Slides from FYS3150/4150 Lectures

Morten Hjorth-Jensen[1,2]

Department of Physics and Center of Mathematics for Applications, University of Oslo[1]

National Superconducting Cyclotron Laboratory, Michigan State University[2]

Aug 29, 2014

Overview of week 34

- Monday: First lecture: Presentation of the course, aims and content
- Monday: Second Lecture: Introduction to C++ programming and numerical precision.
- Tuesday: Numerical precision and C++ programming, continued and exercises for first week
- Numerical differentiation and loss of numerical precision (chapter 3 lecture notes)
- Computer lab: Thursday and Friday. First time: Thursday and Friday this week, Presentation of hardware and software at room FV329 first hour of every labgroup and solution of first simple exercises.

Lectures and ComputerLab

- Lectures: Monday (4.15pm-6pm) and Tuesday (12.15pm-2pm) only this and next week. Thereafter weeks 38 and 39, and weeks 43 and 44 and finally weeks 47 and 48.
- Weekly reading assignments needed to solve projects.
- First hour of each lab session used to discuss technicalities, address questions etc linked with projects.
- Detailed lecture notes, exercises, all programs presented, projects etc can be found at the homepage of the course.
- Computerlab: Thursday (9am-7pm) and Friday (9am-7pm) room FV329.
- Weekly plans and all other information are on the official webpage.
- Final written exam December 12, 9am (four hours).

Course Format

- Several computer exercises, 5 compulsory projects. Electronic reports only.
- Evaluation and grading: The last project (50
- The computer lab (room FV329)consists of 16 Linux PCs, but many prefer own laptops. C/C++ is the default programming language, but Fortran2008 and Python are also used. All source codes discussed during the lectures can be found at the webpage of the course. We recommend either C/C++, Fortran2008 or Python as languages.

ComputerLab

day	teacher
Thursday 9am-1pm	Anders, Morten L., Håvard, MHJ
Thursday 1pm-5pm	Anders, Morten L., Håvard, MHJ
Friday 9am-1pm	Anders, Morten L., Håvard, MHJ
Friday 1pm-5pm	Anders, Morten L., Håvard, MHJ

Topics covered in this course

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

Syllabus FYS3150

Linear algebra and eigenvalue problems chapters 6 and 7.

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

Syllabus FYS3150

Linear algebra and eigenvalue problems chapters 6 and 7.

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

Syllabus FYS3150

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

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

Syllabus FYS3150

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

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

Syllabus FYS3150

Ordinary differential equations (chapters 8 and 9).

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

Syllabus FYS3150

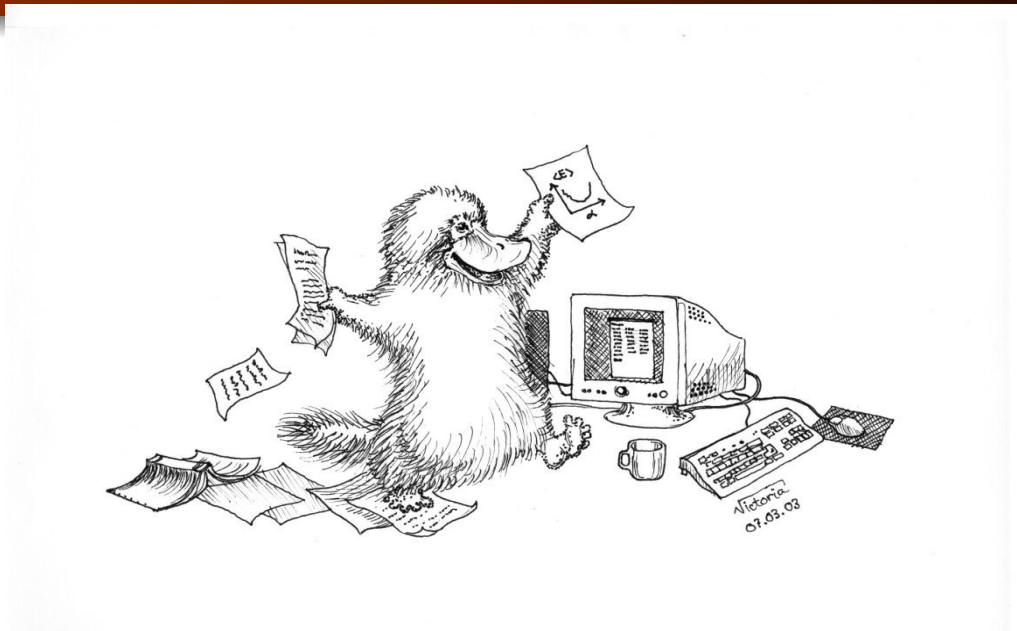
Partial differential equations chapter 10.

