and random walks
- Third week: Random walks and statistical physics
- Fourth week: statistical physics
- Fifth and sixth week: quantum Monte Carlo

Monte Carlo Keywords

Consider it is a numerical experiment

- Random variables
- 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!

Monte Carlo Integration

$$I=\int_0^1 f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i),$$

$$I=\int_0^1 f(x)dx \approx \langle f\rangle.$$

$$\sigma_f^2=\frac{1}{N}\sum_{i=1}^{N} f(x_i)^2-\left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2,$$

or

$$\sigma_f^2=\left(\langle f^2\rangle-\langle f\rangle^2\right).$$

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 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;
      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;
}
```

Week 38, 19-23 September

Numerical Integration and Monte Carlo

- Monday: Repetition from last week
- Brief discussion of project 2.
- Monte Carlo integration
- Random numbers
- PDF
- Wednesday: Monte Carlo integration continues
- Introduction to random walks
- Computer-Lab: thursday and friday 9am-7pm, Project 1.

Radioactive Decay

Probability for a decay of a particle during a time step Δt is

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

λ is inversely proportional to the lifetime

- Choose the number of particles $N(t=0) = N_0$.
- Make a loop over the number of time steps, with maximum time bigger than the number of particles N_0
- At every time step there is a probability λ for decay. Compare this probability with a random number x.
- If $x \leq \lambda$, reduce the number of particles with one i.e., $N = N - 1$. If not, keep the same number of particles till the next time step.
- Increase by one the time step (the external loop)

Probability Distribution Functions PDF

	Discrete PDF	continuous PDF
Domain	$\{x_1, x_2, x_3, \dots, x_N\}$	$[a, b]$
probability	$p(x_i)$	$p(x)dx$
Cumulative	$P_i = \sum_{l=1}^{i} p(x_l)$	$P(x) = \int_a^x p(t)dt$
Positivity	$0 \le p(x_i) \le 1$	$p(x) \ge 0$
Positivity	$0 \le P_i \le 1$	$0 \le P(x) \le 1$
Monotonuous	$P_i \ge P_j$ if $x_i \ge x_j$	$P(x_i) \ge P(x_j)$ if $x_i \ge x_j$
Normalization	$P_N = 1$	$P(b) = 1$

Expectation Values

- Discrete PDF

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

- Continuous PDF

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

- Function $f(x)$

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

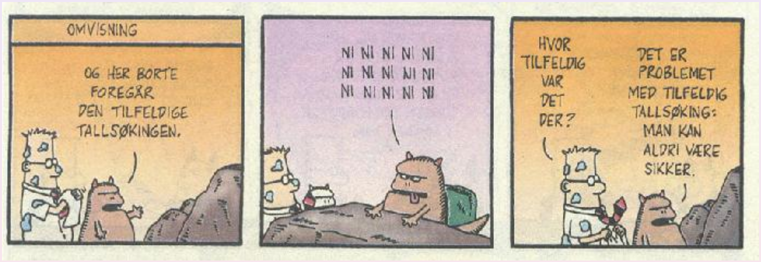
- Variance

$$\sigma^2 = \langle f^2\rangle - \langle f\rangle^2$$

Important PDFs

- uniform distribution
$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$
- 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 $[-\infty, \infty]$

Random Numbers


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

Week 39, 26-30 September

Random Walks, Markov Chains, Diffusion and Metropolis

- Monday: Repetition from last week
- Random walks and diffusion
- Derivation of diffusion equation from Markov chains
- Entropy of a random walk
- Wednesday:
- More diffusion and entropy
- Ergodic assumption and detailed balance
- Metropolis algorithm

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

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i,n+1)+w(i,n)}{\Delta t}, \tag{13}$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1,n)+w(i-1,n)-w(i,n)}{(\Delta x)^2}, \tag{14}$$

resulting in the discretized diffusion equation

$$\frac{w(i,n+1)+w(i,n)}{\Delta t} = D\frac{w(i+1,n)+w(i-1,n)-w(i,n)}{(\Delta x)^2}, \tag{15}$$

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

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \left\{ \begin{array}{cc} \frac{1}{2} & |i-j| = 1 \\ 0 & \mathrm{else} \end{array} \right. \tag{16}$$

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

$$w_i(t = \epsilon) = W(j \rightarrow i) w_j(t = 0). \tag{17}$$

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

$$\sum_i w_i(t) = 1, \tag{18}$$

and

$$\sum_j W(j \rightarrow i) = 1. \tag{19}$$

The further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$. Note that the probability for remaining at the same place is in general not necessarily equal zero. In our Markov process we allow only for jumps to the left or to the right.

Diffusion from Markov Chain

The time development of our initial PDF can now be represented through the action of the transition probability matrix applied n times. At a time $t_n = n\epsilon$ our initial distribution has developed into

$$w_i(t_n) = \sum_j W_{ij}(t_n)w_j(0), \tag{20}$$

and defining

$$W(il - jl, n\epsilon) = (W^n(\epsilon))_{ij} \tag{21}$$

we obtain

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0), \tag{22}$$

or in matrix form

$$w(\hat{n}\epsilon) = \hat{W}^n(\epsilon)\hat{w}(0). \tag{23}$$

Diffusion from Markov Chain

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

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{24}$$

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

$$\hat{R} = \left(\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right), \tag{25}$$

and

$$\hat{L} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right). \tag{26}$$

Diffusion from Markov Chain

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

$$\hat{L}\hat{R} = \hat{R}\hat{L} = 1, \tag{27}$$

and

$$\hat{L} = \hat{R}^{-1} \tag{28}$$

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

$$\hat{L}\hat{R} = \sum_k \hat{L}_{ik}\hat{R}_{kj} = \sum_k \delta_{(i+1),k}\delta_{k,(j+1)} = \delta_{i+1,j+1} = \delta_{i,j}, \tag{29}$$

and only the diagonal matrix elements are different from zero.

Diffusion from Markov Chain

For the first time step we have thus

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{30}$$

and using the properties in Eqs. (27) and (28) we have after two time steps

$$\hat{W}^2(2\epsilon) = \frac{1}{4}\left(\hat{L}^2 + \hat{R}^2 + 2\hat{R}\hat{L}\right), \tag{31}$$

and similarly after three time steps

$$\hat{W}^3(3\epsilon) = \frac{1}{8}\left(\hat{L}^3 + \hat{R}^3 + 3\hat{R}\hat{L}^2 + 3\hat{R}^2\hat{L}\right). \tag{32}$$

Diffusion from Markov Chain

Using the binomial formula

$$\sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{a}^k \hat{b}^{n-k} = (a+b)^n, \tag{33}$$

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

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{R}^k \hat{L}^{n-k}, \tag{34}$$

or

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{L}^{n-2k} = \frac{1}{2^n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \hat{R}^{2k-n}, \tag{35}$$

Diffusion from Markov Chain

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

$$W(il - jl, n\epsilon) = \begin{cases} \frac{1}{2^n} \begin{pmatrix} n \\ \frac{1}{2}(n+i-j) \end{pmatrix} & |i-j| \leq n \\ 0 & \text{else} \end{cases}, \tag{36}$$

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. (23) we have

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0) = \sum_j \frac{1}{2^n} \left(\begin{array}{c} n \\ \frac{1}{2}(n+i-j) \end{array} \right) \delta_{j,0}, \tag{37}$$

resulting in

$$w_i(n\epsilon) = \frac{1}{2^n} \left(\begin{array}{c} n \\ \frac{1}{2}(n+i) \end{array} \right) \qquad |i| \le n \tag{38}$$

Diffusion from Markov Chain

Using the recursion relation for the binomials

$$\left(\begin{array}{c} n+1 \\ \frac{1}{2}(n+1+i)) \end{array}\right) = \left(\begin{array}{c} n \\ \frac{1}{2}(n+i+1) \end{array}\right) + \left(\begin{array}{c} n \\ \frac{1}{2}(n+i)-1 \end{array}\right) \tag{39}$$

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

$$w(x,t) = w(il, n\epsilon) = w_i(n\epsilon), \tag{40}$$

$$w(x, t+\epsilon) = \frac{1}{2}w(x+l, t) + \frac{1}{2}w(x-l, t), \tag{41}$$

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

Diffusion from Markov Chain

$$\frac{w(x,t+\epsilon)-w(x,t)}{\epsilon}=\frac{l^2}{2\epsilon}\frac{w(x+l,t)-2w(x,t)+w(x-l,t)}{l^2}, \tag{42}$$

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.

Entropy and Equilibrium

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

$$S = -\sum_i w_i ln(w_i), \tag{43}$$

where w_i is the probability of finding our system in a state i. For our one-dimensional randow walk it represents the probability for being at position $i = i\Delta x$ after a given number of time steps. Assume now that we have N random walkers at $i = 0$ and $t = 0$ and let these random walkers diffuse as function of time. We compute then the probability distribution for N walkers after a given number of steps i along x and time steps j. We can then compute an entropy S_j for a given number of time steps by summing over all probabilities i. The code used to compute these results is in programs/chapter9/program4.cpp. Here we have used 100 walkers on a lattice of length from $L = -50$ to $L = 50$ employing periodic boundary conditions meaning that if a walker reaches the point $x = L$ it is shifted to $x = -L$ and if $x = -L$ it is shifted to $x = L$.

Entropy

