Computational Astrophysics I: Introduction and basic concepts

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

SoSe 2020
Conventions of used fonts

Meaning of the fonts / shapes

<table>
<thead>
<tr>
<th>font/shape</th>
<th>meaning</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xvzf</code></td>
<td>text to be entered literally</td>
<td><code>man ls</code></td>
</tr>
<tr>
<td>(typewriter)</td>
<td>(e.g., commands)</td>
<td></td>
</tr>
<tr>
<td><code>argument</code></td>
<td>place holder for own text</td>
<td><code>file myfile</code></td>
</tr>
<tr>
<td>(italic)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aims and contents I

- enhance existing basic knowledge in programming (C/C++)
- brief introduction to Fortran → relatively common in astrophysics
- work on astrophysical topics which require computer modeling:
  - solving ordinary differential equations
    → from the two-body problem to $N$-body simulations
    → stellar structure, the Lane-Emden equation
  - solving equations: linear algebra, root finding, data fitting
  - data analysis
    → data analysis and simulations
  - simulation of physical processes
    → Monte-Carlo simulations and radiative transfer
- introduction to parallelization (e.g., OpenMP)
Aims and contents II

Recommended prerequisites:

- basic knowledge of programming, especially in C/C++
- basic knowledge in astrophysics

How to get a certificate of attendance / 6 CP/LP/ECTS (=4 semester periods per week):

- without mark, e.g., Master of Astrophysics, module PHY-765: Topics in Advanced Astrophysics (this module has in total 12 CP!):
  → at least 1./3. of the points of the exercises

- with a mark (other Master courses):
  little programming project at the end of the semester

Please, note that the focus for this course is on the practical part!
What are computers used for in astrophysics?

- control of instruments/telescopes/satellites:

Figure: MUSE, VLA, HST
data analysis / data reduction

Figure: IDL, 3dCube / FITS, Fourier analysis
modeling / numerical simulations

Figure: N-body simulation, hydrodynamics, Monte-Carlo
User accounts

useful for the lecture: your own account for this computer lab (room 0.087 & 1.100)

Please, get your own account!
Sysad: Helge Todt, room 2.004

Guest account

→ see left-hand side whiteboard
only valid per computer and in room 0.087

Attention: Unix/Linux is case sensitive!
Hint: You can choose the session type (e.g., Xfce, IceWM) at login screen.
Security advice
As soon as you got your own account:

```
passwd  Change the user password
```

(Enter the command in a terminal, Xterm, Konsole, or similar.)

Change your initial NIS password(!) to a **strong** password, use
- at least 9 characters, comprising of:
  - capital AND lowercase letters, but not single words
  - AND numbers
  - AND special characters (Attention! Mind the keyboard layout!)
e.g., $cPhT-25@comP2  or tea4Pollen+Ahead

The initial password expires after 14 days.
The computer lab III

Computers:

- 17 NFS\(^1\) mounted Linux computers (openSuSE 42.3/15.0), several Intel Core i7-2600K, i7-4770, i7-7700, i7-8700 + 1 Xeon Gold 6152 44-core compute server
- home server (~user) always-on:
  - bell
  - mahler
  - weber

room 0.087:

- only for lectures
- Please, do not eat or drink in this room.

student’s computer lab in room 1.100:

- open during the day
- b/w printer (500 pages / semester) and color printer (100 pages / semester)
NFS server

NFS client  NFS client

NFS server: provides (home) directories (physical on disk)

NFS clients: mount NFS (home) directories in their root directory

As also other users might have their home directory on your computer

Never switch off the computers!
Linux
Linux is a derivative of the operating system UNIX. It is a multi-user and multitasking operating system.

It was written in 1991 as a UNIX for PCs, now available for (almost) every platform, e.g., as Android or in Wireless routers and under permanent development.

Linux is . . .

- for free
- open source (program code can also be modified)
- the combination of a *monolithic* kernel and (GNU) software
- dominant in supercomputers (more than 90%)
Important X-Window based environments under Linux: GNOME and KDE, here: Xfce

Desktop environment (session type) can be chosen during local login, e.g., Xfce (nice) or IceWM (simple)

<table>
<thead>
<tr>
<th>Desktop environment</th>
<th>Linux distributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>KDE</td>
<td>Ubuntu (Debian)</td>
</tr>
<tr>
<td>Xfce</td>
<td>openSuSE</td>
</tr>
<tr>
<td>GNOME</td>
<td>...</td>
</tr>
</tbody>
</table>
xterm or terminal: input of Linux shell commands, e.g., `cd`, `ls`

emacs or kate: editor for ASCII text files, e.g., `hello.cpp`

g++ or gfortran: gcc compilers, e.g., `g++ -o hello hello.cpp`
Shell and shell commands
Unix provides by the *shell* (command line) an extremely powerful tool. Within the shell Unix commands are executed.

**Unix command syntax**

`command [-option] [argument] <ENTER>`

Attention! Mind the blanks!

Open an xterm or similar terminal/console and enter following: (finish each line with `<ENTER>`):

`echo hello`
and

`echo -n hello`

What’s the difference?
Command history

By ↑ (arrow key up) you can repeat the last commands entered in the shell.

A list of the last commands can be shown via the command `history`

Moreover, you can save typing by using the TAB key, it completes commands or file names:

```
ec <TAB>
```

is completed to

```
echo
```
Directories
The filesystem tree

/  → root of the FS tree
  
  |  → Home directories
  |  
  |-- /home/  
  |    
  |    |  → Homes on weber
  |    |  
  |    |-- /home/weber/  
  |    |    
  |    |    |  → Helge’s home
  |    |    |  
  |    |    |-- /home/weber/htodt/  
  |    |    
  |    |-- /etc/  
  |    
  |    
  |-- /dev/  
  
  → et cetera (config)
  
  → devices
**Navigation through directories**

- `pwd` shows the current directory path (absolute)
e.g., `/home/weber/htodt`

- `cd name` change to directory `name`

- `.` means the current directory

- `..` the parent directory, e.g., `cd ..`

- `/` root of the FS tree

- `~` the home directory, e.g., `cd ~` or just `cd`

- `~user` the home directory of `user`
mkdir name create directory name

rmdir name remove directory name

ls show (list) the content of the directory
Navigation through directories III

**`ls`** show the content of the current directory

**`ls -a`** also show hidden files (starting with a `. `)

**`ls -l`** show the file attributes, owner, creation time

---

**File attributes**

```
```

- `d` = directory
- `r` = readable
- `w` = writeable
- `x` = executable
- `htodt` = owner
- `users` = group
- `4096` = size in byte
- `14. Oct 13:35` = creation time
- `Documents` = name of the file (here: of the directory)

**Hint:** `ls -lc`  → time of last modification ; `ls -lu`  → time of last access
man ls

Manual pages (help for the command ls)

info ls

Info pages (alternative help for the command ls)

ls --help

Help for the command ls

ls --help | less
if more than one screen page

---

**man page navigation – also less, more**

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>quit</td>
</tr>
<tr>
<td>&lt;SPACE&gt;</td>
<td>next page</td>
</tr>
<tr>
<td>/</td>
<td>forward search</td>
</tr>
<tr>
<td>n</td>
<td>next occurrence</td>
</tr>
<tr>
<td>&gt;</td>
<td>jump to the end</td>
</tr>
</tbody>
</table>
Copying text with the mouse

Linux: Copy by Selection

mark the text with the mouse:
- press left mouse button, keep it pressed
- move mouse cursor until end of the region you want to mark
  → marked region will be highlighted

marked text was copied to the *clipboard*

paste the copied text:
- move cursor to the intended position
- press the middle(!) mouse button (or wheel)
  → the previously copied text was inserted
→ to create pure ASCII files (e.g., as input for g++)

```
emacs file &
```

Starting programs in background:

The ampersand `&` at the end of a command let the command run in the background (bg) of the shell.

Hence, the input line of the shell can be still used.

If forgotten: `<CTRL>+z` followed by `bg <ENTER>`. 
Remark: In Linux almost everything is a file (also directories and devices, see `ls -l /dev/`).

- `mv source target` move (rename) files
- `cp source target` copy files
- `rm file` remove file
- `rm -rf directory` remove directory
Compress files and directories

**tar action archive files**  
use *action* on *archive*

**tar actions**

- **c**: create archive from file/directory
- **x**: extract archive
- **v**: show executed actions (*verbose*)
- **z**: zipp archive
- **t**: show content of archive
- **f**: archive is a file (default: tape device)

**Example: Untar a tarball**

```
 tar xvzf muCommander.tar.gz
```
Connecting to other hosts (computers)
hostname

this command shows the name of the host you’re currently logged in
Connection to another host (remote host) under Linux/Unix with the secure shell, within the same domain (e.g., within the computer lab cluster)

```
ssh host name
```

After successful login, in the same terminal/window a shell is shown that runs on the remote host.
The SecureSHell

Client-server system for establishing a secure connection (encrypted), login on the remote host
(remote host = SSH server)

if SSH client and SSH server support X11:

    ssh host name -Y

allows the SSH server to open a graphical window (e.g., for evince or kate) on the SSH client

Besides the interactive use of the SSH one can also just let a program run on the remote host via ssh:

    ssh hostname "ls -l"
The connection will be automatically closed after program/command is finished.
Login from outside (e.g., from home):

```
ssh username@bell.stud.physik.uni-potsdam.de
```

There are SSH clients for Windows that are for free, e.g., PuTTY. Moreover, MobaXterm, Xming, X2Go (requires also installation on the server) or with help of the Windows Subsystem for Linux (requires the installation of Linux distribution) it is also possible to perform a graphical SSH login from Windows to Unix/Linux.