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

Overarching aims of this course

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

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



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

Other courses in Computational Science at UiO

Bachelor/Master/PhD Courses.

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

Extremely useful tools, strongly recommended

and discussed at the lab sessions the first week.

- GIT for version control (see webpage)
- ipython notebook
- QTcreator for editing and mastering computational projects (for C++ codes, see webpage of course)
- Armadillo as a useful numerical library for C++, highly recommended
- Unit tests, see also webpage
- Devilry for handing in projects

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.

Compiling and linking without QTcreator.

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

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

where the compiler is called through the command c++/g++. The compiler option -Wall means that a warning is issued in case of non-standard language. The executable file is in this case `myprogram`. The option `-c` is for compilation only, where the program is translated into machine code, while the `-o` option links the produced object file `myprogram.o` and produces the executable `myprogram` .
For Fortran2008 we use the Intel compiler, replace c++ with `ifort`. Also, to speed up the code use compile options like

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

Makefiles and simple scripts

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

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

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

Hello world

The C encounter.

Here we present first the C version.

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

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[])
```

Dissection II.

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

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

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

Hello World

Now in C++.

Here we present first the C++ version.

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

C++ Hello World

Dissection I.

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

Brief summary

C/C++ program.

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

Brief summary

Arrays.

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

Serious problems and representation of numbers

Integer and Real Numbers.

- Overflow
- Underflow
- Roundoff errors
- Loss of precision

Limits, you must declare variables

C++ and Fortran declarations.

type in C/C++ and Fortran2008	bits	range
`int/INTEGER (2)`	16	-32768 to 32767
`unsigned int`	16	0 to 65535
`signed int`	16	-32768 to 32767
`short int`	16	-32768 to 32767
`unsigned short int`	16	0 to 65535
`signed short int`	16	$-$32768 to 32767
`int/long int/INTEGER (4)`	32	-2147483648 to 2147483647
`signed long int`	32	-2147483648 to 2147483647
`float/REAL(4)`	32	3.4×10^{-44} to $3.4 \times 10^{+38}$
`double/REAL(8)`	64	1.7×10^{-322} to $1.7 \times 10^{+308}$
`long double`	64	1.7×10^{-322} to $1.7 \times 10^{+308}$

From decimal to binary representation

How to do it.

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

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

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

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

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

From decimal to binary representation

Integer numbers.

```
using namespace std;
#include <iostream>
int main (int argc, char* argv[])
{
  int i;
  int terms[32]; // storage of a0, a1, etc, up to 32 bits
  int number = atoi(argv[1]);
  // initialise the term a0, a1 etc
  for (i=0; i < 32 ; i++){ terms[i] = 0;}
  for (i=0; i < 32 ; i++){
    terms[i] = number%2;
    number /= 2;

  // write out results
  cout << "Number of bytes used= " << sizeof(number) << endl;
  for (i=0; i < 32 ; i++){
    cout << " Term nr: " << i << "Value= " << terms[i];
    cout << endl;

  return 0;
```

From decimal to binary representation

Integer numbers Fortran.

```
PROGRAM binary_integer
IMPLICIT NONE
  INTEGER  i, number, terms(0:31) ! storage of a0, a1, etc, up to 3

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

END PROGRAM binary_integer
```

Integer Numbers

Possible Overflow for Integers.

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

// End: program main()
```

Loss of Precision

Machine Numbers.

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

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

and a real non-zero number could be generalized as

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

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

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

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

Loss of Precision

Machine Numbers.

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

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

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

Loss of Precision

Machine Numbers.