```
// loop over all time steps
  for (int step=1; step <= time_steps; step++){
    // move all walkers with periodic boundary conditions
    for (int walks = 1; walks <= walkers; walks++){
      if (ran0(&idum) <= move_probability) {
if ( x[walks] +1 > length) {
  x[walks] = -length;
}
else{
  x[walks] += 1;
}
      }
      else {
if ( x[walks] -1 < -length) {
  x[walks] = length;
        }
else{
  x[walks] -= 1;
}
      }
    }  // end of loop over walks
```

Entropy

```
// at the final time step we compute the probability
// by counting the number of walkers at every position
for ( int i = -length; i <= length; i++){
  int count = 0;
  for( int j = 1; j <= walkers; j++){
    if ( x[j] == i ) {
      count += 1;
    }
  }
  probability[i+length] = count;
}
```

Entropy

```
// Writes the results to screen
void output(int length, int time_steps, int walkers, int *probability)
{
  double entropy, histogram;
  // find norm of probability
  double norm = 1.0/walkers;
  // compute the entropy
  entropy = 0.; histogram = 0.;
  for( int  i = -length; i <=  length; i++){
    histogram = (double) probability[i+length]*norm;
    if ( histogram > 0.0) {
    entropy -= histogram*log(histogram);
    }
  }
    cout << setiosflags(ios::showpoint | ios::uppercase);
    cout << setw(6) << time_steps;
    cout << setw(15) << setprecision(8) << entropy << endl;
}  // end of function output
```

Entropy

At small time steps the entropy is very small, reflecting the fact that we have an ordered state. As time elapses, the random walkers spread out in space (here in one dimension) and the entropy increases as there are more states, that is positions accesible to the system. We say that the system shows an increased degree of disorder. After several time steps, we see that the entropy reaches a constant value, a situation called a steady state. This signals that the system has reached its equilibrium situation and that the random walkers spread out to occupy all possible available states. At equilibrium it means thus that all states are equally probable and this is not baked into any dynamical equations such as Newton's law of motion. It occursbecause the system is allowed to explore all possibilities. An important hypothesis, which has never been proven rigorously but for certain systems, 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

Test of the Metropolis Algorithm

Want to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{44}$$

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

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{45}$$

with mass $m = 1$.

Test of the Metropolis Algorithm

- Read in the temperature T, the number of Monte Carlo cycles, and the initial velocity. You should also read in the change in velocity δv used in every Monte Carlo step. Let the temperature have dimension energy.
- Thereafter choose a maximum velocity given by e.g., $v_{max} \sim 10\sqrt{T}$. Then you construct a velocity interval defined by v_{max} and divided it in small intervals through v_{max}/N, with $N \sim 100 - 1000$. For each of these intervals find out how many times a given velocity during the Monte Carlo sampling appears in each specific interval.
- The number of times a given velocity appears in a specific interval is used to construct a histogram representing $P(v)dv$. To achieve this construct a vector $P[N]$ which contains the number of times a given velocity appears in the subinterval $v, v + dv$.

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
   }
```

Week 40, 3-7 October

Statistical Physics, Spins Systems and Phase Transitions

- Monday: Repetition from last week
- Introduction to Statistical Physics
- Phase transitions in magnetic spin systems,
- Wednesday:
- Phase transitions in magnetic spin systems,
- Discussion of the Ising model in 1 and 2 dimensions
- How to model phase transitions for spin systems

Detailed Balance

- 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

$$\sum_j W(j \rightarrow i)w_j = \sum_i W(i \rightarrow j)w_i$$

  Ensures that it is the Boltzmann distrubution 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

$$W(j \rightarrow i)w_j = W(i \rightarrow j)w_i$$

  Satisfies the detailed balance condition

Boltzmann Distribution

- At equilibrium detailed balance gives

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

- Boltzmann distribution

$$\frac{w_i}{w_j} = \exp\left(-\beta(E_i - E_j)\right)$$

Ergodicity

It should be possible for any Markov process to reach every possible state of the system from any starting point if the simulations is carried out for a long enough time.
Any state in a Boltzmann distribution has a probability different from zero and if such a state cannot be reached from a given starting point, then the system is not ergodic.

Selection Rule

- 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

$$\frac{g(j \rightarrow i)A(j \rightarrow i)}{g(i \rightarrow j)A(i \rightarrow j)} = \exp\left(-\beta(E_i - E_j)\right)$$

Metropolis Algorithm

For a system which follows the Boltzmann distribution the Metropolis algorithm reads

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

This algorithm satisfies the condition for detailed balance and ergodicity.

Implementation

- Establish an initial energy E_b
- Do a random change of this initial state by e.g., flipping an individual spin. This new state has energy E_t. Compute then $\Delta E = E_t - E_b$
- If $\Delta E \leq 0$ accept the new configuration.
- If $\Delta E > 0$, compute $w = e^{-(\beta \Delta E)}$.
- Compare w with a random number r. If $r \leq w$ accept, else keep the old configuration.
- Compute the terms in the sums $\sum A_s P_s$.
- Repeat the above steps in order to have a large enough number of microstates
- For a given number of MC cycles, compute then expectation values.

Statistical Physics

	Microcanonical	Canonical	Grand canonical	Pressure canon
Exchange of heat with the environment	no	yes	yes	yes
Exchange of particles with the environemt	no	no	yes	no
Thermodynamical parameters	$V, \mathcal{M}, \mathcal{D}$ E N	$V, \mathcal{M}, \mathcal{D}$ T N	$V, \mathcal{M}, \mathcal{D}$ T μ	$P, \mathcal{H}, \mathcal{E}$ T N
Potential	Entropy	Helmholtz	PV	Gibbs
Energy	Internal	Internal	Internal	Enthalpy

Microcanonical Ensemble

Entropy

$$S = k_B ln\Omega \tag{46}$$

$$dS = \frac{1}{T}dE + \frac{p}{T}dV - \frac{\mu}{T}dN \tag{47}$$

Temperature

$$\frac{1}{k_B T} = \left(\frac{\partial ln\Omega}{\partial E}\right)_{N,V} \tag{48}$$

Pressure

$$\frac{p}{k_B T} = \left(\frac{\partial ln\Omega}{\partial V}\right)_{N,E} \tag{49}$$