**Hint:**
With help of the graphical login you can, e.g., use Mathematica on the computer lab cluster at home.
For login without password:

1. run `ssh-keygen` on the client, answer all question just with <ENTER>
2. add the resulting `~/.ssh/id_rsa.pub` from the client host to `~/.ssh/authorized_keys` on the remote host
With help of the SSH protocol it is also possible to transfer files between different computers:

```
scp document.txt username@bell.stud.physik.uni-potsdam.de:
   secure copy to the remote host
scp username@bell.stud.physik.uni-potsdam.de:document.txt .
   secure copy from the remote host (mind the dot!)
```

After the colon `:` is a path given, either absolute or relative to the home directory.
Remote copy II

To copy only files that have been modified (comparison of source and target):

```
rsync -rtvz username@host.domain:directory/ .
```

secure copy from the remote host, only modified files

some useful options:
- `r` recursive: also directories
- `t` time: keep time stamps of transferred files
- `v` verbose: print information during transfer
- `z` zip: compressed file transfer (faster for slow connections)
- `c` checksum: use check sums (instead of time stamps) for comparison
Copy files via konqueror from other hosts

konqueror allows to show directories of remote hosts with help of the fish protocol. So, enter in the address bar, e.g.,

fish://user@weber.stud.physik.uni-potsdam.de
More Unix commands

df -h  shows free space on hard drive

du -hs  shows total size of current directory

ps ux  shows running processes of the current user

top  shows load and running processes (interactive)

htop

kill -9 PID  “kills” the process with the given process ID (PID)
One can, e.g., distinguish:

scripting languages
- bash, csh → Unix shell
- Perl, Python
- IRAF, IDL, Midas → especially for data reduction in astrophysics

compiler-level languages
- C/C++ → very common, therefore our favorite language
- Fortran → very common in astrophysics, especially in radiative transfer
## Programming languages II

<table>
<thead>
<tr>
<th>scripting language</th>
<th>compiler-level language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>examples</strong></td>
<td>C/C++, Fortran, Pascal, ...</td>
</tr>
<tr>
<td>shell (bash, tcsh), Perl, Mathematica, MATLAB, ...</td>
<td></td>
</tr>
<tr>
<td><strong>source code</strong></td>
<td>translated to machine code, e.g., 0x90 → no operation (NOP)</td>
</tr>
<tr>
<td>directly executable</td>
<td></td>
</tr>
<tr>
<td><strong>runtime behavior</strong></td>
<td>error handling difficult</td>
</tr>
<tr>
<td>interpreter runs as a program → full control over execution → error messages, argument testing</td>
<td>→ task of the programmer, often only crash</td>
</tr>
<tr>
<td><strong>speed</strong></td>
<td>very fast by optimization</td>
</tr>
<tr>
<td>usually slow → analysis tools</td>
<td>→ simulations, number crunching</td>
</tr>
</tbody>
</table>

→ moreover, also bytecode compiler (JAVA) for virtual machine, Just-in-time (JIT) compiler (JavaScript, Perl)
C is a *procedural* (imperative) language

C++ is an *object oriented* extension of C with the same syntax

C++ is because of its additional structures (template, class) \(\Rightarrow\) C

---

**Basic structure of a C++ program**

```cpp
#include <iostream>
using namespace std;
int main () {
    instructions of the program;
    // comment
    return 0;
}
```

every instruction must be finished with a `;` (semicolon)!
Compiling a C++ program:

source file
.cpp, .C

compiler + linker
.o, .so, .a

executable program
.a.out, program
Command for compiling + linking:

```bash
g++ -o program program.cpp
```

(GNU compiler for C++)

- only compiling, do not link:
  ```bash
g++ -c program.cpp
```
  creates `program.o` (object file, not executable)

- option `-o name` defines a name for a file that contains the executable program, otherwise program file is called: `a.out`
  
the name of the executable program can be arbitrarily chosen
Simple program for output on screen

Example: C++ output via streams

```cpp
#include <iostream>

using namespace ::std ;

int main () {
    cout << endl << "Hello world!" << endl ;
    return 0 ; // all correct
}
```
Simple program for output on screen II

- `<iostream>` ... is a C++ library (input/output)
- `main()` ... program (function)
- `return 0` ... returns the return value 0 to main (all ok)
- Source code can be freely formatted, i.e., it can contain an arbitrary number of spaces and empty lines (white space) → useful for visual structuring
- Comments are started with `//` - everything after it (in the same line) is ignored,
  C has only `/*` comment `*/` for comment blocks
- `cout` ... output on screen/terminal (C++)
- `<<` ... output/concatenate operator (C++)
- `string "Hello world!"` must be set in quotation marks
- `endl` ... manipulator: new line and stream flush (C++)
- A block several instructions which are hold together by curly braces
C/C++ is a procedural language
The procedures of C/C++ are *functions*.

- Main program: function with specific name `main(){}
- every function has a type (for return), e.g.: `int main (){}
- functions can get arguments by call, e.g.:
  `int main (int argc, char *argv[]){}
- functions must be *declared before* they can be called in the main program,
  e.g., `void swap(int &a, int &b) ;
  or included via a header file:
  `#include <cmath>
- within the curly braces `{ }`, the so-called function body, is the *definition* of the function (what shall be done how), e.g.:
  `int main () { return 0 ; }`
Example

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

float cube(float x);

int main() {
    float x = 4.;
    cout << cube(x) << endl;
    return 0;
}

float cube(float x) {
    return x*x*x;
}
```
A variable is a piece of memory.
in C/C++ data types are explicit and static

We distinguish regarding visibility ("scope"):
- global variables → declared outside of any function, before main
- local variables → declared in a function or in a block { }, only there visible

...regarding data types → intrinsic data types:
- int → integer, e.g., int n = 3 ;
- float → floats (floating point numbers), e.g., float x = 3.14, y = 1.2E-4 ;
- char → characters, e.g., char a_character ;
- bool → logical (boolean) variables, e.g., bool btest = true ;
Integer numbers are represented exactly in the memory with help of the binary number system (base 2), e.g.

\[ 13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \equiv 1101_2 \quad \text{(binary)} \]

In the assignment

\[ a = 3 \]

3 is an integer literal (literal constant). Its bit pattern \( (3 = 1 \cdot 2^0 + 1 \cdot 2^1 \equiv 11_2) \) is inserted at the corresponding positions by the compiler.

\(^1\)doesn’t correspond necessarily to the sequential order used by the computer \( \rightarrow \) “Little Endian”: store least significant bit first, so actually: 1011
Integer data types II

on 64-bit systems

\textbf{int} \quad \text{compiler reserves 32 bit (}= 4 \text{ byte}) \text{ of memory}

1 \text{ bit for sign and}

\[ 2^{31} = 2,147,483,648 \text{ values (incl. 0)}: \rightarrow \text{range:} \]

\[ \text{int} = -2,147,483,648 \ldots + 2,147,483,647 \]

\textbf{unsigned int} \quad 32 \text{ bit, no bit for sign} \rightarrow 2^{32} \text{ values (incl. 0)}

\[ \text{unsigned int} = 0 \ldots 4,294,967,295 \]

\textbf{long} \quad \text{on 64 bit systems: 64 bit (}= 8 \text{ byte}),

1 \text{ bit for sign: } -9.2 \times 10^{18} \ldots 9.2 \times 10^{18} \text{ (quintillions)}

\textbf{unsigned long} \quad 64 \text{ bit without sign: } 0 \ldots 1.8 \times 10^{19}

and also: \textbf{char} (1 \text{ byte}), \text{smallest addressable (!)}; \textbf{short} (2 \text{ byte}); \textbf{long long} (8 \text{ bytes})
Two’s complement

Table: Representation: unsigned value (0s), value and sign (sig), two’s complement (2’S) for a nibble (1/2 byte)

<table>
<thead>
<tr>
<th>binary</th>
<th>0s</th>
<th>sig</th>
<th>2’S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-0</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-1</td>
<td>-7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-7</td>
<td>-1</td>
</tr>
</tbody>
</table>

Disadvantages of representation as value and sign:
∃ 0 and -0; Which bit is sign?
(→ const number of digits, fill up with 0s);

Advantage of 2’S:
negative numbers always with highest bit=1
→ cf. $+1 + (-1)$ bitwise for value & sign vs. 2’S
Floating point numbers are an approximate representation of real numbers. Floating point numbers can be declared via, e.g.,:

```c
float radius, pi, euler, x, y;
double radius, z;
```

Valid assignments are, e.g.,:

```c
x = 3.0;
y = 1.1E-3;
z = x / y;
```
Floating point data types II

- representation (normalization) of floating point numbers are described by standard IEEE 754:

\[ x = s \cdot m \cdot b^e \]  

(1)

with base \( b = 2 \) (IBM Power6: also \( b = 10 \)), sign \( s \), and normalized significand (mantissa) \( m \), bias

- So for 32 Bit (Little ENDIAN\(^\dagger\)), 8 bit exponent, 23 bit mantissa:


31 24 23 16 15 8 7 0

exponent mantissa

sign

\(\dagger\) read each part: \(\rightarrow\)
• mantissa is *normalized* to the form (e.g.)
  \[ 1,0100100 \times 2^4 \]
i.e. with a 1 before the decimal point. This 1 is not stored, so \( m = 1.f \)

Moreover, a bias (127 for 32 bit, 1023 for 64 bit) is added to the exponent (results in non-negative integer)

**Example: Conversion of a decimal number to IEEE-32-Bit**

172.625 base 10
10101100.101 \( \times 2^0 \) base 2
1.0101100101 \( \times 2^7 \) base 2 normalized

add bias of 127 to exponent = 134 = \( 1 \cdot 2^7 + \ldots + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \)

0 10000110 01011001010000000000000000
• single precision (32 bit) **float**: exponent 8 bit, significand 23 bit
  
  \[-126 \leq e \leq 127 \text{ (basis 2)}\]
  
  \[\rightarrow \approx 10^{-45} \ldots 10^{38}\]

  **digits: 7-8** \((= \log 2^{23+1} = 24 \log 2)\)

• for 64 bit (double precision) – **double**: exponent 11 bit, significand 52 bit

  \[-1022 \leq e \leq 1023 \text{ (basis 2)}\]

  \[\rightarrow \approx 10^{-324} \ldots 10^{308}\]

  **digits: 15-16** \((= \log 2^{52+1})\)
some real numbers cannot be presented exactly in the binary numeral system (cf. 1/3 in decimal):

\[ 0.1 \approx 1.10011001100110011001101 \times 2^{-4} \] (2)

**Warning**

Do not compare two floating point numbers blindly for equality (e.g., 0.362 * 100.0 == 36.2), but rather use an accuracy limit:

- \( \text{abs}(x - y) \leq \text{eps} \)
- better: relative error \( \text{abs}(1-y/x) \leq \text{eps} \)
Floating point data types VI

Floating point arithmetics

Subtraction of floating point numbers

consider $1.000 \times 2^5 - 1.001 \times 2^1$ (only 3 bit mantissa)

$\rightarrow$ bitwise subtraction, requires same exponent

\[
\begin{array}{ccc}
1.000 & 0000 & \times 2^5 \\
- & 0.000 & 1001 \times 2^5 \\
\hline
0.111 & 0111 \times 2^5 & \text{infinite precision} \\
1.110 & 111 & \times 2^4 & \text{shifted left to normalize} \\
1.111 & & \times 2^4 & \text{rounded up, as last digits \(> \frac{1}{2} \text{ ULP}^{\dagger}\}}
\end{array}
\]

$\dagger$ unit in the last place = spacing between subsequent floating point numbers
Floating point data types VII

Properties of floating point arithmetics (limited precision):

- loss of significance / catastrophic cancellation: occurs for subtraction of almost equal numbers

Example for loss of significance

\[ \pi - 3.141 = 3.14159265 \ldots - 3.141 \text{ with 4-digit mantissa; maybe expected: } = 0.00059265 \ldots \approx 5.927 \times 10^{-4}; \text{ in fact: } 1.0000 \times 10^{-3}, \]

because \( \pi \) is already rounded to 3.142

- absorption (numbers of different order of magnitude): addition of subtraction of a very small number does not change the larger number

Example for absorption

for 4-digit mantissa: \( 0.001 + 100 = 100: 1.000 \times 10^2 + 1.000 \times 10^{-3} = 1.000 \times 10^2 + 0.00001 \times 10^2 = 1.000 \times 10^2 + 0.000 \times 10^2 = 1.000 \times 10^2 \), same for subtraction
Floating point data types VIII

- distributive and associative law usually not fulfilled, i.e. in general

\[(x + y) + z \neq x + (y + z)\]  \hspace{1cm} (3)

\[(x \cdot y) \cdot z \neq x \cdot (y \cdot z)\]  \hspace{1cm} (4)

\[x \cdot (y + z) \neq (x \cdot y) + (x \cdot z)\]  \hspace{1cm} (5)

\[(x + y) \cdot z \neq (x \cdot z) + (y \cdot z)\]  \hspace{1cm} (6)

- solution of equations, e.g., \((1 + x) = 1\) for 4-bit mantissa solved by any \(x < 10^{-4}\) (see absorption) \(\rightarrow\) smallest float number \(\epsilon\) with \(1 + \epsilon > 1\) called machine precision

Multiplication and division of floating point numbers:
mantissas multiplied/divided, exponents added/subtracted
\(\rightarrow\) no cancellation or absorption problem
Guard bit, round bit, sticky bit (GRS)

- in floating point arithmetics: if mantissa shifted right → loss of digits
- therefore: during calculation 3 extra bits (GRS)
  - Guard bit: 1st bit, just extended precision
  - Round bit: 2nd (Guard) bit, just extended precision (same as G)
  - Sticky bit: 3rd bit, set to 1, if any bit beyond the Guard bits non-zero, stays then 1(!) → sticky

- example

| Before 1st shift: 1.11000000000000000000100 0 0 0 |
| After 1 shift: 0.11100000000000000000010 0 0 0 |
| After 2 shifts: 0.01110000000000000000001 0 0 0 |
| After 3 shifts: 0.00111000000000000000000 1 0 0 |
| After 4 shifts: 0.00011100000000000000000 0 1 0 |
| After 5 shifts: 0.00001110000000000000000 0 0 1 |
| After 6 shifts: 0.00000111000000000000000 0 0 1 |
| After 7 shifts: 0.00000011100000000000000 0 0 1 |
| After 8 shifts: 0.00000001110000000000000 0 0 1 |
GRS bits – possible values and stored values

<table>
<thead>
<tr>
<th>extended sum</th>
<th>stored value</th>
<th>why</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0100 000</td>
<td>1.0100</td>
<td>truncated because of GR bits</td>
</tr>
<tr>
<td>1.0100 001</td>
<td>1.0100</td>
<td>truncated because of GR bits</td>
</tr>
<tr>
<td>1.0100 010</td>
<td>1.0100</td>
<td>rounded down because of GR bits</td>
</tr>
<tr>
<td>1.0100 011</td>
<td>1.0100</td>
<td>rounded down because of GR bits</td>
</tr>
<tr>
<td>1.0100 100</td>
<td>1.0100</td>
<td>rounded down because of S bit</td>
</tr>
<tr>
<td>1.0100 101</td>
<td>1.0101</td>
<td>rounded up because of S bit</td>
</tr>
<tr>
<td>1.0100 110</td>
<td>1.0101</td>
<td>rounded up because of GR bits</td>
</tr>
<tr>
<td>1.0100 111</td>
<td>1.0101</td>
<td>rounded up because of GR bits</td>
</tr>
</tbody>
</table>
IEEE representation of 32 bit floats:

<table>
<thead>
<tr>
<th>Number name</th>
<th>sign, exp., f</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>$0 &lt; e &lt; 255$</td>
<td>$(-1)^s \times 2^{e-127} \times 1.f$</td>
</tr>
<tr>
<td>subnormal</td>
<td>$e = 0, f \neq 0$</td>
<td>$(-1)^s \times 2^{-126} \times 0.f$</td>
</tr>
<tr>
<td>signed zero ($\pm 0$)</td>
<td>$e = 0, f = 0$</td>
<td>$(-1)^s \times 0.0$</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$s = 0, e = 255, f = 0$</td>
<td>+INF</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$s = 1, e = 255, f = 0$</td>
<td>-INF</td>
</tr>
<tr>
<td>Not a number</td>
<td>$e = 255, f \neq 0$</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- if float $> 2^{128}$ → overflow, result may be NaN or unpredictable
- if float $< 2^{-128}$ → underflow, result is set to 0

If not default by compiler: enable floating-point exception handling (e.g., `-fpe-all0` for ifort)
In C/C++ many data type conversions are already predefined, which will be invoked automatically:

```c
int main () {
    int a = 3;
    double b;
    b = a; // implicit conversion of a to double
    b = 1. / 3; // implicit conversion of 3 to double
    return 0.2; // implicit conversion of 0.2 to integer 0
}
```
Moreover, a type conversion/casting can be done explicitly:

```c
int main () {
    int a = 3;
    double b;
    b = (double) a; // type cast
    return 0;
}
```

- obviously possible: integer $\leftrightarrow$ floating point
- but also: pointer (see below) $\leftrightarrow$ data types
- Caution: For such C casts there is no type checking during runtime!
The better way: use the functions of the same name for type conversion

```c
int i, k = 3;
float x = 1.5, y;
i = int(x) + k;
y = float(i) + x;
```
Logical variables

```cpp
class Logical variables

bool b;

intrinsically data type, has effectively only two different values:

bool btest, bdo;
btest = true; // = 1
bdo = false; // = 0

but also:

btest = 0. ; // = false
btest = -1.3E-5 ; // = true

Output via cout yields 0 or 1 respectively. By using cout << boolalpha << b ; is also possible to obtain t and f for output.

Note: minimum addressable piece of memory is 1 byte → bool needs more memory than necessary
```
Executable control constructs modify the program execution by selecting a block for repetition (loops, e.g., for) or branching to another statement (conditional, e.g., if/unconditional, e.g., goto).

Repeated execution of an instruction/block:

```cpp
for (int k = 0 ; k < 6 ; ++k ) sum = sum + 7 ;

for (float x = 0.7 ; x < 17.2 ; x = x + 0.3) {
    y = a * x + b ;
    cout << x << " " << y << endl;  
}
```
Execution control - for-loops II

Structure of the loop control (header) of the for loop:

There are (up to) three arguments, separated by semicolons:

1. **initialization of the loop variable (loop counter), if necessary with declaration**, e.g.:
   
   ```
   int k = 0 ;
   ```

   → **is executed before the first iteration**

2. **condition for termination of the loop**, usually via arithmetic comparison of the loop variable, e.g.,

   ```
   k < 10 ;
   ```

   is **tested before each iteration**

3. **expression**: incrementing/decrementing of the loop variable, e.g.,

   ```
   ++k or --k or k += 3
   ```

   is **executed after each iteration**

† interestingly also: `int k = 0, j = 1;`
Increment operators

\[
\text{sum } += \ a \\
\rightarrow \text{sum } = \text{sum } + \ a
\]

\[
++x \\
\rightarrow x = x + 1 \text{ (increment operator)}
\]

\[
--x \\
\rightarrow x = x - 1 \text{ (decrement operator)}
\]

Note that there is also a post increment/decrement operator: \(x++\), \(x--\), i.e. incrementing/decrementing is done after any assignment, e.g., \(y = x++\).
Logical operators I - Comparisons/inequalities

→ return either(!) true or false:

\[
\begin{align*}
    a > b & \quad \text{greater than} \\
    a \geq b & \quad \text{greater than or equal} \\
    a = b & \quad \text{equal} \\
    a \neq b & \quad \text{not equal} \\
    a \leq b & \quad \text{less than or equal} \\
    a < b & \quad \text{less than}
\end{align*}
\]

Caution!

The exact equality == should not be used for float-type variables because of the limited precision in the representation.
Moreover, there exist also:

**while loops**

```c
while (x < 0.) x = x + 2. ;

do x = x + 2. ; // do loop is executed
while (x < 0.) ; // at least once!
```

**Instructions for loop control**

```c
break ; // stop loop execution / exit current loop
continue ; // jump to next iteration
```
In C/C++: no real “for loops”

→ loop variable (counter, limits) can be changed in loop body
slow, harder to optimize for compiler/processor

Recommendation: *local* loop variables

→ declaration in loop header
→ scope limited to loop body
Conditional execution via if:

\[
\text{if } (z \neq 1.0) \text{ } k = k + 1;
\]

Conditional/branching

\[
\text{if } (a == 0) \text{ cout } << \text{ "result" } ; \text{ // one-liner}
\]

\[
\text{if } (a == 0) a = x2 ; \text{ // branching}
\]

\[
\text{else if } (a > 1) \{ \text{ a = x1 ; } \}
\]

\[
\text{else a = x3 ;}
\]
If the variable used for branching has only discrete values (e.g., int, char, but not floats!), it is possible to formulate conditional statements via `switch/case`:

```java
Branching II
switch (Ausdruck) {
    case value1 : instruction ; break ;
    case value2 : instruction1 ;
        instruction2 ; break ;
    default : instruction ;
}
```

**Heads up!**
Every case instruction section should be finished with a `break`, otherwise the next case instruction section will be executed automatically.
Example: switch

```cpp
int k;
cout << "Please enter number, 0 or 1: " ;
cin >> k ;
switch (k) {
    case 0 : cout << "pessimist" << endl ; break ;
    case 1 : cout << "optimist" << endl ; break ;
    default : cout << "neutral" << endl ;
}
```
Declarations of variables should be at the beginning of a block, exception: loop variables

```c
float x, y; // declaration of x and y
int n = 3; // declaration and initialization of n
```

Local variables / variables in general
- are only visible within the block (e.g., in `int main() { }`), where they have been declared
- are local regarding this block, their value can only be changed within this block
- are unknown outside of this block, i.e., they don’t exist there
Global variables

- must be declared outside of any function, e.g., before `main()`
- are visible/known to all following functions within the same program
- have file wide visibility (i.e., if you split your source code into different files, you have to put the declaration into every file)
- are only removed from memory when execution of the program is ended

A locally declared variable will hide a global variable of the same name. The global variable can be still accessed with help of the scope operator `::`, e.g., `cout << ::m ;`
Global and local variables

```c
int m = 0; // global variable

void calc() {
    int k = 0; // local variable
    m = 1; // ok, global variable
    ++j; // error, as j only known in main
}

int main() {
    int j = 3;
    ++j; // ok
    for (int i = 1; i < 10; ++i) {
        j = m + i; // ok, all visible
    }
    m = j - i; // error: i not visible
    return j;
}
```
Values (e.g., numbers) that do not change during the program execution, should be *defined* as constants:

```cpp
const float e = 2.71828;
```

Constants must be initialized during declaration.

After initialization their value cannot be changed.

Use `const` whenever possible!
Character variables

```c
char character;
```

are encoded as integer numbers:

```c
char character = 'A';
char character = 65;
```

mean the same character (ASCII code)

Assignments of character literals to character variables require single quotation marks ’:

```c
char yes = 'Y';
```
Arrays in C/C++

**Static array declaration for a one-dimensional array of type double:**

double a[5] ;  one-dimensional array with 5 elements of type double (e.g., vectors)

Access to individual elements:


**Heads up!**

In C/C++ the index for arrays starts always at 0 and runs in this example until 4, so the last element is a[4].

*A common source of errors in C/C++ !!!*

Note: While the size of the array can be set during runtime, the size cannot be changed after declaration (**static** declaration).
an \( m \times n \) matrix (rows \( \times \) columns):

\[
\begin{pmatrix}
\begin{array}{cccc}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & \cdots \\
    \vdots \\
    a_{m1} & & & a_{mn}
\end{array}
\end{pmatrix}
\]

\[
\text{int } a[m][n] \ldots \text{ static allocation of two-dimensional array, e.g., for matrices } (m, n \text{ must be constants})
\]

access via, e.g., \( a[i][j] \)

\( i \) is the index for the rows,
\( j \) for the columns.
Two-dimensional arrays II

\[
e.g., \quad a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}
\]

Note that in C/C++ the second (last) index runs first, i.e. the entries of \(a[2][3]\) are in this order in the memory:

\[
\text{a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2]}
\]

\[
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6
\]

(row-major order → stored row by row)
Initialization of arrays

An array can be initialized by curly braces:

```c
int array[5] = {0, 1, 2, 3, 4} ;

short field[] = {0, 1} ; // array field is automatically
                       // dimensioned

float x[77] = {0} ; // set all values to 0
```
Strings

There are no string variables in C. Therefore strings are written to one-dimensional character arrays:

```c
char text[6] = "Hello" ;
```

The string literal constant "Hello" consists of 5 printable characters and is terminated automatically by the compiler with the null character \0, i.e. the array must have a length of 6 characters! Note the double quotation marks!

Example

```c
char text[80] ;
cout << endl << "Please enter a string:" ;
cin >> text ;
cout << "You have entered " << text << "." << endl ;
```
Pointer variables – or pointer for short – allow a direct access (i.e. not via the name) to a variable.

### Declaration of pointers

```c
int    *pa    ; // pointer to int
float  *px    ; // pointer to float
int    **ppb   ; // pointer to pointer to int
```
A pointer is a variable that contains an address, i.e. it points to a specific part of the memory.
As every variable in C/C++ a pointer variable must have a data type. The value at address (memory) to which the pointer points, must be of the declared data type.

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.5</td>
<td>x</td>
</tr>
<tr>
<td>1004</td>
<td>42</td>
<td>n</td>
</tr>
<tr>
<td>1008</td>
<td>3.141...</td>
<td>d</td>
</tr>
<tr>
<td>1012</td>
<td>...5926</td>
<td></td>
</tr>
<tr>
<td>1016</td>
<td>HEY!</td>
<td>salutation</td>
</tr>
<tr>
<td>1020</td>
<td>1000</td>
<td>px</td>
</tr>
<tr>
<td>1024</td>
<td>1008</td>
<td>pd</td>
</tr>
<tr>
<td>1028</td>
<td>1004</td>
<td>pn</td>
</tr>
<tr>
<td>1032</td>
<td>1016</td>
<td>psalutation</td>
</tr>
<tr>
<td>1036</td>
<td>1028</td>
<td>pp</td>
</tr>
</tbody>
</table>
Pointers must be always **initialized** before usage!

**Initialization of pointers**

```c
int  *pa ; // pointer to int
int b ;   // int
pa = &b ; // assigning the address of b to a
```

The character `&` is called the address operator ("address of") (not to be confused with the reference `int &i = b ;`).

**Declaration and initialization**

```c
int b ;
int *pa = &b ;
```

→ **content of pa = address of b**
With help of the **dereference operator** `*` it is possible to get access to the value of the variable `b`, one says, pointer `pa` is dereferenced:

### Dereferencing a pointer

```c
int b, *pa = &b ;
*pa = 5 ;
```

Here, `* ...` is the **dereference operator** and means “value at address of ...”. The part of the memory to which `pa` points, contains the value `5`, that is now also the value of the variable `b`.

```c
cout << b << endl ; // yields 5  
cout << pa << endl ; // e.g., 0x7fff5fbff75c
```
Once again:

Pointer declaration:

```c
float *pz, a = 2.1;
```

Pointer initialization:

```c
pz = &a;
```

Result – output:

```c
cout << "address of variable a (content of pz): " << pz << endl;
cout << "content of variable a: " << *pz << endl;
pz = 5.2; // change value of a
```
A reference is a new name, an alias for a variable. So, it is possible to address the same part of the memory (variable) by different names within the program. Every modification of the reference is a modification of the variable itself - and vice versa.

References are declared via the & character (reference operator) and must be initialized instantaneously:

```c
int a;
int &b = a;
```

This initialization cannot be changed any more within the program!
Passing variables to functions I

Structure of functions – definition

\[ \text{type name (arg1, ...)} \{ \ldots \} \]

example: \text{int main (int argc, char *argv[])} \{ \}

- in parenthesis: arguments of the function / formal parameters
- when function is called: copy arguments (values of the given variables) to function context → \textit{call by value / pass by value}

\text{setzero (float x)} \{ \ x = 0. ; \}

\text{int main ()} \{ \\
\quad \text{float y = 3. ;} \\
\quad \text{setzero (y) ;} \\
\quad \text{cout << y ; // prints 3. } \}

H. Todt (UP)
Computational Astrophysics
SoSe 2020
Call by value

Pros:
- the value of a passed variable cannot be changed unintentionally within the function

Cons:
- the value of a passed variable can also not be changed on purpose
- for every function call all value must be *copied* → extra overhead (time)
  (exception: if parameter is an array, only *start address* is passed → pointer)
void swap(int &a, int &b) ;

Passing arguments as references:

The variables passed to the function `swap` are changed in the function and keep these values after returning from `swap`.

```c
void swap (int &a, int &b) {
    int t = a ;  a = b ;  b = t ; }
```

→ and called via: `swap (n, m)` ;

Thereby we can pass an arbitrary number of values back from a function.

**Hint:** The keyword `const` prevents that a passed argument can be changed within the function:

```c
sum (int const &a, int const &b) ;
```
A function for swapping two `int` variables can also be written by using pointers:

```c
void swap(int *a, int *b) { // pointers as formal parameters
    int t = *a ; *a = *b ; *b = t ;
}
```

Call in `main()`:

```c
swap (&x, &y) ; // Passing addresses (!)
// of x and y
```

Passing arrays to functions

In contrast to (scalar) variables, arrays are automatically passed by address (pointer) to functions, e.g.,

```c
myfunc ( float x[] )
```
Pointer variables

- store addresses
- must be dereferenced (to use the value of the spotted variable)
- can be assigned as often as desired to different variables (of the same, correct type) within the program

References

- are alias names for variables,
- can be used by directly using their names (without dereferencing)
- the (necessary!) initialization at declaration cannot be changed later
Besides the intrinsic (/basic) data types there are many other data types, which can be defined by the programmer

```c
struct complex {
    float re;
    float im;
};
```

In this example the data type `complex` is defined, it contains the member variables for real and imaginary part.
Structs and classes – defining new data types II

Structs can be imagined as collections of variables.

```c
struct
{
    char full_name[30] ;
    unsigned short binarity ;
    float luminosity_1sun ;
} ;
```

These (self defined) data types can be used in the same way as intrinsic data types:

```
complex z, c ;
star sun ;
```
Concrete structs which are declared in this way are called *instances* or *objects* (→ object-oriented programming) of a class (struct).

### Declaration and initialization

```c
complex z = {1.1, 2.2} ;
star sun = {"Sun", 1, 1.0} ;
```

The access to *member variables* is done by the *member selection operator* . (dot):

### Access to members

```c
real_part = z.re ;
sun.luminosity_1sun = 1.0 ;
```
It is also possible to define functions (so-called *methods*) within structs:

```cpp
struct complex {
    ...
    float absolute () {
        return (sqrt(re*re + im*im)) ;
    }
}
complex c = {2., 4.} ;
cout << c.absolute() << endl ;
```

The call of the *member function* is also done with the `.`, the function (method) is associated with the object.
Output to a file by using library fstream:

1. \texttt{#include <fstream>}
2. create an \texttt{object of the class} \texttt{ofstream}:
   \begin{verbatim}
   ofstream fileout;
   \end{verbatim}
3. method \texttt{open} of the class \texttt{ofstream}:
   \begin{verbatim}
   fileout.open("graphic.ps");
   \end{verbatim}
4. writing data: e.g.
   \begin{verbatim}
   fileout << x;
   \end{verbatim}
5. close file via method \texttt{close}:
   \begin{verbatim}
   fileout.close();
   \end{verbatim}

Alternatively (Unix): Use \texttt{cout} and redirection operator \texttt{> or >>} of the shell:
\begin{verbatim}
./program > output.txt
\end{verbatim}
By including the `<fstream>` library, one can also read from a file.

```cpp
char line[132] ;
ifstream filein ; // create ifstream object
filein.open("data.txt") ; // open the file
while ( filein.good() ) {
    filein.getline(line,132) ; // read in line;
        // use buffer size (132)
    x[i] = atof(line) ; // read into float array
}
```

The method `good()` checks, whether the end of file (EOF) is reached or an error occurred.
Templates allow to create universal definitions of certain structures. The final realization for a specific data type is done by the compiler.

### Function templates

```cpp
template <class T> // instead of class also typename
T sqr(const T &x) {
    return x * x;
}
```

The keyword `template` and the angle brackets `< >` signalize the compiler that `T` is a template parameter. The compiler will process this function if a specific data type is invoked by a function call, e.g.,

```cpp
double w = 3.34; int k = 2;
cout << sqr(w) << " " << sqr(k);```

H. Todt (UP)  Computational Astrophysics  SoSe 2020  105 / 647
Moreover, templates can be used to create structs/classes. For example, the class `complex` of the standard C++ library (`#include <complex>`) is realized as template class:

```cpp
Class templates
template <class T>
class std::complex {
    T re, im;
public:
    ...
    T real() const return re;
}
```

Therefore, the member variables `re` and `im` can be arbitrary (numerical) data types.
The `typedef` directive is used to define new type names for existing data types. By using `typedef datatype aliasname` one can declare new names for data types:

```c
typedef unsigned long large;
typedef char* pchar;
typedef std::complex<double> complex_d;
```

These new type names can then be used for variable declarations:

```c
large mmm;
pchar Bpoint;
complex_d z = complex_d(1.2, 3.4);
```

In the last example, the constructor for the class template `std::complex` gets the same name as the variable through the `typedef` command.
A major strength of C++ is the ability to handle runtime errors, so called exceptions:

```cpp
try {
    cin >> x ;
    if ( x < 0.) throw "Negative value!" ;
    y = g(x) ;
}
catch (char* info) { // catch exception from try block
    cout << "Program stops, because of: " << info << endl ;
    exit (1) ;
}
double g (double x) {
    if (x > 1000.) throw "x too large!" ; ... }
```
Exception handling – exceptions II

```java
try { ... }

- within a try block an arbitrary exception can be thrown
```

```java
throw e ;
```