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

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

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

where the first bit is reserved for the sign, 1 in this case yielding a

Loss of Precision

Machine Numbers.

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

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

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

In this case we have an exact machine representation with 32 bits (actually, we need less than 23 bits for the mantissa).
If our number x can be exactly represented in the machine, we call x a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a

Loss of Precision

Machine Numbers.

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

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

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

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

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

Loss of Precision

Machine Numbers.

In the machine a number is represented as

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

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

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

or

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

meaning that

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

and if $b \approx c$ we see that there is a potential for an increased error

Loss of Precision

Machine Numbers.

Define the absolute error as

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

whereas the relative error is

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

The above subraction is thus

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

yielding

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

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

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

Loss of numerical precision

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

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

while the second expression results in

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

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

Loss of numerical precision

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

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

Loss of precision can cuae serious problems

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

$$x = 1.2345678911121314 68 = 0.12345678911121314 68 \times 10^1 \tag{15}$$

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

More on loss of precision

Real Numbers.

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

A problematic case

Three ways of computing e^{-x}.

Brute force:

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

Recursion relation for

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

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

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

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

Program to compute $\exp(-x)$

Brute Force.

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

Program to compute $\exp(-x)$

Still Brute Force.

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

Program to compute $\exp(-x)$

Oh it never ends!

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

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

        return fac;
    } // End: function factorial()
```

Results $\exp(-x)$

What is going on?

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

Program to compute $\exp(-x)$

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

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

Program to compute $\exp(-x)$

Last statements.

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

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

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

Results $\exp(-x)$

More Problems.

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

Most used formula for derivatives

3 point formulae.

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

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

Second derivative

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

Error Analysis

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

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

For the computed second derivative we have

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

and the truncation or approximation error goes like

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

Error Analysis

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

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

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

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

Error Analysis

Our total error becomes

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

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

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

With double precision and $x = 10$ we obtain

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

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

Error Analysis

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

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

as

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

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

Error Analysis

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

Error Analysis

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

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
10^{-1}	2.010008336111607 0	$1.000833611160723 0 \times 10^{-2}$
10^{-2}	2.000100000833335 8	$1.000008333360558 1 \times 10^{-4}$
10^{-3}	2.000001000000083 6	$1.000000083406504 8 \times 10^{-6}$
10^{-5}	2.000000009999999 9	$1.000000005024759 3 \times 10^{-8}$
10^{-5}	2.000000000100000 0	$9.999989725173463 7 \times 10^{-11}$
10^{-6}	2.000000000001000 1	$9.999778782798785 0 \times 10^{-13}$
10^{-7}	2.000000000000009 8	$9.992007221626408 9 \times 10^{-15}$
10^{-8}	2.000000000000000 0	$0.000000000000000 0 \times 10^{0}$
10^{-9}	2.000000000000000 0	$1.110223024625156 5 \times 10^{-16}$
10^{-10}	2.000000000000000 0	$0.000000000000000 0 \times 10^{0}$

Week 35

Overview of week 35

- Monday: Repetition from last week
- Numerical differentiation
- C/C++ programming details, pointers, read/write to/from file
- Tuesday: Intro to linear Algebra and presentation of project 1.
- Matrices in C++ and Fortran2008
- Gaussian elimination and discussion of project 1.
- Computer-Lab: start project 1. Reading asssignments and preparation for project 1: sections 2.5 and 3.1 for general C++ and Fortran features. Sections 6.1-6.4 (till page 182) are relevant for project 1.

Technical Matter in C/C++: Pointers

A pointer specifies where a value resides in the computer's memory (like a house number specifies where a particular family resides on a street).
A pointer points to an address not to a data container of any kind!
Simple example declarations:

```
using namespace std; // note use of namespace
int main()
{
  // what are the differences?
  int var;
  cin >> var;
  int *p, q;
  int *s, *t;
  int * a new[var];     // dynamic memory allocation
  delete [] a;
}
```

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

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

Dissection: Pointer example I

Discussion.