Chemical potential

$$\frac{\mu}{k_B T} = -\left(\frac{\partial ln\Omega}{\partial N}\right)_{V,E} \tag{50}$$

Canonical Ensemble

Helmholtz Free Energy

$$F = -k_B T ln Z \tag{51}$$

$$dF = -SdT - pdV + \mu dN \tag{52}$$

Entropy

$$S = k_B ln Z + k_B T \left(\frac{\partial ln Z}{\partial T}\right)_{N,V} \tag{53}$$

Pressure

$$p = k_B T \left(\frac{\partial ln Z}{\partial V}\right)_{N,T} \tag{54}$$

Chemical Potential

$$\mu = -k_B T \left(\frac{\partial ln Z}{\partial N}\right)_{V,T} \tag{55}$$

Energy (internal only)

$$E = k_B T^2 \left(\frac{\partial ln Z}{\partial T}\right)_{V,N} \tag{56}$$

Grand Canonical Ensemble

Potential

$$pV = k_B T ln\Xi \tag{57}$$

$$d(pV) = SdT + Nd\mu + pdV \tag{58}$$

Entropy

$$S = k_B ln\Xi + k_B T \left(\frac{\partial ln\Xi}{\partial T}\right)_{V,\mu} \tag{59}$$

Particles

$$N = k_B T \left(\frac{\partial ln\Xi}{\partial \mu}\right)_{V,T} \tag{60}$$

Pressure

$$p = k_B T \left(\frac{\partial ln\Xi}{\partial V}\right)_{\mu,T} \tag{61}$$

Pressure Canonical Ensemble

Gibbs Free Energy

$$G = -k_B T ln\Delta \quad (62)$$

$$dG = -SdT + Vdp + \mu dN \quad (63)$$

Entropy

$$S = k_B ln\Delta + k_B T \left(\frac{\partial ln\Delta}{\partial T}\right)_{p,N} \quad (64)$$

Volume

$$V = -k_B T \left(\frac{\partial ln\Delta}{\partial p}\right)_{N,T} \quad (65)$$

Chemical potential

$$\mu = -k_B T \left(\frac{\partial ln\Delta}{\partial N}\right)_{p,T} \quad (66)$$

Expectation Values

At a given temperature we have the probability distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{67}$$

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

$$Z = \sum_{i=1}^{M} e^{-\beta E_i}, \tag{68}$$

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

$$\langle E\rangle = \sum_{i=1}^{M} E_i P_i(\beta) = \frac{1}{Z}\sum_{i=1}^{M} E_i e^{-\beta E_i}, \qquad (69)$$

with a corresponding variance defined as

$$\sigma_E^2 = \langle E^2\rangle - \langle E\rangle^2 = \frac{1}{Z}\sum_{i=1}^{M} E_i^2 e^{-\beta E_i} - \left(\frac{1}{Z}\sum_{i=1}^{M} E_i e^{-\beta E_i}\right)^2. \qquad (70)$$

If we divide the latter quantity with kT^2 we obtain the specific heat at constant volume

$$C_V = \frac{1}{kT^2}\left(\langle E^2\rangle - \langle E\rangle^2\right). \qquad (71)$$

Expectation Values

We can also write

$$\langle E\rangle = -\frac{\partial \ln Z}{\partial\beta}. \tag{72}$$

The specific heat is

$$C_V = \frac{1}{kT^2}\frac{\partial^2 \ln Z}{\partial\beta^2} \tag{73}$$

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.

Expectation Values

$$\langle \mathcal{M} \rangle = \sum_i^M \mathcal{M}_i P_i(\beta) = \frac{1}{Z} \sum_i^M \mathcal{M}_i e^{-\beta E_i}, \tag{74}$$

and the corresponding variance

$$\sigma_{\mathcal{M}}^2 = \langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 = \frac{1}{Z} \sum_{i=1}^M \mathcal{M}_i^2 e^{-\beta E_i} - \left(\frac{1}{Z} \sum_{i=1}^M \mathcal{M}_i e^{-\beta E_i} \right)^2 . \tag{75}$$

This quantity defines also the susceptibility χ

$$\chi = \frac{1}{kT} \left(\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 \right) . \tag{76}$$

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 diverges we talk of a phase transition of order of the derivative.

Ising Model

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

$$E = -J \sum_{<kl>}^{N} s_k s_l - B \sum_{k}^{N} s_k, \qquad (77)$$

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.

Week 42, 17-21 October

Statistical Physics, Spins Systems and Phase Transitions

- Monday: Repetition from last week
- Discussion of project 4
- Discussion of the code for the Ising Model
- Phase transitions in magnetic spin systems
- Wednesday:
- Quantum Monte Carlo, variational Monte Carlo
- The Hydrogen atom and the Helium atom

Potts Model, Project 4

Energy given by

$$E = -J \sum_{<kl>}^{N} \delta_{s_l,s_k},$$

where the spin s_k at lattice position k can take the values $1, 2, \ldots, q$. N is the total number of spins. For $q = 2$ the Potts model corresponds to the Ising model, we can rewrite the last equation as

$$E = -\frac{J}{2} \sum_{<kl>}^{N} 2(\delta_{s_l,s_k} - \frac{1}{2}) - \sum_{<kl>}^{N} \frac{J}{2}.$$

Now, $2(\delta_{s_l,s_k} - \frac{1}{2})$ is +1 when $s_l = s_k$ and -1 when they are different. Equivalent except the last term which is a constant and that $J \rightarrow J/2$. Tip when comparing results with the Ising model: remove the constant term. The first step is thus to check that your algorithm for the Potts model gives the same results as the ising model. Note that critical temperature for the $q = 2$ Potts model is half of that for the Ising model.

Potts Model, Project 4

For $q = 3$ and higher you can proceed as follows:

- Do a calculation with a small lattice first over a large temperature region. Use typical temperature steps of 0.1.
- Establish a small region where you see the heat capacity and the susceptibility start to increase.
- Decrease the temperature step in this region and perform calculations for larger lattices as well.

For $q = 6$ and $q = 10$ we have a first order phase transition, the energy starts to diverge.

Project 4: 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 ΔE 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.

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.

Modelling the Ising Model

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 ΔE. 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 $\Delta E = 8J$.

$$E = -4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E = 4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

Modelling the Ising Model

The four other possibilities are as follows

$$E = -2J \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E = 2J \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

with $\Delta E = 4J$,

$$E = 0 \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \uparrow & \uparrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = 0 \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \downarrow & \uparrow \\ & \downarrow & \end{array}$$

with $\Delta E = 0$

Modelling the Ising Model

$$E = 2J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \uparrow & \uparrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = -2J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \downarrow & \uparrow \\ & \downarrow & \end{array}$$

with $\Delta E = -4J$ and finally

$$E = 4J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \uparrow & \downarrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = -4J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \downarrow & \downarrow \\ & \downarrow & \end{array}$$

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.

The loop over T in main

```
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] = 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 Fortran program

```
   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
```

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
```

Analytic Results: one-dimensional Ising model

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

$$Z_N = \sum_{s_1=\pm 1}\cdots\sum_{s_N=\pm 1}\exp\left(\beta J\sum_{j=1}^{N-1} s_j s_{j+1}\right), \qquad (78)$$

and since the last spin occurs only once in the last sum in the exponential, we can single out the last spin as follows

$$\sum_{s_N=\pm 1}\exp\left(\beta J s_{N-1}s_N\right) = 2cosh(\beta J). \qquad (79)$$

The partition function consists then of a part from the last spin and one from the remaining spins resulting in

$$Z_N = Z_{N-1}2cosh(\beta J). \qquad (80)$$

Analytic Results: one-dimensional Ising model

We can repeat this process and obtain

$$Z_N = (2cosh(\beta J))^{N-2} Z_2, \tag{81}$$

with Z_2 given by

$$Z_2 = \sum_{s_1=\pm 1} \sum_{s_2=\pm 1} \exp(\beta J s_1 s_2) = 4cosh(\beta J), \tag{82}$$

resulting in

$$Z_N = 2(2cosh(\beta J))^{N-1}. \tag{83}$$

In the thermodynamical limit where we let $N \rightarrow \infty$, 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.