- throw an exception e

- the data type of e is used to identify to the corresponding catch block to which the program will jump

- exceptions can be intrinsic or self defined data types
catch ( type e ) { ... } 

- after a try one or more catch blocks can be defined
- from the data type of e the first matching catch block will be selected
- any exception can be caught by catch (...)
- if after a try no matching catch block is found, the search is continued in the next higher call level
- if no matching block at all is found, the terminate function is called; its default is to call abort
Sometimes it is more convenient to pass the parameters the program needs directly at the call of the program, e.g,

./rstarcalc 3.5 35.3

this can be realized with help of the library stdlib.h

Read an integer number from command line call

```c
#include "stdlib.h"
int main (int narg, char *args[]) {
    int k;
    // convert char array to integer
    if (narg > 1) k = atoi(args[1]) ;
}
```

- if the string cannot be converted to int, the returned value is 0
- there exist also atol and atof for conversion to long and float
Common mistakes in C/C++:

- forgotten semicolon ;

- wrong dimensioning/access to arrays
  ```c
  ```

- wrong data type in instructions / function calls
  ```c
  float x ; ... switch (x)
  void swap (int *i, int *j) ; ... swap(n,m) ;
  ```

- confusing assignment operator = with the equality operator ==
  ```c
  if(i = j) → if(i == j)
  ```

- forgotten function parenthesis for functions without parameters
  ```c
  clear ; → clear();
  ```

- ambiguous expressions
  ```c
  if (i == 0 && ++j == 1)
  no increment of j, if i ≠ 0
  ```
Some recommendations I

- use always(!) the . for floating point literals: \( x = 1. / 3. \) instead of \( x = 1 / 3 \)

- whitespace is for free \( \rightarrow \) use it extensively for structuring your source code (indentation, blank lines)

- comment so that you(!) understand your source code in a year

- use self-explaining variable names, e.g., \( \text{Teff} \) instead of \( T \) (think about searching for this variable in the editor)

- use integer loop variables:
  
  ```
  for (int i = 1; i < n ; ++i) {
      x = x + 0.1 ; ... }
  ```

  instead of
  
  ```
  for (float x = 0.; x < 100. ; x = x + 0.1) {... }
  ```

- take special care of user input, usually: \( t_{\text{input}} \ll t_{\text{calc}} \), so exception catching for input is never wasted computing time
Numerical precision
as seen for float $7 + 1.E-7$: because of only 23 bit for mantissa result is 7

due to: machine precision $\epsilon_m$ is maximum possible number for which $1_c + \epsilon_m = 1_c$, where $c$ means computer representation

hence: for any number $x_c$

$\begin{align*} x_c &= x(1 \pm \epsilon), \\
|\epsilon| &\leq \epsilon_m \end{align*}$

remember: for all 32 bit floats $\rightarrow$ error in 6th decimal place, for 64 bit doubles $\rightarrow$ error in 15th place
Determining machine precision

```cpp
float eps = 1.f ;
for (int i = 1 ; i < 100 ; ++i){
    eps = eps / 2.f ; // float literal 2.f
    cout << i << " " << eps << " "
        << setprecision(9)
        << 1.f + eps << endl ;
}
```
e.g., for float:
23 1.1920929e-07  1.00000012
24 5.96046448e-08  1
We may distinguish:

- **random errors**: caused by non-perfect hardware, e.g., aging of RAM cells; can be minimized by, e.g., by ECC techniques (corrects 1 bit errors, recognizes 2 bit errors) → likelihood increases with runtime

- **approximation errors**: because of finiteness of computers, e.g., stopping series calculation, finite integration steps, ...

\[
e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \approx \sum_{n=0}^{N} \frac{(-x)^n}{n!} = e^{-x} + \mathcal{E}(x, N) \tag{7}
\]

where \( \mathcal{E} \) vanishes for \( N \to \infty \), hence we require \( N \gg x \), expecting large \( \mathcal{E} \) for \( x \approx N \)
• **roundoff errors**: limitation in the representation of real numbers (finite number of digits), e.g., if only three decimals are stored: \(1/9 = 0.111\) and \(5/9 = 0.556\), hence

\[
5 \left( \frac{1}{9} \right) - \frac{5}{9} = 0.555 - 0.556 = -0.001 \neq 0
\]  

→ error is intrinsic and *accumulates with the number of calculation steps*
→ some algorithms unstable because of roundoff errors

again: for a *float* number like

\[
x = 11223344556677889900. = 1.1223344556677889900 \times 10^{19}
\]  

only the first part (32 bit: 1.12233) is stored, while exponent is stored exactly
consider computer representation \( x_c \) of an exact number \( x \):

\[
x_c \simeq x(1 + \epsilon_x)
\]  
(10)

with relative error \( \epsilon_x \) in \( x_c \) (similar to machine precision)

so for subtraction

\[
a = b - c \rightarrow a_c \simeq b_c - c_c \simeq b(1 + \epsilon_b) - c(1 + \epsilon_c)
\]
(11)

\[
\rightarrow \frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a}
\]
(12)

(weighted errors) and if \( b \simeq c \)

\[
\frac{a_c}{a} = 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a} \max(|\epsilon_b|, |\epsilon_c|)
\]
(13)

as \( b \simeq c \rightarrow b/a \gg 1 \rightarrow \) relative error in \( a \) blown up
Warning

When subtracting two large numbers resulting in a small number, significance is lost.

Examples:

- computation of derivatives according to \( \frac{f(x+h)-f(x)}{h} \)
- the original Verlet method: \( v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t} \)
- solution of quadratic equation for \( b \gg 4ac \):
  \[
  x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{or} \quad x_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \tag{14}
  \]
- in \( e^{-x} \) for large \( x \): the first terms \( (1 - x + x^2/2 - \ldots) \) are large \( \rightarrow \) as result is small \( \rightarrow \) subtraction by other large terms \( \rightarrow \) improve algorithm by calculating \( 1/e^x \)
Roundoff error accumulation:

\[ a = b \times c \rightarrow a_c = b_c \times c_c = b(1 + \epsilon_b) \times c(1 + \epsilon_c) \]  
\[ \rightarrow \frac{a_c}{a} = (1 + \epsilon_b)(1 + \epsilon_c) \approx 1 + \epsilon_b + \epsilon_c \]  

(neglecting very small \( \epsilon^2 \) terms) \( \rightarrow \) as for physical error-propagation:

so, model for error-propagation: similar to random-walk (see later) where accumulated distance after \( N \) steps of length \( \ell \) is \( \approx \sqrt{N\ell} \), roundoff error may accumulate randomly:

\[ \epsilon_{\text{roundoff}} \approx \sqrt{N} \epsilon_m \]  

\( \rightarrow \) if no detailed error analysis available;

otherwise, if not random: \( \epsilon_{\text{roundoff}} \approx N\epsilon \)
Usually: if $A$ is correct result and numerical approximation is $A(N)$, accuracy of $A(N)$ improves by adding more terms, i.e.

$$
\epsilon_{\text{appr}} \simeq \frac{\alpha}{N^\beta}
$$

(18)

with some constants $\alpha, \beta$ depending on algorithm

However, each calculation step might increase roundoff error, so

$$
\epsilon_{\text{tot}} = \epsilon_{\text{appr}} + \epsilon_{\text{roundoff}} \simeq \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_m
$$

(19)

Hopefully: $\epsilon_{\text{appr}}$ dominant, but $\epsilon_{\text{roundoff}}$ grows slowly

$\rightarrow$ stop calculation (optimum $N$) for minimum $\epsilon_{\text{tot}}$
If we knew exact $A$ (then we also wouldn’t need to calculate $A(N)$):

$$A(N) \simeq A + \frac{\alpha}{N^\beta} \quad (20)$$

Can get handle on $\epsilon_{\text{appr}}$ by performing calculation 2nd time with $2N$ steps, then (if $\epsilon_{\text{appr}} \gg \epsilon_{\text{roundoff}}$):

$$A(N) - A(2N) \simeq \frac{\alpha}{N^\beta} \simeq \epsilon_{\text{appr}} \quad (21)$$
Minimize the error

Let’s assume that some algorithm behaves like

\[ \epsilon_{\text{appr}} \simeq \frac{1}{N^2} \rightarrow \epsilon_{\text{tot}} \simeq \frac{1}{N^2} + \sqrt{N} \epsilon_m \]  

(22)

Then the best result (minimum total error) is achieved for an \( N \) from

\[ \frac{d\epsilon_{\text{tot}}}{dN} = 0 \rightarrow N^{\frac{5}{2}} = \frac{4}{\epsilon_m} \]  

(23)

So, for single precision (\( \epsilon_m \simeq 10^{-7} \))

\[ N^{\frac{5}{2}} = \frac{4}{10^{-7}} \rightarrow N \simeq 1099 \rightarrow \epsilon_{\text{tot}} = 4 \times 10^{-6} \]  

(24)

→ total error dominated by \( \epsilon_m \), typical for single precision
Minimize the error II

So, if another algorithm

\[ \epsilon_{\text{appr}} \sim \frac{2}{N^4} \rightarrow \epsilon_{\text{tot}} \sim \frac{2}{N^4} + \sqrt{N} \epsilon_m \]  

(25)

And again minimum error obtained for an \( N \)

\[ \frac{d\epsilon_{\text{tot}}}{dN} = 0 \rightarrow N^{9/2} = \frac{16}{\epsilon_m} \rightarrow N \approx 67 \rightarrow \epsilon_{\text{tot}} = 9 \times 10^{-7} \]  

(26)

So, need less steps and also obtain better precision

The better algorithm is not more elegant but needs less calculation steps and achieves a better precision.
Arrays, libraries, make, X11
Arrays
Declaration of a 1d-array:

```cpp
int m[6] ; // statically dimensioned
```

Declaration of a function with an array type argument:

```cpp
int sumsor (int m[], int n) ; // n = length of m
```

Calling a function with an array type argument:

```cpp
sum = sumsor (m, 6) ;
```

→ passing the array is implicitly done by a pointer, i.e. only the *start address* of the array will be passed to the function
Correspondence of pointers and arrays

→ see exercise

• the assignment

\[ a[i] = 1 ; \]

is equivalent to

\[ *(a + i) = 1 ; \]

• when passing 1d-arrays to functions the start address and the data type (size of the entries) is sufficient
Problem:
When using multi-dimensional arrays, passing of the start address alone is not sufficient. Every dimensioning after the first one must be explicitly written.