```c
int main()
{
  int var;      // Define an integer variable var
  int *p;       // Define a pointer to an integer
  p = &var;     // Extract the address of var
  var = 421;    // Change content of var
  printf("Address of integer variable var : %p\n", &var);
  printf("Its value: %d\n", var);  // 421
  printf("Value of integer pointer p : %p\n", p);  // = &var
  // The content of the variable pointed to by p is *p
  printf("The value p points at :  %d\n", *p);
  // Address where the pointer is stored in memory
  printf("Address of the pointer p : %p\n", &p);
  return 0;
}
```

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);
```

Dissection: Pointer example II

```
int matr[2];     // Define integer array with two elements
int *p;          // Define pointer to integer
p = &matr[0];    // Point to the address of the first element in mat
matr[0] = 321;   // Change the first element
matr[1] = 322;   // Change the second element
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);
```

Output of Pointer example II

```
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 f
  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 cont.

```
out_file = fopen( argv[2], "w"); // returns a pointer to the outpu
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

```
#include <fstream>

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

File handling, C++-way

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

File handling, C++-way

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

}  // end of function output
```

File handling, C++-way

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

File handling, C++-way

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

//      Read something from the file:

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

//      Can test on success of reading:

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

Call by value and reference

```
int main(int argc, char argv[]) {
int  a:
int *b;
a = 10;
b = new int[10];
for (i = 0; i < 10; i++) {
  b[i] = i;
}
func(a, b);
delete [] b;
return 0;
}
```

Call by value and reference

Morten: Too complicated LaTeX code for computer code to be decoded....

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` is `a`. The address of `a` is `&a`. The value of `b` is `*b`. The address of `b` is `&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`
- line 7: here we deallocate the variable `b`.

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` is the address of the first element in `b` in `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
  ....
}
```

Call by value and reference, F90/95

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

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

This hinders unwanted changes and increases readability.

Important Matrix and vector handling packages

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

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

Basic Matrix Features

Matrix properties reminder.

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

The inverse of a matrix is defined by

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

Basic Matrix Features

Matrix Properties Reminder.

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

Some famous Matrices

- Diagonal if $a_{ij} = 0$ for $i \neq j$
- Upper triangular if $a_{ij} = 0$ for $i > j$
- Lower triangular if $a_{ij} = 0$ for $i < j$
- Upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$
- Lower Hessenberg if $a_{ij} = 0$ for $i < j + 1$
- Tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$
- Lower banded with bandwidth p: $a_{ij} = 0$ for $i > j + p$
- Upper banded with bandwidth p: $a_{ij} = 0$ for $i < j + p$
- Banded, block upper triangular, block lower triangular....

Basic Matrix Features

Some Equivalent Statements.

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

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

Important Mathematical Operations

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

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

scalar-matrix multiplication

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

vector-matrix multiplication

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

matrix-matrix multiplication

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

Important Mathematical Operations

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

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

scalar-vector multiplication

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

vector-vector multiplication (called Hadamard multiplication)

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

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

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

and the outer product, which yields a matrix,

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

Static.

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

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

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

Matrix Handling in C/C++

Row Major Order Addition.

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

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

In C/C++ this would be coded like

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

Matrix Handling in C/C++

Row Major Order Multiplication.

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

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

In C/C++ this would be coded like

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

Matrix Handling in Fortran 90/95

Column Major Order.

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

Fortran 90 writes the above statements in a much simpler way

```
a=b+c
```

Multiplication

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

Fortran contains also the intrinsic functions TRANSPOSE and CONJUGATE.

Dynamic memory allocation in C/C++

At least three possibilities in this course

- Do it yourself
- Use the functions provided in the library package lib.cpp
- Use Armadillo `http://arma.sourceforgenet` (a C++ linear algebra library, discussion next two weeks, both here and at lab). !split

Matrix Handling in C/C++, Dynamic Allocation

Do it yourself.

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

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

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

Armadillo, recommended!!

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

Armadillo, simple examples

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

using namespace std;
using namespace arma;

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

  cout << A*B << endl;

  return 0;
```

Armadillo, how to compile and install

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

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

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

Armadillo, simple examples

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

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

Armadillo, simple examples

```
mat B;

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

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

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

Armadillo, simple examples

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

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

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

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

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

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

Armadillo, simple examples

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

Armadillo, simple examples

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

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

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

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


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

  return 0;
```