Analytic Results: one-dimensional Ising model

We can then calculate the mean energy with free ends from the above formula for the partition function using

$$\langle E\rangle = -\frac{\partial lnZ}{\partial\beta} = -(N-1)Jtanh(\beta J). \tag{84}$$

Helmholtz's free energy is given by

$$F = -k_BTlnZ_N = -Nk_BTln\left(2cosh(\beta J)\right). \tag{85}$$

The specific heat in one-dimension with free ends is

$$C_V = \frac{1}{kT^2}\frac{\partial^2}{\partial\beta^2}lnZ_N = (N-1)k\left(\frac{\beta J}{cosh(\beta J)}\right)^2. \tag{86}$$

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.

Analytic Results: one-dimensional Ising model

If we use periodic boundary conditions, the partition function is given by

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp\left(\beta J \sum_{j=1}^{N} s_j s_{j+1}\right), \tag{87}$$

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

$$Z_N = \sum_{\{s_i=\pm 1\}} \prod_{i=1}^{N} \exp\left(\beta J s_j s_{j+1}\right), \tag{88}$$

where the first sum is meant to represent all lattice sites. Introducing the matrix $\hat{\mathbf{T}}$ (the so-called transfer matrix)

$$\hat{\mathbf{T}} = \begin{pmatrix} e^{\beta J} & e^{-\beta J} \\ e^{-\beta J} & e^{\beta J} \end{pmatrix}, \tag{89}$$

Analytic Results: one-dimensional Ising model

$$Z_N = \sum_{\{s_i=\pm 1\}} \hat{\mathbf{T}}_{s_1 s_2} \hat{\mathbf{T}}_{s_2 s_3} \dots \hat{\mathbf{T}}_{s_N s_1} = Tr \hat{\mathbf{T}}^N. \tag{90}$$

The 2×2 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 λ_1^N and λ_2^N and the trace of $\hat{\mathbf{T}}^N$ is just the sum over eigenvalues resulting in a partition function

$$Z_N = \lambda_1^N + \lambda_2^N = 2^N \left([cosh(\beta J)]^N + [sinh(\beta J)]^N \right). \tag{91}$$

Helmholtz's free energy is in this case

$$F = -k_B T ln(\lambda_1^N + \lambda_2^N) = -k_B T \left\{ N ln(\lambda_1) + ln \left(1 + (\frac{\lambda_2}{\lambda_1})^N \right) \right\} \tag{92}$$

which in the limit $N \to \infty$ results in $F = -k_B T N ln(\lambda_1)$

Analytic Results: one-dimensional Ising model

Hitherto we have limited ourselves to studies of systems with zero external magnetic field, viz $\mathcal{H} = \imath$. We will mostly study systems which exhibit a spontaneous magnitization. It is however instructive to extend the one-dimensional Ising model to $\mathcal{H} \neq \imath$, yielding a partition function (with periodic boundary conditions)

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp\left(\beta \sum_{j=1}^{N} (J s_j s_{j+1} + \frac{\mathcal{H}}{2}(s_i + s_{j+1})\right), \qquad (93)$$

which yields a new transfer matrix with matrix elements $t_{11} = e^{\beta(J+\mathcal{H})}$, $t_{1-1} = e^{-\beta J}$, $t_{-11} = e^{\beta J}$ and $t_{-1-1} = e^{\beta(J-\mathcal{H})}$ with eigenvalues

$$\lambda_1 = e^{\beta J} cosh(\beta J) + \left(e^{2\beta J} sinh^2(\beta\mathcal{H}) + \rceil^{-\beta\in\mathcal{J}}\right)^{1/2}, \qquad (94)$$

and

$$\lambda_2 = e^{\beta J} cosh(\beta J) - \left(e^{2\beta J} sinh^2(\beta\mathcal{H}) + \rceil^{-\beta\in\mathcal{J}}\right)^{1/2}. \qquad (95)$$

Analytic Results: one-dimensional Ising model

It is now useful to compute the expectation value of the magnetisation per spin

$$\langle \mathcal{M}/N \rangle = \frac{1}{NZ} \sum_{i}^{M} \mathcal{M}_i e^{-\beta E_i} = -\frac{1}{N} \frac{\partial F}{\partial \mathcal{H}}, \qquad (96)$$

resulting in

$$\langle \mathcal{M}/N \rangle = \frac{sinh(\beta\mathcal{H})}{\left(sinh^2(\beta\mathcal{H}) + \rceil^{-\beta\in\mathcal{J}}\right)^{1/2}}. \qquad (97)$$

We see that for $\mathcal{H} = \prime$ 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.

Analytic Results: two-dimensional Ising model

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

$$Z_N = \left[2cosh(\beta J)e^I\right]^N, \tag{98}$$

with

$$I = \frac{1}{2\pi}\int_0^{\pi} d\phi ln\left[\frac{1}{2}\left(1+(1-\kappa^2 sin^2\phi)^{1/2}\right)\right], \tag{99}$$

and

$$\kappa = 2sinh(2\beta J)/cosh^2(2\beta J). \tag{100}$$

Analytic Results: two-dimensional Ising model

The resulting energy is given by

$$\langle E \rangle = -Jcoth(2\beta J)\left[1 + \frac{2}{\pi}(2tanh^2(2\beta J) - 1)K_1(q)\right], \tag{101}$$

with $q = 2sinh(2\beta J)/cosh^2(2\beta J)$ and the complete elliptic integral of the first kind

$$K_1(q) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - q^2 sin^2\phi}}. \tag{102}$$

Analytic Results: two-dimensional Ising model

Differentiating once more with respect to temperature we obtain the specific heat given by

$$C_V = \frac{4k_B}{\pi}(\beta Jcoth(2\beta J))^2$$

$$\left\{K_1(q) - K_2(q) - (1 - tanh^2(2\beta J))\left[\frac{\pi}{2} + (2tanh^2(2\beta J) - 1)K_1(q)\right]\right\},$$

with

$$K_2(q) = \int_0^{\pi/2} d\phi\sqrt{1 - q^2 sin^2\phi}. \qquad (103)$$

is the complete elliptic integral of the second kind. Near the critical temperature T_C the specific heat behaves as

$$C_V \approx -\frac{2}{k_B\pi}\left(\frac{J}{T_C}\right)^2 ln\left|1 - \frac{T}{T_C}\right| + \text{const.} \qquad (104)$$

Analytic Results: two-dimensional Ising model

In theories of critical phenomena one has that

$$C_V \sim \left|1 - \frac{T}{T_C}\right|^{-\alpha}, \tag{105}$$

and Onsager's result is a special case of this power law behavior. The limiting form of the function

$$lim_{\alpha\rightarrow 0}\frac{1}{\alpha}(Y^{-\alpha} - 1) = -lnY, \tag{106}$$

meaning that the analytic result is a special case of the power law singularity with $\alpha = 0$.