Therefore:

```c
float absv (float vector[], int n) ; \ 1d-array
float trace (float matrix[][10]) ; \ 2d-array
float maxel (float tensor[][13][13]) ; \ 3d-array
```

→ special matrix-classes simplify the passing to functions
→ in Fortran, passing arrays to functions is much easier
Libraries
Excursus: Libraries I

→ collection of functions, variables, operators

```
#include <iostream>
```

- already seen: even simple input/output needs an additional library (e.g., `iostream`)

- idea of C/C++ in contrast to many other languages: only a few builtin instructions (e.g., `return`),
everything else realized by corresponding libraries
  ⇒ high flexibility because of “outsourcing”

- also mathematical functions only available by corresponding libraries (e.g., `cmath` for sin and power)

- libraries allow easily the reuse of functions in different programs
Excursus: Libraries II

Including libraries in C++:

- at compile time:
  automatic call of the C preprocessor (cpp) by g++:
  read all instructions which start with a #, especially

  ```
  #include <iostream>
  ```

  → look in the specified (default) directory paths (e.g., /usr/include/) for header files, usually with extension .h, here: iostream
  → include the corresponding header file
  → pass output to compiler

The `<iostream>` header

The header file for the iostream library is in
/usr/include/c++/x.x/iostream, where x.x depends on the specific version. It basically contains further include instructions.
The C preprocessor

CPP statements start with #, no semicolon ; at the end, but can be commented out via //

If the preprocessor is called explicitly:

cpp rcalc.cpp output

then from the source file rcalc.cpp, it generates an output file output, in which, e.g., #define instructions are resolved

- at link time:
  look for the libraries which belong to the header files, translate the names (symbols) used in the library to (relative) memory addresses;
  static linking: include the necessary library symbols in the program
Dynamic libraries

The Unix command \texttt{ldd} lists the dynamically linked-in libraries for a given program (or object file/library), e.g., \texttt{ldd -v rcalc}:

\begin{verbatim}
linux-vdso.so.1 (0x00007fff72bff000) †
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007ff2d9c0b000)
\end{verbatim}

The path to the library and the memory address is printed.

- at runtime:
  - dynamic linking: loading program and libraries to memory (RAM)
  - advantage (over static linking): library is loaded only once and can be used by other programs

†vdso = virtual dynamic shared object
Excursus: Libraries V

C Preprocessor (cpp)

⇓

Compiler (g++)

⇓

Linker (ld)
Overview: Unix commands for developers

- **cpp**: C preprocessor for the #-instructions
- **g++**: C++ compiler
- **ld**: link editor (usually called by the compiler)
- **ldd**: lists the used libraries of an object file (also program or library)
- **nm**: lists the *symbols* of an object file (etc.)

**Symbols**

In a C++ program `main` belongs to the symbols labeled with letter T. I.e., it is a symbol from the text (code) section of the file.
sometimes necessary for using some specific libraries: explicit specification (name) of the library at link time

specification of a library libpthread.so via lower case l:

```
-lpthread
```

when calling the compiler for creation of the executables

specification of the path to the library via upper case L:

```
-L/usr/lib/ -lpthread
```

when calling the compiler for creation of the executables

Heads up: The path must be given before the library!
dynamic libraries must be located in a default system path (e.g., /lib) or the path must be added to the environment variable 

LD_LIBRARY_PATH

E.g. for the bash via

export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:

and for the csh respectively

setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:

→ extending the path to dynamic libraries for the current working directory
static libraries (file extension `.a`) are archives of object files
these objects files are fixed included in the binary output during the
procedure of static linking (→ large program files)

**Sequence for static linking**

If a library/program `libA` needs symbols from the library `libB`, the name
of `libA` must be given before that of `libB` at link time for static linking:
`-lA -lB`

- (complete) static linking isn’t supported anymore by modern OSs (e.g. MacOS) at normal developer level
- but against some libraries (e.g., `libgfortran`, MKL) it can be
  selectively statically linked
make
Purpose of make:

- automatic determination of the program parts (usually source files) that must be re-compiled via
  - a given definition of the dependencies (implicit, explicit)
  - comparison of time stamps (file system)
- calling the required commands for re-compilation:

  typical use: ./configure ; make ; make install

useful especially for large programs with many source files
Main idea of `make` is the *rule*:

**Target : Dependencies**
<TAB> command for creation of the target

e.g.,

```
myprogram : myprogram.o
<TAB> g++ -o $@ $? 
```

**Note**
- explicit rules are defined via an ASCII file, the so-called *makefile*
- every command belonging to a rule must be started with a <TAB>!
- the macros $@ and $? are called automatic variables, i.e., they are replaced by `make`: $@ is replaced by the target, $? by the dependencies that are newer than the target
Implicit rules:

- some rules for compilation are re-occurring, e.g., for C++ .o files are always created from .cpp files
- make has therefore a number of implicit rules, hence make can also be used without a makefile

Example

```bash
echo 'int main() {}' > myprog.cpp
make myprog
executes g++ -o myprog myprog.cpp
```

- make uses implicit rules if no explicit rule for creation of the target has been found

1 make invokes g++ automatically, or the C++ compiler that is specified in the environment variable CXX
Explicit rules

- an explicit rule is usually specified in a text file that has one of the following default names: makefile, Makefile
- every rule must define at least one target
- it is possible to define several dependencies for one target
- a rule can contain an arbitrary number of commands

Moreover, explicit rules overwrite implicit rules:

```
.c.o :
<TAB> $CPP -c $?

$(PROJECT) : $(OBJECTS)
<TAB> $(CPP) $(CPPLAGS) -o $(@) $(OBJECTS)
```
Usual run of a make call:

1. after calling make the makefile is parsed (read)
2. read and substitute variables (see below) and determination of the highest target(s) (given in the beginning), evaluation of the dependencies
3. creation of a tree of dependencies
4. determination of the time stamps for all dependencies of the corresponding files and comparison with those of the next step in the tree
5. targets whose dependencies are newer than the targets are re-compiled
Variables

- during processing of the rules `make` uses automatic variables, e.g., `$@` and `$?` (see above)
- variables can also be defined explicitly before the first rule, syntax is shell-like:

  \[
  \begin{align*}
  \text{CC} &= \text{gcc} \\
  \text{CFLAGS} &= -O3 \\
  \text{PROJECT} &= \text{galaxy}
  \end{align*}
  \]
- variables can, as in the shell, be hold together with help of curly braces
  \[
  ${OBJECTFILES}$, or with help of round parentheses
  $(CFLAGS)$
Usual pseudo targets → Call via make *pseudo target*

- don’t create a file, or don’t have dependencies, e.g.
- clean, for make clean, defines explicitly how the intermediate and final products (targets) of the compilation shall be removed
- all  creates all project files
- install  if the targets (programs, libraries) shall be copied to a specific directory (or similar), it should be stated in install

Pseudo targets (e.g., clean) can only be used if defined in the makefile.
Example of a makefile

CXX = g++ -O3
CPFLAGS = -Wall
LIBRARIES = -lX11

OBJECTS = componentA.o componentB.o
PROJECT = myprogram

$(PROJECT) : $(OBJECTS)
   ${CXX} $(CPFLAGS) $(OBJECTS) -o $@ ${LIBRARIES}

.cpp.o :
   ${CXX} -c ${CPFLAGS} $? 

clean :
   rm -f $(OBJECTS)
Makefile uses a shell-like syntax:

- comments are started with a #:
  
  ```shell
  # a comment
  ```

- one command per line, multiple commands via `;` and line continuation via `\`
  ```shell
  $FC $? ; ldconfig
  ```

- every command corresponds to a shell command, and is printed before execution:
  ```shell
  .c.o :
    echo "Hello ${USER}"
  ```

the print-out of commands can be suppressed with `@` before the command

  ```shell
  @echo "Hi ${DATE}"'
  ```
variables are set without $ and used/referenced with a $

progname = opdat
PROJECT = $(progname).exe

Variable names that contain multiple characters should always hold together with parentheses () or curly braces {}.
Special targets:

- problem: pseudo target clean is not executed, if a file with that name exists (why?)
- solution: pseudo targets can be marked as such via the special target .PHONY:

  .PHONY: clean install

- special targets start with a .
Some more special targets:

- `.INTERMEDIATE` : dependencies are only created if another dependency before the target is newer, or if a dependency of an intermediate file is newer than the actual target. The intermediate target is deleted after the target was created:
  
  `.INTERMEDIATE : colortable.o`

```
xapple.exe : xapple.cpp colortable.o
    $(CXX) -o xapple.exe xapple.cpp colortable.o
```

```
colortable.o : colortable.cpp
    $(CXX) -c colortable.cpp
```

Here, `colortable.o` is only created if `xapple.cpp` or if `colortable.cpp` are newer than `xapple.exe`. After the creation of `xapple.exe` the target `colortable.o` will be removed.
.SECONDARY: like .INTERMEDIATE, but the dependencies are not removed automatically

.IGNORE: errors during creation of the specified dependencies will not lead to an abort of the make procedure

**Hint**

The tool make is not bound to programming languages, but can also be used for, e.g., automatic compilation of .tex files etc.
Graphics with X11
Graphical output with X11

- there are many libraries for graphical output:
  - Qt, e.g., for Mathematica
  - Simple DirectMedia Layer for simple games
  - ...

- Pros: large support, comprehensive literature, often platform independent (e.g. via ports)

- Cons: often huge frameworks even for simplest tasks, huge libraries (memory consumption), usually high thresholds for beginners

- always available under Unix/Linux: X11 or just X with many abilities:
  - creation of windows incl. internal structures (panels)
  - simple routines for drawing lines, circles, colors
  - keyboard and mouse inquiry
  - graphical forwarding (ssh -X)

→ We want to use X11 more or less directly with help of the library Xgraphics.
The two-body problem
We remember (?):

The Kepler’s laws of planetary motion (1619)

1. Each planet moves in an elliptical orbit where the Sun is at one of the foci of the ellipse.

2. The velocity of a planet increases with decreasing distance to the Sun such, that the planet sweeps out equal areas in equal times.

3. The ratio $P^2/a^3$ is the same for all planets orbiting the Sun, where $P$ is the orbital period and $a$ is the semimajor axis of the ellipse.

The 1. and 3. Kepler’s law describe the shape of the orbit (Copernicus: circles), but not the time dependence $\vec{r}(t)$. This can in general not be expressed by elementary mathematical functions (see below). Therefore we will try to find a numerical solution.
Earth-Sun system

→ two-body problem → one-body problem via reduced mass of lighter body (partition of motion):

\[ \mu = \frac{M \, m}{m + M} = \frac{m}{\frac{m}{M} + 1} \] (27)

as \( m_E \ll M_\odot \) is \( \mu \approx m \), i.e. motion is relative to the center of mass \( \equiv \) only motion of \( m \). Set \((0,0)\) to the source of the force field of \( M \).

Moreover: Newton’s 2. law:

\[ m \frac{d^2 \vec{r}}{dt^2} = \vec{F} \] (28)

and force field according to Newton’s law of gravitation:

\[ \vec{F} = -\frac{G M m}{r^3} \vec{r} \] (29)
Kepler’s laws, as well as the assumption of a central force imply *conservation of angular momentum* → motion is only in a *plane* (→ Kepler’s 1st law).