Armadillo, simple examples

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

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

  mat A;

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

  A.print("A =");

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

Armadillo, simple examples

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

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

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


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

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

  return 0;
```

Gaussian Elimination

We start with the linear set of equations

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

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

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

or

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

Gaussian Elimination

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

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

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

$$x_i = \frac{1}{b_{ii}}\left(y_i - \sum_{j=i+1}^{n} b_{ij}x_j\right) \quad i = n-1, n-2, \ldots, 1 \quad (26)$$

Gaussian Elimination

Our actual 4×4 example reads after the first operation

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

or

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

The new coefficients are

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

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

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

with a new right-hand side given by

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

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

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

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

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

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

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

Even though the matrix elements along the diagonal are not zero, numerically small numbers may appear and subsequent divisions may lead to large numbers, which, if added to a small number may

Gaussian Elimination and Tridiagonal matrices, project 1

Suppose we want to solve the following boundary value equation

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

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

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

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

Gaussian Elimination and Tridiagonal matrices, project 1

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

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

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

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

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

Gaussian Elimination and Tridiagonal matrices, project 1

We start with the linear set of equations

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

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

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

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

Gaussian Elimination and Tridiagonal matrices, project 1

We can rewrite as

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

Gaussian Elimination and Tridiagonal matrices, project 1

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

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

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

Gaussian Elimination and Tridiagonal matrices, project 1

In case your system of equations leads to a tridiagonal matrix, it is clearly an overkill to employ Gaussian elimination or the standard LU decomposition. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter 10.

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

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

Gaussian Elimination and Tridiagonal matrices, project 1

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

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

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

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

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \geq |a_k| + |c_k|$ for $k = 2, 3, \dots, n-1$. This is a relevant but not sufficient condition to guarantee that the matrix $\mathbf{A}$ yields a solution to a linear equation problem. The matrix needs also to be irreducible. A

Project 1, hints

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

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

and finally

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

Project 1, hints

We notice the sub-blocks which get repeated

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

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

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

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

Linear Algebra Methods

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

LU Decomposition

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

LU Decomposition, why?

There are at least three main advantages with LU decomposition compared with standard Gaussian elimination:

- It is straightforward to compute the determinant of a matrix
- If we have to solve sets of linear equations with the same matrix but with different vectors **y**, the number of FLOPS is of the order n^3.
- The invers is such an operation !split

LU Decomposition, linear equations

With the LU decomposition it is rather simple to solve a system of linear equations

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

LU Decomposition, linear equations

The previous equation can be calculated in two steps

$$\mathbf{Ly} = \mathbf{w}; \qquad \mathbf{Ux} = \mathbf{y}.$$

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as

$$\mathbf{LUx} = \mathbf{w},$$

and since the determinat of $\mathbf{L}$ is equal to 1 (by construction since the diagonals of $\mathbf{L}$ equal 1) we can use the inverse of $\mathbf{L}$ to obtain

$$\mathbf{Ux} = \mathbf{L}^{-1}\mathbf{w} = \mathbf{y},$$

which yields the intermediate step

$$\mathbf{L}^{-1}\mathbf{w} = \mathbf{y}$$

and as soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{Ux} = \mathbf{y}$.

LU Decomposition, why?

For our four-dimentional example this takes the form

$$\begin{aligned} y_1 &= w_1 \\ l_{21}y_1 + y_2 &= w_2 \\ l_{31}y_1 + l_{32}y_2 + y_3 &= w_3 \\ l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= w_4. \end{aligned}$$

and

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

This example shows the basis for the algorithm needed to solve the set of n linear equations.

LU Decomposition, linear equations

The algorithm goes as follows

- Set up the matrix **A** and the vector **w** with their correct dimensions. This determines the dimensionality of the unknown vector **x**.
- Then LU decompose the matrix **A** through a call to the function `ludcmp(double a, int n, int indx, double &d)`. This functions returns the LU decomposed matrix **A**, its determinant and the vector indx which keeps track of the number of interchanges of rows. If the determinant is zero, the solution is malconditioned.
- Thereafter you call the function `lubksb(double a, int n, int indx, double w)` which uses the LU decomposed matrix **A** and the vector **w** and returns **x** in the same place as **w**. Upon exit the original content in **w** is destroyed. If you wish to keep this information, you should make a backup of it in your calling function.

LU Decomposition, the inverse of a matrix

If the inverse exists then

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

the identity matrix. With an LU decomposed matrix we can rewrite the last equation as

$$\mathbf{LUA}^{-1} = \mathbf{I}.$$

If we assume that the first column (that is column 1) of the inverse matrix can be written as a vector with unknown entries

$$\mathbf{A}_1^{-1} = \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix},$$

then we have a linear set of equations

LU Decomposition, the inverse

In a similar way we can compute the unknow entries of the second column,

$$\mathbf{LU}\begin{pmatrix} a_{12}^{-1} \\ a_{22}^{-1} \\ \dots \\ a_{n2}^{-1} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ \dots \\ 0 \end{pmatrix},$$

and continue till we have solved all n sets of linear equations.

How to use the Library functions

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

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

using namespace std;

/* function declarations */

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

How to use the Library functions

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

How to use the Library functions

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

  free_matrix((void **) y);      // release local memory
  delete [] col;
  delete []indx;
}  // End: function inverse()
```