Analytic Results: two-dimensional Ising model

One can also show that the mean magnetization per spin is

$$\left[1-\frac{(1-tanh^2(\beta J))^4}{16tanh^4(\beta J)}\right]^{1/8}$$

for $T < T_C$ and 0 for $T > T_C$. The behavior is thus as $T \rightarrow T_C$ from below

$$M(T) \sim (T_C - T)^{1/8}$$

The susceptibility behaves as

$$\chi(T) \sim |T_C - T|^{-7/4}$$

Scaling Reuslts

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

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta}\,, \tag{107}$$

where β is a so-called critical exponent. A similar relation applies to the heat capacity

$$C_V(T) \sim |T_C - T|^{-\gamma}\,, \tag{108}$$

and the susceptibility

$$\chi(T) \sim |T_C - T|^{-\alpha}\,. \tag{109}$$

Scaling Results

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

$$T_C(L) - T_C(L=\infty) \sim aL^{-1/\nu}, \tag{110}$$

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta} \rightarrow L^{-\beta/\nu}, \tag{111}$$

$$C_V(T) \sim |T_C - T|^{-\gamma} \rightarrow L^{\gamma/\nu}, \tag{112}$$

and

$$\chi(T) \sim |T_C - T|^{-\alpha} \rightarrow L^{\alpha/\nu}. \tag{113}$$

We can compute the slope of the curves for M, C_V and χ as function of lattice sites and extract the exponent ν.

Quantum Monte Carlo and Schrödinger's equation

For one-body problems (one dimension)

$$-\frac{\hbar^2}{2m}\nabla^2\Psi(x,t)+V(x,t)\Psi(x,t)=\imath\hbar\frac{\partial\Psi(x,t)}{\partial t},$$

$$P(x,t)=\Psi(x,t)^*\Psi(x,t)$$

$$P(x,t)dx=\Psi(x,t)^*\Psi(x,t)dx$$

Interpretation: probability of finding the system in a region between x and $x+dx$. Always real

$$\Psi(x,t)=R(x,t)+\imath I(x,t)$$

yielding

$$\Psi(x,t)^*\Psi(x,t)=(R-\imath I)(R+\imath I)=R^2+I^2$$

Variational Monte Carlo uses only $P(x,t)$!!

Quantum Monte Carlo and Schrödinger's equation

Petit digression

Choose $\tau = it/\hbar$.
The time-dependent (1-dim) Schrödinger equation becomes then

$$\frac{\partial \Psi(x,\tau)}{\partial \tau} = \frac{\hbar^2}{2m}\frac{\partial^2 \Psi(x,\tau)}{\partial x^2} - V(x,\tau)\Psi(x,\tau).$$

With $V = 0$ we have a diffusion equation in complex time with diffusion constant

$$D = \frac{\hbar^2}{2m}.$$

Used in diffusion Monte Carlo calculations. Topic for FYS4410, Computational Physics II

Quantum Monte Carlo and Schrödinger's equation

Conditions which Ψ has to satisfy:

1. Normalization

$$\int_{-\infty}^{\infty} P(x,t)dx = \int_{-\infty}^{\infty} \Psi(x,t)^*\Psi(x,t)dx = 1$$

   meaning that

$$\int_{-\infty}^{\infty} \Psi(x,t)^*\Psi(x,t)dx < \infty$$

2. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be finite
3. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be continuous.
4. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be single valued

Square integrable functions.

First Postulate

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

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

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

Second Postulate

The only possible outcome of an ideal measurement of the physical quantity A are the eigenvalues of the corresponding quantum mechanical operator $\widehat{\mathbf{A}}$.

$$\widehat{\mathbf{A}}\psi_\nu = a_\nu\psi_\nu,$$

resulting in the eigenvalues $a_1, a_2, a_3, \cdots$ as the only outcomes of a measurement. The corresponding eigenstates $\psi_1, \psi_2, \psi_3 \cdots$ contain all relevant information about the system.

Third Postulate

Assume Φ is a linear combination of the eigenfunctions ψ_ν for $\widehat{\mathbf{A}}$,

$$\Phi = c_1\psi_1 + c_2\psi_2 + \cdots = \sum_\nu c_\nu \psi_\nu.$$

The eigenfunctions are orthogonal and we get

$$c_\nu = \int (\Phi)^* \psi_\nu d\tau.$$

From this we can formulate the third postulate:

When the eigenfunction is Φ, the probability of obtaining the value a_ν as the outcome of a measurement of the physical quantity A is given by $|c_\nu|^2$ and ψ_ν is an eigenfunction of $\widehat{\mathbf{A}}$ with eigenvalue a_ν.

Third Postulate

As a consequence one can show that:
when a quantal system is in the state Φ, the mean value or expectation value of a physical quantity $A(\vec{r}, \vec{p})$ is given by

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

We have assumed that Φ has been normalized, viz., $\int (\Phi)^* \Phi d\tau = 1$. Else

$$\langle A \rangle = \frac{\int (\Phi)^* \widehat{\mathbf{A}} \Phi d\tau}{\int (\Phi)^* \Phi d\tau}.$$

Fourth Postulate

The time development of of a quantal system is given by

$$i\hbar\frac{\partial\Psi}{\partial t} = \widehat{\mathbf{H}}\Psi,$$

with $\widehat{\mathbf{H}}$ the quantal Hamiltonian operator for the system.

Quantum Monte Carlo

Most quantum mechanical problems of interest in e.g., atomic, molecular, nuclear and solid state physics consist of a large number of interacting electrons and ions or nucleons. The total number of particles N is usually sufficiently large that an exact solution cannot be found. Typically, the expectation value for a chosen hamiltonian for a system of N particles is

$$\langle H\rangle =$$

$$\frac{\int d\mathbf{R}_1 d\mathbf{R}_2\dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)H(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2\dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)},$$

an in general intractable problem.

Quantum Monte Carlo

As an example from the nuclear many-body problem, we have Schrödinger's equation as a differential equation

$$\hat{H}\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A) = E\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A)$$

where

$$\mathbf{r}_1, .., \mathbf{r}_A,$$

are the coordinates and

$$\alpha_1, .., \alpha_A,$$

are sets of relevant quantum numbers such as spin and isospin for a system of A nucleons ($A = N + Z$, N being the number of neutrons and Z the number of protons).

Quantum Monte Carlo

There are

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

coupled second-order differential equations in $3A$ dimensions.

For a nucleus like ^{10}Be this number is **215040**. This is a truely challenging many-body problem.

Quantum Monte Carlo

Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

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

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \langle H \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system.

Quantum Monte Carlo

The trial wave function can be expanded in the eigenstates of the hamiltonian since they form a complete set, viz.,

$$\Psi_T(\mathbf{R}) = \sum_i a_i \Psi_i(\mathbf{R}),$$

and assuming the set of eigenfunctions to be normalized one obtains

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

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

Quantum Monte Carlo

In most cases, a wave function has only small values in large parts of configuration space, and a straightforward procedure which uses homogenously distributed random points in configuration space will most likely lead to poor results. This may suggest that some kind of importance sampling combined with e.g., the Metropolis algorithm may be a more efficient way of obtaining the ground state energy. The hope is then that those regions of configurations space where the wave function assumes appreciable values are sampled more efficiently.

The tedious part in a VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). DMC is a way of solving exactly the many-body Schrödinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

Quantum Monte Carlo

- Construct first a trial wave function $\psi_T^{\alpha}(\mathbf{R})$, for a many-body system consisting of N particles located at positions $\mathbf{R} = (\mathbf{R_1}, \dots, \mathbf{R_N})$. The trial wave function depends on α variational parameters $\alpha = (\alpha_1, \dots, \alpha_N)$.
- Then we evaluate the expectation value of the hamiltonian H

$$\langle H \rangle = \frac{\int d\mathbf{R}\Psi^{*}_{T_\alpha}(\mathbf{R})H(\mathbf{R})\Psi_{T_\alpha}(\mathbf{R})}{\int d\mathbf{R}\Psi^{*}_{T_\alpha}(\mathbf{R})\Psi_{T_\alpha}(\mathbf{R})}. \tag{114}$$

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

Quantum Monte Carlo

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

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

This is our new probability distribution function (PDF).

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

where Ψ is the exact eigenfunction.