So, we use Cartesian coordinates in the $xy$-plane:

\[
\begin{align*}
F_x &= -\frac{GMm}{r^3} x \quad (30) \\
F_y &= -\frac{GMm}{r^3} y \quad (31)
\end{align*}
\]

The equations of motion are then:

\[
\begin{align*}
d\frac{d^2 x}{dt^2} &= -\frac{GM}{r^3} x \quad (32) \\
&= \frac{d^2 y}{dt^2} = -\frac{GM}{r^3} y \quad (33)
\end{align*}
\]

where $r = \sqrt{x^2 + y^2}$ (34)
To derive the analytic solution for equation of motion $\mathbf{r}(t) \rightarrow$ use polar coordinates: $\phi, r$

1. use conservation of angular momentum:

$$\mu r^2 \dot{\phi} = \ell = \text{const.} \quad (35)$$

$$\dot{\phi} = \frac{\ell}{\mu r^2} \quad (36)$$

2. use conservation of energy:

$$E = \frac{1}{2} \mu \dot{r}^2 + \frac{\ell^2}{2 \mu r} - \frac{GM\mu}{r} \quad (37)$$

$$\dot{r}^2 = \frac{2E}{\mu} - \frac{\ell^2}{\mu^2 r^2} + \frac{2GM}{r} \quad (38)$$

$\rightarrow$ two coupled equations for $r$ and $\phi$
Excursus: Analytic solution of the Kepler problem II

Decouple Eq. (36), use the orbit equation \( r = \frac{\alpha}{1 + e \cos \phi} \) with numeric eccentricity \( e \) and \( \alpha \equiv \frac{\ell^2}{GM\mu^2} \) gives separable equation for \( \dot{\phi} \):

\[
\dot{\phi} = \frac{d\phi}{dt} = \frac{G^2 M^2 \mu^3}{\ell^3}(1 + e \cos \phi)^2
\]

(39)

\[
t = \int dt' = k \int \frac{d\phi'}{(1 + e \cos \phi')^2} = f(\phi)
\]

(40)

Right-hand side integral can be looked up in, e.g., Bronstein:

\[
e\sin \phi = \frac{e \sin \phi}{(e^2 - 1)(1 + e \cos \phi)} - \frac{1}{e^2 - 1} \int \frac{d\phi}{1 + e \cos \phi}
\]

(41)

\( \rightarrow \ e \neq 1: \) parabola excluded; the integral can be further simplified:

\[
0 \leq e < 1: \int \frac{d\phi}{1 + e \cos \phi} = \frac{2}{\sqrt{1 - e^2}} \arctan \left( \frac{(1 - e) \tan \frac{\phi}{2}}{\sqrt{1 - e^2}} \right)
\]

(42)
for the ellipse; and for the hyperbola:

\[ e < 1 : \int \frac{d\phi}{1 + e \cos \phi} = \frac{1}{\sqrt{e^2 - 1}} \ln \left( \frac{(e - 1) \tan \frac{\phi}{2} + \sqrt{e^2 - 1}}{(e - 1) \tan \frac{\phi}{2} - \sqrt{e^2 - 1}} \right) \]  (43)

→ Eqn. (42) & (43) must be inverted to get \( \phi \)!
→ only easy for \( e = 0 \) → circular orbit

\[ t = k \int d\phi' = k\phi \rightarrow \phi(t) = k^{-1}t = \frac{G^2 M^2 \mu^3}{\ell^3} t \]  (44)

and from orbit equation \( r = \alpha = \frac{\ell^2}{GM\mu^2} = \text{const.} \).

For the general case, it is much easier to solve the equations of motion numerically.
Excursus: The Kepler equation I

Alternative formulation for time dependency in case of ellipse ($0 \leq e < 1$):

Orbit, circumscribed by auxiliary circle with radius $a$ (semi-major axis); true anomaly $\phi$, eccentric anomaly $\psi$. Sun at $S$, planet at $P$, circle center at $O$. Perapsis $\Pi$ and apapsis $A$.
Consider line normal to $\overline{AP}$ through $P$ on the ellipse, intersecting circle at $Q$ and $\overline{AP}$ at $R$.

consider an angle $\psi$ (or $E$, eccentric anomaly) defined by $\angle \Pi OQ$
Excursus: The Kepler equation II

The position \((r, \phi)\) of the body \(P\) can be described in terms of \(\psi\):

\[
x_P = r \cos \phi = a \cos \psi - ae
\]

\[
y_P = r \sin \phi = a \sin \psi \sqrt{1 - e^2}
\]

(with \(\frac{PR}{QR} = \frac{b}{a} = \sqrt{1 - e^2}\)), square both equations and add them up:

\[
r = a(1 - e \cos \psi)
\]

Now, to find \(\psi = \psi(t)\), need relationship between \(d\phi\) and \(d\psi\), so combine Eqn. (46) & (47):

\[
\sin \phi = \frac{b \sin \psi}{a(1 - e \cos \psi)} |d/dx'\quad (48)
\]

\[
\cos \phi d\phi = \frac{b(\cos \psi(1 - e \cos \psi)d\psi - e \sin^2 \psi d\psi)}{a(1 - e \cos \psi)^2} \quad (49)
\]

\[
d\phi = \frac{b}{a(1 - e \cos \psi)} d\psi \quad (50)
\]
Excursus: The Kepler equation III

together with the angular momentum \( d\phi = \frac{\ell}{\mu r^2} dt \):

\[
(1 - e \cos \psi) d\psi = \frac{\ell}{\mu ab} dt \tag{51}
\]

\[
= \text{set } t = 0 \rightarrow \psi(0) = 0, \text{ integration: } \tag{52}
\]

\[
\psi - e \sin \psi = \frac{\ell t}{\mu ab} \tag{53}
\]

use Kepler’s 2nd law \( \frac{\pi ab}{P} = \frac{\ell}{2\mu} \) with \( \pi ab \) the area of the ellipse, we get \( \ell/(\mu ab) = 2\pi/P \equiv \omega \) (orbital angular frequency), so:

**Kepler’s equation**

\[
\psi - e \sin \psi = \omega t \tag{54}
\]

\[
E - e \sin E = M \quad \text{(astronomer’s version)} \tag{55}
\]

\( M \): mean anomaly = angle for constant angular velocity
Kepler’s equation $E(t) - e \sin E(t) = M(t)$

- is a transcendental equation
- can be solved by, e.g., Newton’s method
- because of $E = M + e \sin E$, also (Banach) fixed-point iteration possible (slow, but stable), already used by Kepler (1621):

$$E = M;$$
$$\text{for (int } i = 0; i < n; ++i)$$
$$E = M + e \times \sin(E);$$

- can be solved, e.g., by Fourier series $\rightarrow$ Bessel (1784-1846):

\[
E = M + \sum_{n=1}^{\infty} \frac{2}{n} J_n(ne) \sin(nM) \tag{56}
\]

\[
J_n(ne) = \frac{1}{\pi} \int_0^\pi \cos(nx - ne \sin x) dx \tag{57}
\]
A special case as a solution of the equations of motion (32) & (33) is the circular orbit. Then:

\[ \ddot{r} = \frac{v^2}{r} \quad (58) \]

\[ \frac{mv^2}{r} = \frac{GMm}{r^2} \quad \text{(equilibrium of forces)} \quad (59) \]

\[ \Rightarrow \quad v = \sqrt{\frac{GM}{r}} \quad (60) \]

The relation (60) is therefore the condition for a circular orbit. Moreover, Eq. (60) yields together with

\[ P = \frac{2\pi r}{v} \quad (61) \]

\[ \Rightarrow \quad P^2 = \frac{4\pi^2}{GM} r^3 \quad (62) \]
Astronomical units

For our solar system it is useful to use astronomical units (AU):

\[ 1 \text{AU} = 1.496 \times 10^{11} \text{m} \]

and the unit of time is the (Earth-) year

\[ 1 \text{a} = 3.156 \times 10^7 \text{s} \quad (\approx \pi \times 10^7 \text{s}), \]

so, for the Earth \( P = 1 \text{a} \) and \( r = 1 \text{AU} \)

Therefore it follows from Eq. (62):

\[
GM = \frac{4\pi^2 r^3}{P^2} = 4\pi^2 \text{AU}^3 \text{ a}^{-2} \quad (63)
\]

I.e. we set \( GM \equiv 4\pi^2 \) in our calculations.

**Advantage: handy numbers!**

Thus, e.g. \( r = 2 \) is approx. \( 3 \times 10^{11} \text{m} \) and \( t = 0.1 \) corresponds to \( 3.16 \times 10^6 \text{s} \), and \( v = 6.28 \) is roughly 30 km/s.
The Euler method I

The equations of motion (32) & (33):

\[
\frac{d^2 \vec{r}}{dt^2} = -\frac{GM}{r^3} \vec{r}
\]  

(64)

are a system of differential equations of 2nd order, that we shall solve now. Formally: *integration* of the equations of motion to obtain the *trajectory* \( \vec{r}(t) \).

**Step 1: reduction**

Rewrite Newton’s equations of motion as a system of differential equations of *1st order* (here: 1d):

\[
\nu(t) = \frac{dx(t)}{dt} \quad \& \quad a(t) = \frac{dv(t)}{dt} = \frac{F(x, \nu, t)}{m}
\]

(65)
The Euler method II

Step 2: Solving the differential equation

Differential equations of the form (initial value problem)

\[ \frac{dx}{dt} = f(x, t), \quad x(t_0) = x_0 \] (66)

can be solved numerically (discretization\(^1\)) by as simple method:

**Explicit Euler method (“Euler’s polygonal chain method”)**

1. choose step size \( \Delta t > 0 \), so that \( t_n = t_0 + n\Delta t, \quad n = 0, 1, 2, \ldots \)
2. calculate the values (iteration):
   \[ x_{n+1} = x_n + f(x_n, t_n)\Delta t \]

Obvious: The smaller the step size \( \Delta t \), the more steps are necessary, but also the more accurate is the result.

\(^1\)i.e. we change from calculus to algebra, which can be solved by computers.
Why “polygonal chain method”? 