How to use the Library functions

For Fortran users:

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

How to use the Library functions

For Fortran users:

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

Week 36

Overview of week 36

Linear Algebra.

- Monday: Repetition from last week
- Discussion of Project 1, deadline 16 September (noon).
- Gaussian elimination, LU decomposition and linear equations
- Inverse of a matrix.
- Tuesday: Further discussion of linear algebra methods.
- Dynamic memory allocation in C/C++ and Fortran2008, use of the libraries like Armadillo for C++ users. How to use the C/C++ and Fortran 90/95 libraries. Programming classes in C++.
- Discussion of version control. Dropbox and GIT.
- Computer-Lab: Project 1.

Object orientation

Why object orientation?

- Three main topics: objects, class hierarchies and polymorphism
- The aim here is to be to be able to write a more general code which can easily be tailored to new situations.
- **Polymorphism** is a term used in software development to describe a variety of techniques employed by programmers to create flexible and reusable software components. The term is Greek and it loosely translates to "many forms". Strategy: try to single out the variables needed to describe a given system and those needed to describe a given solver. !split

Object orientation

In programming languages, a polymorphic object is an entity, such as a variable or a procedure, that can hold or operate on values of differing types during the program's execution. Because a polymorphic object can operate on a variety of values and types, it can also be used in a variety of programs, sometimes with little or

Programming classes

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

Programming classes

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

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

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

Programming classes

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

To achieve this we need to introduce two important entities in C++ programming, classes and templates.

Programming classes

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

Programming classes

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

Programming classes

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

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

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

Programming classes

We proceed by splitting our task in three files.

We define first a header file complex.h which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

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

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

Programming classes

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

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

Programming classes

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

Programming classes

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + i1.3$, $b = 3.0 + i0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - i2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

Programming classes

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

Programming classes

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

Programming classes

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

Programming classes

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

Programming classes

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

Programming classes

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

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

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

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

Programming classes

Another possibility is

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

which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.

A call like `Complex a(0.1,1.3);` means that we could call the member function 'Complex(double, double)'as

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

Programming classes

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

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

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

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

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

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

Programming classes

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

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

without the constant declaration and define thereafter a function `myabs` as

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

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

Programming classes

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

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

Programming classes

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

Programming classes

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

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

Programming classes

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

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

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

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

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

The compiler translates `c = a + b;` into `c = operator+ (a, b);`. Since this implies the call to function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop.

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

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

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

Programming classes

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

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

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

Programming classes

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

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

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

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

Programming classes

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

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

Programming classes

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

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

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

Programming classes

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

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

Programming classes

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

The definitions `operator-` and `operator/` follow the same set up.

Programming classes

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

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

Programming classes, templates

What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace `double` with for example `int`.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword `template` followed by a list of template arguments enclosed in brackets.

Programming classes

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

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

Programming classes

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

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

Programming classes

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

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

Programming classes

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

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

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

Programming classes

Here follows a very simple first class

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

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

};
```

Programming classes

and we would use it as

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

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