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

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

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

Quantum Monte Carlo

Algo:

- Initialisation: Fix the number of Monte Carlo steps. Choose an initial **R** and variational parameters α and calculate $\left|\psi_T^{\alpha}(\mathbf{R})\right|^2$.
- Initialise the energy and the variance and start the Monte Carlo calculation (thermalize)
  1. Calculate a trial position $\mathbf{R}_p = \mathbf{R} + r * step$ where r is a random variable $r \in [0, 1]$.
  2. Metropolis algorithm to accept or reject this move
     $$w = P(\mathbf{R}_p)/P(\mathbf{R}).$$
  3. If the step is accepted, then we set $\mathbf{R} = \mathbf{R}_p$. Update averages
- Finish and compute final averages.

Week 43, 24-28 October

Quantum Monte Carlo, Ordinary Differential Equations

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

Quantum Monte Carlo

The radial Schrödinger equation for the hydrogen atom can be written as

$$-\frac{\hbar^2}{2m}\frac{\partial^2 u(r)}{\partial r^2}-\left(\frac{ke^2}{r}-\frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r)=Eu(r),$$

or with dimensionless variables

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2}-\frac{u(\rho)}{\rho}+\frac{l(l+1)}{2\rho^2}u(\rho)-\lambda u(\rho)=0,$$

with the hamiltonian

$$H=-\frac{1}{2}\frac{\partial^2}{\partial \rho^2}-\frac{1}{\rho}+\frac{l(l+1)}{2\rho^2}.$$

Use variational parameter α in the trial wave function

$$u_T^{\alpha}(\rho)=\alpha\rho e^{-\alpha\rho}.$$

Quantum Monte Carlo

Inserting this wave function into the expression for the local energy E_L gives

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right).$$

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03

Quantum Monte Carlo

We note that at $\alpha = 1$ we obtain the exact result, and the variance is zero, as it should. The reason is that we then have the exact wave function, and the action of the hamiltonan on the wave function

$$H\psi = \mathrm{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

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

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 $r_1 \rightarrow 0$.

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H\psi_T(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})}\left(-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1}\right)\psi_T(\mathbf{R}) + \text{finite terms}.$$

$$E_L(R) = \frac{1}{\mathcal{R}_T(r_1)}\left(-\frac{1}{2}\frac{d^2}{dr_1^2} - \frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathcal{R}_T(r_1) + \text{finite terms}$$

For small values of r_1, the terms which dominate are

$$\lim_{r_1 \rightarrow 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_1)}\left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathcal{R}_T(r_1),$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin.

Quantum Monte Carlo

This results in

$$\frac{1}{\mathcal{R}_T(r_1)}\frac{d\mathcal{R}_T(r_1)}{dr_1} = -Z,$$

and

$$\mathcal{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have

$$\frac{1}{\mathcal{R}_T(r)}\frac{d\mathcal{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Similary, 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.

Differential Equations

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

$$\frac{dy}{dt} = f(t, y). \tag{115}$$

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

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{116}$$

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

$$m\frac{d^2x}{dt^2} = -kx, \tag{117}$$

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

Differential Equations

partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t}=-\frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2}+\frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2}+\frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right)+V(\mathbf{x})\psi(\mathbf{x},t), \tag{118}$$

may depend on several variables. In certain cases, like the above equation, the wave function can be factorized in functions of the separate variables, so that the Schrödinger equation can be rewritten in terms of sets of ordinary differential equations. These equations are discussed in chapter 15. Involve boundary conditions in addition to initial conditions.

Differential Equations

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

$$\frac{dy}{dt} = g^3(t)y(t), \tag{119}$$

is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{120}$$

is a non-linear ODE.

Differential Equations

Another concept which dictates the numerical method chosen for solving an ODE, is that of initial and boundary conditions. To give an example, in our study of neutron stars below, we will need to solve two coupled first-order differential equations, one for the total mass m and one for the pressure P as functions of ρ

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

and

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

where ρ is the mass-energy density. The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, yielding $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star.
In the solution of the Schrödinger equation for a particle in a potential, we may need to apply boundary conditions as well, such as demanding continuity of the wave function and its derivative.

Differential Equations

In many cases it is possible to rewrite a second-order differential equation in terms of two first-order differential equations. Consider again the case of Newton's second law in Eq. (117). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{121}$$

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

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{122}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{123}$$

Differential Equations, Finite Difference

These methods fall under the general class of one-step methods. The algoritm is rather simple. Suppose we have an initial value for the function $y(t)$ given by

$$y_0 = y(t = t_0). \tag{124}$$

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

$$h = \frac{b-a}{N}. \tag{125}$$

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

$$y_1 = y(t_1 = t_0 + h), \tag{126}$$

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{127}$$

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{128}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{129}$$

Differential Equations

We define

$$y'(t_i) = f(t_i, y_i) \tag{130}$$

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

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{131}$$

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. (122) and (123). We define $y_0 = y^{(1)}(t=0)$ an $v_0 = y^{(2)}(t=0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{132}$$

and

$$y_1^{(2)} = v_0 - hy_0k/m + O(h^2). \tag{133}$$

Differential Equations

The Euler method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at $y_1^{(1)}$ using the velocity at $y_0^{(2)} = v_0$. A simple variation is to determine $y_{n+1}^{(1)}$ using the velocity at $y_{n+1}^{(2)}$, that is (in a slightly more generalized form)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{134}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{135}$$

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

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \tag{136}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f \tag{137}$$

and we can rewrite Eq. (128) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) + O(h^3), \tag{138}$$

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

Differential Equations

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \dots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}). \quad (139)$$

These methods, based on higher-order derivatives, are in general not used in numerical computation, since they rely on evaluating derivatives several times. Unless one has analytical expressions for these, the risk of roundoff errors is large.

Differential Equations

The most obvious improvements to Euler's and Euler-Cromer's algorithms, avoiding in addition the need for computing a second derivative, is the so-called midpoint method. We have then

$$y^{(1)}_{n+1} = y^{(1)}_{n} + \frac{h}{2}\left(y^{(2)}_{n+1} + y^{(2)}_{n}\right) + O(h^2) \tag{140}$$

and

$$y^{(2)}_{n+1} = y^{(2)}_{n} + ha_n + O(h^2), \tag{141}$$

yielding

$$y^{(1)}_{n+1} = y^{(1)}_{n} + hy^{(2)}_{n} + \frac{h^2}{2}a_n + O(h^3) \tag{142}$$

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

Differential Equations

Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.
One method that avoids this is the so-called half-step method. Here we define

$$y^{(2)}_{n+1/2} = y^{(2)}_{n-1/2} + ha_n + O(h^2), \tag{143}$$

and

$$y^{(1)}_{n+1} = y^{(1)}_{n} + hy^{(2)}_{n+1/2} + O(h^2). \tag{144}$$

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

$$y^{(2)}_{1/2} = y^{(2)}_{0} + ha_0 + O(h^2). \tag{145}$$

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

Differential Equations

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

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{146}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{147}$$

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

Differential Equations, Runge-Kutta methods

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}.
To see this, consider first the following definitions

$$\frac{dy}{dt} = f(t, y), \tag{148}$$

and

$$y(t) = \int f(t, y)dt, \tag{149}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{150}$$

Differential Equations, Runge-Kutta methods

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t,y)$ around the center of the integration interval t_i to t_{i+1}, i.e., at $t_i + h/2$, h being the step. Using the midpoint formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t,y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{151}$$

This means in turn that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{152}$$

Differential Equations, Runge-Kutta methods

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We have then

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{153}$$

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

$$k_1 = hf(t_i, y_i), \tag{154}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{155}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{156}$$

Differential Equations, Runge-Kutta methods

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely $t_i + h/2 = t_{(i+1/2)}$ where we evaluate the derivative f. This involves more operations, but the gain is a better stability in the solution.

Differential Equations, Runge-Kutta methods

The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, has the following algorithm

$$k_1 = hf(t_i, y_i), \tag{157}$$

$$k_2 = hf(t_i + h/2, y_i + k_1/2), \tag{158}$$

$$k_3 = hf(t_i + h/2, y_i + k_2/2) \tag{159}$$

$$k_4 = hf(t_i + h, y_i + k_3) \tag{160}$$

with the final value

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{161}$$

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 $O(h^4)$.

Runge-Kutta methods, code

```
void runge_kutta_4(double *y, double *dydx, int n,
                 double x, double h,
          double *yout, void (*derivs)(double, double *, double *))
{
  int i;
  double      xh,hh,h6;
  double *dym, *dyt, *yt;
  //   allocate space for local vectors
  dym = new double [n];
  dyt =  new double [n];
  yt =  new double [n];
  hh = h*0.5;
  h6 = h/6.;
  xh = x+hh;
```

Runge-Kutta methods, code

```
    for (i = 0; i < n; i++) {
      yt[i] = y[i]+hh*dydx[i];
    }
    (*derivs)(xh,yt,dyt);     // computation of k2
    for (i = 0; i < n; i++) {
      yt[i] = y[i]+hh*dyt[i];
    }
    (*derivs)(xh,yt,dym); //  computation of k3
    for (i=0; i < n; i++) {
      yt[i] = y[i]+h*dym[i];
      dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);    // computation of k4
    //      now we upgrade y in the array yout
    for (i = 0; i < n; i++){
      yout[i] = y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
    }
    delete []dym;
    delete [] dyt;
    delete [] yt;
  }       //  end of function Runge-kutta 4
```

Week 44, 31 October - 4 November

Ordinary Differential Equations

- Monday: Repetition from last week
- Discussion of project 5
- Discussion of the Pendulum and driven nonlinear oscillations.
- Wednesday:
- Ordinary Differential Equations: driven nonlinear oscillations and the route to chaos.
- Ordinary Differential equations with boundary conditions

More on the Pendulum Project

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

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{162}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{163}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{164}$$

More on the Pendulum Project

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{165}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \tag{166}$$

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

More on the Pendulum Project

b) Write then a code which solves

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

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

More on the Pendulum Project

c) Now we switch to

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

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

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

More on the Pendulum Project

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

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

More on the Pendulum Project, Period Doubling

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

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

More on the Pendulum Project

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

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

More on the Pendulum

Petit summary

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

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

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

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

Classes for ODE methods

In this course we have not emphasized the use of classes. However for the ordinary differential equations it can be very useful to make a Class which contains all possible methods discussed. In Fortran we can use the MODULE keyword in order to can methods and keep the variables private and hidden from other parts of our code. This allows for a generalization which can be used to tackle other ODEs as well.

Classes for ODE methods

In program2.cpp of chapter13 we have canned the following methods

- void euler();
- void euler_cromer();
- void midpoint();
- void euler_richardson();
- void half_step();
- void rk2(); //runge-kutta-second-order
- void rk4_step(double,double*,double*,double); // we need it in function rk4() and asc()
- void rk4(); //runge-kutta-fourth-order
- void asc(); //runge-kutta-fourth-order with adaptive stepsize control

Classes for ODE methods

```
class pendelum
 {
 private:
   double Q, A_roof, omega_0, omega_roof,g; //
   double y[2];          //for the initial-values of phi and v
   int n;                // how many steps
   double delta_t,delta_t_roof;

 public:
   void derivatives(double,double*,double*);
   void initialise();
   void euler();
   void euler_cromer();
   void midpoint();
   void euler_richardson();
   void half_step();
   void rk2(); //runge-kutta-second-order
   void rk4_step(double,double*,double*,double); // we need it in func
   void rk4(); //runge-kutta-fourth-order
   void asc(); //runge-kutta-fourth-order with adaptive stepsize contr
 };
```

Classes for ODE methods

```
void pendelum::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()
{
  pendelum testcase;
  testcase.initialise();
  testcase.euler();
  testcase.euler_cromer();
  testcase.midpoint();
  testcase.euler_richardson();
  testcase.half_step();
  testcase.rk2();
  testcase.rk4();
  return 0;
}  // end of main function
```

Classes for ODE methods

In Fortran we would use

```
MODULE pendelum
   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(..)
    ETC ETC

END MODULE pendulum
```

Week 45, 7- 11 November

Ordinary Differential Equations and Eigenvalue Problems

- Monday: Repetition from last week
- Discussion of project 5
- Ordinary Differential equations with boundary conditions
- Eigenvalues: Jacobi's method and Householder's QR algorithm
- Wednesday:
- QR Algorithm of Francis for the eigenvalue problem
- Eigenvalue problems for large matrices, Lanczo's algorithm.

Boundary Value Problems

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

In our case we are interested in attractive potentials

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

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

Boundary Value Problems

The final equation can be written as

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

where

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

Boundary Value Problems

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

and choose $\alpha = a$. Then

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

Boundary Value Problems

For small ρ we get

$$\frac{d^2}{d\rho^2}u(\rho) - \frac{l(l+1)}{\rho^2}u(\rho) = 0,$$

with solutions $u(\rho) = \rho^{l+1}$ or $u(\rho) = \rho^{-l}$. Since the final solution must be finite everywhere we get the condition for our numerical solution

$$u(\rho) = \rho^{l+1} \quad \text{for small } \rho$$

Boundary Value Problems

For large ρ we get

$$\frac{d^2}{d\rho^2}u(\rho) - \gamma\epsilon u(\rho) = 0 \quad \gamma > 0,$$

with solutions $u(\rho) = \exp(\pm\gamma\epsilon\rho)$ and the condition for large ρ means that our numerical solution must satisfy

$$u(\rho) = e^{-\gamma\epsilon\rho} \quad \text{for large } \rho$$

Boundary Value Problems

In order to find a bound state we start integrating, with a trial negative value for the energy, from small values of the variable ρ, usually zero, and up to some large value of ρ. As long as the potential is significantly different from zero the function oscillates. Outside the range of the potential the function will approach an exponential form. If we have chosen a correct eigenvalue the function decreases exponetially as $u(\rho) = e^{-\gamma\epsilon\rho}$. However, due to numerical inaccuracy the solution will contain small admixtures of the undesireable exponential growing function $u(\rho) = e^{+\gamma\epsilon\rho}$.

Boundary Value Problems

The final solution will then become unstable. Therefore, it is better to generate two solutions, with one starting from small values of ρ and integrate outwards to some matching point $\rho = \rho_m$. We call that function $u^<(\rho)$. The next solution $u^>(\rho)$ is then obtained by integrating from some large value ρ where the potential is of no importance, and inwards to the same matching point ρ_m. Due to the quantum mechanical requirements the logarithmic derivative at the matching point ρ_m should be well defined.

Boundary Value Problems

We obtain the following condition

$$\frac{\frac{d}{d\rho}u^{<}(\rho)}{u^{<}(\rho)} = \frac{\frac{d}{d\rho}u^{>}(\rho)}{u^{>}(\rho)} \quad \text{at} \quad \rho = \rho_m.$$

We can modify this expression by normalizing the function $u^{<}u^{<}(\rho_m) = Cu^{>}u^{<}(\rho_m)$.

$$\frac{d}{d\rho}u^{<}(\rho) = \frac{d}{d\rho}u^{>}(\rho) \quad \text{at} \quad \rho = \rho_m$$

We can calculate the first order derivatives by

$$\frac{d}{d\rho}u^{<}(\rho_m) \approx \frac{u^{<}(\rho_m) - u^{<}(\rho_m - h)}{h}$$
$$\frac{d}{d\rho}u^{>}(\rho_m) \approx \frac{u^{>}(\rho_m + h) - u^{>}(\rho_m)}{h}$$

Thus the criterium for a proper eigenfunction will be

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

which should be smaller than a fixed number.

Boundary Value Problems

The algorithm could then take the following form

- Initialise the problem by choosing minimum and maximum values for the energy, E_{min} and E_{max}, the maximum number of iterations $\mathrm{max\_iter}$ and the desired numerical precision.
- Search then for the roots of the function $f(E)$, where the root(s) is(are) in the interval $E \in [E_{\mathrm{min}}, E_{\mathrm{max}}]$ using e.g., the bisection method.

Boundary Value Problems

```
do {
    i++;
    e = (e_min+e_max)/2.;  //bisection
    if ( f(e)*f(e_max) > 0 ) {
       e_max = e; //change search interval
    }
    else { e_min = e; }
} while ( (fabs(f(e) > convergence_test) !! (i <= max_iterations))
```

Boundary Value Problems

- The use of a root-searching method forms the shooting part of the algorithm. We have however not yet specified the matching part.
- The matching part is given by the function $f(e)$ which receives as argument the present value of E. This function forms the core of the method and is based on an integration of Schrödinger's equation from $\rho = 0$ and $\rho = \infty$. If

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

  we have a solution.

The function $f(E)$ receives as input a guess for the energy. In the version implemented below, we use the standard three-point formula for the second derivative, namely

$$f_0'' \approx \frac{f_h - 2f_0 + f_{-h}}{h^2}.$$

Boundary Value Problems

```
void f(double step, int max_step, double energy, double *w, double *wf
{
   int      loop, loop_1,match;
   double   fac, wwf, norm;
// adding the energy guess to the array containing the potential
   for(loop = 0; loop <= max_step; loop ++) {
      w[loop] = (w[loop] - energy) * step * step + 2;
   }
}
```

Boundary Value Problems

```
// integrating from large r-values
   wf[max_step]     = 0.0;
   wf[max_step - 1] = 0.5 * step * step;
// search for matching point
   for(loop = max_step - 2; loop > 0; loop--) {
      wf[loop] = wf[loop + 1] * w[loop + 1] - wf[loop + 2];
      if(wf[loop] <= wf[loop + 1]) break;
   }
   match = loop + 1;
   wwf   = wf[match];
```

Boundary Value Problems

```
// start integrating up to matching point from r =0
   wf[0] = 0.0; wf[1] = 0.5 * step * step;
   for(loop = 2; loop <= match; loop++) {
      wf[loop] = wf[loop -1] * w[loop - 1] - wf[loop - 2];
      if(fabs(wf[loop]) > INFINITY) {
         for(loop_1 = 0; loop_1 <= loop; loop_1++) {
            wf[loop_1] /= INFINITY;
         }
      }
   }
   return fabs(wf[match-1]-wf[match+1]);
```

Week 46, 14- 18 November

Partial Differential Equations (PDEs)

- Monday: Repetition from last week
- Parabolic PDEs: The diffusion equation, implicit and explicit finite difference schemes
- Discussion of project 6
- Wednesday:
- Discussion of project 6
- Hyperbolic and Elliptic PDEs: Laplace's equation and the standard wave equation

Week 47, 21- 25 November

Partial Differential Equations (PDEs)

- Monday: Repetition from last week and discussion of Project 6
- Hyperbolic and Elliptic PDEs: Laplace’s equation and the standard wave equation
- Wednesday:
- Examples of physical systems which can be solved with PDEs

Partial Differential Equations

General 2+1-dim PDE

$$A(x,y)\frac{\partial^2 U}{\partial x^2} + B(x,y)\frac{\partial^2 U}{\partial x \partial y} + C(x,y)\frac{\partial^2 U}{\partial y^2} = F(x,y,U,\frac{\partial U}{\partial x},\frac{\partial U}{\partial y})$$

Examples

$$B = C = 0,$$

give e.g., 1+1-dim diffusion equation

$$A\frac{\partial^2 U}{\partial x^2} = \frac{\partial U}{\partial t}$$

and is an example of a parabolic PDE

Partial Differential Equations

More examples 2+1-dim wave equation

$$A\frac{\partial^2 U}{\partial x^2} + C\frac{\partial^2 U}{\partial y^2} = \frac{\partial^2 U}{\partial t^2}$$

Poisson's (Laplace's $\rho = 0$) equation

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

Heat Diffusion Equation

Diffusion equation

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

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

Explicit Scheme for the Diffusion Equation

In one dimension we have thus the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t}, \tag{167}$$

or

$$u_{xx} = u_t, \tag{168}$$

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

$$u(x,0) = g(x) \qquad 0 \leq x \leq L \tag{169}$$

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

$$u(0,t) = a(t) \qquad t \geq 0, \tag{170}$$

and

$$u(L,t) = b(t) \qquad t \geq 0, \tag{171}$$

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

Explicit Scheme, Forward Euler

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}, \tag{172}$$

and

$$u_{xx} \approx \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{173}$$

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

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+i,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{174}$$

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

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{175}$$

Explicit Scheme

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

with

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

yielding

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

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

$$\Delta t/\Delta x^2 \leq 1/2 \tag{176}$$

Implicit Scheme

Choose now

$$u_t \approx \frac{u(x_i,t_j)-u(x_i,t_j-k)}{k}$$

and

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

Define $\alpha = k/h^2$. Gives

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

Here $u_{i,j-1}$ is the only unknown quantity.

Have

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

with

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

which gives

$$V_j = A^{-1}V_{j-1} = \cdots = A^{-j}V_0$$

Need only to invert a matrix

Brute Force Implicit Scheme, inefficient algo

```
!  now invert the matrix
       CALL matinv( a, ndim, det)
       DO i = 1, m
          DO l=1, ndim
             u(l) = DOT_PRODUCT(a(l,:),v(:))
          ENDDO
          v = u
          t = i*k
          DO  j=1, ndim
              WRITE(6,*) t, j*h, v(j)
          ENDDO
       ENDDO
```

Brief Summary of the Explicit and the Implicit Methods

- Explicit is straightforward to code, but avoid doing the matrix vector multiplication since the matrix is tridiagonal.

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

- The implicit method can be applied in a brute force way as well as long as the element of the matrix are constants.

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

- However, it is more efficient to use a linear algebra solver for tridiagonal matrices.

Simple Test

Problem

$$\begin{cases} u_{xx} = u_t \\ u(x,0) = sin(\pi x) \\ u(0,t) = u(1,t) = 0 \\ u(x,t) = e^{-\pi^2 t} sin(\pi x) \end{cases}$$

More complicated initial and boundary conditions can be included.

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

$$\left(2\hat{I} + 2\alpha\hat{B}\right) V_j = \left(2\hat{I} - 2\alpha\hat{B}\right) V_{j-1},$$

where the vector V_j is the same as defined in the implicit case while the matrix $\hat{B}$ is

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

Tip for the analytic Solution of Project 6

Define a new function

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

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

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

Laplace's and Poisson's equations

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{177}$$

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

$$u_{xx} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}, \tag{178}$$

and

$$u_{yy} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2}, \tag{179}$$

Laplace's and Poisson's equations

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{180}$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{181}$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \tag{182}$$

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

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \rho_{i,j}. \tag{183}$$

Solution Approach

The way we solve these equations is based on an iterative scheme called the relaxation method. Its steps are rather simple. We start with an initial guess for $u_{i,j}^{(0)}$ where all values are known. To obtain a new solution we solve Eq. (182) or Eq. (183) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (182). This solution is in turn used to obtain a new and improved $u_{i,j}^{(2)}$. We continue this process till we obtain a result which satisfies some specific convergence criterion.

Iterative Scheme

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

Laplace's Equation, Simple Case

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

Week 48, 28 November - 2 December

End of PDE part, Summary and exam discussion

- Monday: Repetition from last week and discussion of Project 6
- End of partial differential equations part, discussion of how to solve Schrödinger's equation.
- Fast Fourier transforms, Danielson-Lanczos' algorithm.
- Wednesday:
- Summary and exam discussions. Project 6 ends this week.