![Graph showing exact solution and numerical solution.](image-url)

Exact solution (–) and numerical solution (–).
Derivation from the Fundamental theorem of calculus

Integration of the ODE
\[ \frac{dx}{dt} = f(x, t) \text{ from } t_0 \text{ till } t_0 + \Delta t \quad (67) \]

\[ \int_{t_0}^{t_0+\Delta t} \frac{dx}{dt} \, dt = \int_{t_0}^{t_0+\Delta t} f(x, t) \, dt \quad (68) \]

\[ \Rightarrow x(t_0 + \Delta t) - x(t_0) = \int_{t_0}^{t_0+\Delta t} f(x(t), t) \, dt \quad (69) \]

Apply rectangle method for the integral:
\[ \int_{t_0}^{t_0+\Delta t} f(x(t), t) \, dt \approx \Delta t f(x(t_0), t_0) \quad (70) \]

Equating (69) with (70) yields Euler step
\[ x(t_0 + \Delta t) = x(t_0) + \Delta t f(x(t_0), t_0) \quad (71) \]


Derivation from Taylor expansion

\[ x(t_0 + \Delta t) = x(t_0) + \Delta t \frac{dx}{dt}(t_0) + O(\Delta t^2) \quad (72) \]

Use \( \frac{dx}{dt} = f(x, t) \) \quad (73)

\[ x(t_0 + \Delta t) = x(t_0) + \Delta t f(x(t_0), t_0) \quad (74) \]

while neglecting term of higher order in \( \Delta t \)
For the system (65)

\[ v(t) = \frac{dx(t)}{dt} \quad \& \quad a(t) = \frac{dv(t)}{dt} = \frac{F(x, v, t)}{m} \]

this means

Euler method for solving Newton’s equations of motion

\[ v_{n+1} = v_n + a_n \Delta t = v_n + a_n(x_n, t) \Delta t \]  \hspace{1cm} (75)

\[ x_{n+1} = x_n + v_n \Delta t \]  \hspace{1cm} (76)

We note:

- the velocity at the end of the time interval \( v_{n+1} \) is calculated from \( a_n \), which is the acceleration at the beginning of the time interval
- analogously \( x_{n+1} \) is calculated from \( v_n \)
Example: Harmonic oscillator

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

int main () {
    int n = 10001, nout = 500 ;
    double t, v, v_old, x ;
    double const dt = 2. * M_PI / double(n-1) ;
    x = 1. ; t = 0. ; v = 0. ;
    for (int i = 0 ; i < n ; ++i) {
        t = t + dt ; v_old = v ;
        v = v - x * dt ;
        x = x + v_old * dt ;
        if (i % nout == 0) // print out only each nout step
            cout << t << " " << x << " " << v << endl ;
    }
    return 0 ;
}
```
The Euler-Cromer method

We will slightly modify the explicit Euler method, but such that we obtain the same differential equations for $\Delta t \rightarrow 0$.

For this new method we use $v_{n+1}$ for calculating $x_{n+1}$:

### Euler-Cromer method (semi-implicit Euler method)

\begin{align*}
v_{n+1} &= v_n + a_n \Delta t \quad \text{(as for Euler)} \quad (77) \\
x_{n+1} &= x_n + v_{n+1} \Delta t \quad (78)
\end{align*}

Advantage of this method:

- as for Euler method, $x$, $v$ need to be calculated only once per step
- especially appropriate for oscillating solutions, as energy is conserved much better (see below)
Proof of stability (Cromer 1981):

\[ v_{n+1} = v_n + F_n \Delta t \quad (= v_n + a(x_n) \Delta t, \; m = 1) \]  \hspace{1cm} (79)

\[ x_{n+1} = x_n + v_{n+1} \Delta t \]  \hspace{1cm} (80)

Without loss of generality, let \( v_0 = 0 \). Iterate Eq. (79) \( n \) times:

\[ v_n = (F_0 + F_1 + \ldots + F_{n-1}) \Delta t = S_{n-1} \]  \hspace{1cm} (81)

\[ x_{n+1} = x_n + S_n \Delta t \]  \hspace{1cm} (82)

\[ S_n := \Delta t \sum_{j=0}^{n} F_j \]  \hspace{1cm} (83)

Note that for explicit Euler Eq. (82) is \( x_{n+1} = x_n + S_{n-1} \Delta t \).
The change in the kinetic energy $K$ between $t_0 = 0$ and $t_n = n\Delta t$ is because of Eq. (79) and $v_0 = 0$

$$\Delta K_n = K_n - K_0 = K_n = \frac{1}{2} S_{n-1}^2 \quad (84)$$

The change in the potential energy $U$:

$$\Delta U_n = - \int_{x_0}^{x_n} F(x) dx \quad (85)$$
Now use the trapezoid rule for this integral

\[ \Delta U_n = -\frac{1}{2} \sum_{i=0}^{n-1} (F_i + F_{i+1})(x_{i+1} - x_i) \]  \hspace{1cm} (86)

\[ = -\frac{1}{2} \Delta t \sum_{i=0}^{n-1} (F_i + F_{i+1})S_i \quad (\rightarrow \text{Eq. 80}) \]  \hspace{1cm} (87)

\[ = -\frac{1}{2} \Delta t^2 \sum_{i=0}^{n-1} \sum_{j=0}^{i} (F_i + F_{i+1})F_j \quad (\rightarrow \text{Eq. 83}) \]  \hspace{1cm} (88)
Excursus: Proof of stability for the Euler-Cromer method IV

As \( j \) runs from 0 to \( i \to \Delta U_n \) has same squared terms as \( \Delta K_n \), see:

\[
\Delta U_n = -\frac{1}{2} \Delta t^2 \left( \sum_{i=0}^{n-1} F_i^2 + \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} F_i F_j + \sum_{i=1}^{n} \sum_{j=0}^{i-1} F_i F_j \right) \tag{89}
\]

\[
= -\frac{1}{2} \Delta t^2 \left( \sum_{i=0}^{n-1} F_i^2 + 2 \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} F_i F_j + F_n \sum_{j=0}^{i-1} F_j \right) \tag{90}
\]

\[
= -\frac{1}{2} S^2_{n-1} - \frac{1}{2} \Delta t F_n S_{n-1} \tag{91}
\]

Hence the total energy changes as

\[
\Delta E_n = \Delta K_n + \Delta U_n = \frac{1}{2} S^2_{n-1} - \frac{1}{2} S^2_{n-1} - \frac{1}{2} \Delta t F_n S_{n-1} \tag{92}
\]

\[
= -\frac{1}{2} \Delta t F_n S_{n-1} = -\frac{1}{2} \Delta t F_n v_n \tag{93}
\]
For oscillatory motion: \( v_n = 0 \) at turning points, \( F_n = 0 \) at equilibrium points \( \rightarrow \Delta E_n = -\frac{1}{2} \Delta t F_n v_n \) is 0 four times of each cycle \( \rightarrow \Delta E_n \) oscillates with \( T/2 \).

As \( F_n \) and \( v_n \) are bound \( \rightarrow \Delta E_n \) is bound, more important: average of \( \Delta E_n \) over half a cycle \( (T) \)

\[
\langle \Delta E_n \rangle = \frac{\Delta t^2}{T} \sum_{n=0}^{\frac{T}{\Delta t}} F_n v_n \approx \frac{\Delta t}{T} \int_0^{\frac{T}{2}} F v \, dt = \frac{\Delta t}{T} \int_{x(0)}^{x(T/2)} F \, dx
\quad (94)
\]

\[
= -\frac{\Delta t}{T} (U(T/2) - U(0)) = 0
\quad (95)
\]
as \( U \) has same value at each turning point
\( \rightarrow \) energy conserved on average with Euler-Cromer for oscillatory motion

\( \square \)
For comparison: with explicit Euler method $\Delta E_n$ contains term $\sum_{i=0}^{n-1} F_i^2$ which increases monotonically with $n$ and

$$\Delta E_n = -\frac{1}{8} \Delta t^2 (F_0^2 - F_n^2) \quad (96)$$

with $v_0 = 0 \rightarrow F_0^2 \geq F_n^2 \rightarrow \Delta E_n$ oscillates between 0 and $-\frac{1}{8} \Delta t^2 F_0^2$ per cycle.

Energy is bounded as for Euler-Cromer, but $\langle \Delta E_n \rangle \neq 0$
Consider the following ODE

\[ \frac{dx}{dt} = -cx \]  

(97)

with \( c > 0 \) and \( x(t = 0) = x_0 \). Analytic solution is \( x(t) = x_0 \exp(-ct) \).

The explicit Euler method gives:

\[ x_{n+1} = x_n + \dot{x}_n \Delta t = x_n - cx_n \Delta t = x_n(1 - c \Delta t) \]  

(98)

So, every step will give \((1 - c \Delta t)\) and after \( n \) steps:

\[ x_n = (1 - c \Delta t)^n x_0 = (a)^n x_0 \]  

(99)

But, with \( a = 1 - c \Delta t \):

\[
\begin{align*}
0 < a < 1 & \quad \Rightarrow \quad \Delta t < 1/c & \text{monotonic decline of } x_n \\
-1 < a < 0 & \quad \Rightarrow \quad 1/c < \Delta t < 2/c & \text{oscillating decline of } x_n \\
a < -1 & \quad \Rightarrow \quad \Delta t > 2/c & \text{oscillating increase of } x_n
\end{align*}
\]  

(100)
Stability analysis of the Euler method II

Stability of the explicit Euler method for different $a = 1 - c \Delta t$

In contrast, consider implicit Euler method (Euler-Cromer):

$$x_{n+1} = x_n + \dot{x}_{n+1} \Delta t = x_n - cx_{n+1} \Delta t$$ \hspace{1cm} (101)

$$\Rightarrow x_{n+1} = \frac{x_n}{1 + c \Delta t}$$ \hspace{1cm} (102)

declines for all $\Delta t$ (!)
The Euler-Richardson method

Sometimes it is better, to calculate the velocity for the midpoint of the interval:

Euler-Richardson method ("Euler half step method")

\[
a_n = \frac{F(x_n, v_n, t_n)}{m} \tag{103}
\]

\[
v_M = v_n + a_n \frac{1}{2} \Delta t \tag{104}
\]

\[
x_M = x_n + v_n \frac{1}{2} \Delta t \tag{105}
\]

\[
a_M = \frac{F\left(x_M, v_M, t_n + \frac{1}{2} \Delta t\right)}{m} \tag{106}
\]

\[
v_{n+1} = v_n + a_M \Delta t \tag{107}
\]

\[
x_{n+1} = x_n + v_M \Delta t \tag{108}
\]

We need twice the number of steps of calculation, but may be more efficient, as we might choose a larger step size as for the Euler method.
The (special) three-body problem
We will not solve the general case of the three-body problem, but consider only the following configuration \((m_1, m_2 < M)\):

\[
m_1 \frac{d^2 \vec{r}_1}{dt^2} = -\frac{GMm_1}{r_1^3} \vec{r}_1 + \frac{Gm_1m_2}{r_{21}^3} \vec{r}_{21}(109)\]
\[
m_2 \frac{d^2 \vec{r}_2}{dt^2} = -\frac{GMm_2}{r_2^3} \vec{r}_2 - \frac{Gm_1m_2}{r_{21}^3} \vec{r}_{21}(110)\]

\[\dagger\] not to confuse with the restricted three-body problem, where \(m_1 \approx m_2 \gg m_3\)

→ Lagrangian points, e.g, \(L_1\) for SOHO, \(L_2\) for JWST
It is useful to divide the Eqn. (109) & (110) each by $m_1$ and $m_2$ respectively:

\[
\frac{d^2 \mathbf{r}_1}{dt^2} = -\frac{GM}{r_1^3} \mathbf{r}_1 + \frac{Gm_2}{r_{21}^3} \mathbf{r}_{21} \tag{111}
\]

\[
\frac{d^2 \mathbf{r}_2}{dt^2} = -\frac{GM}{r_2^3} \mathbf{r}_2 - \frac{Gm_1}{r_{21}^3} \mathbf{r}_{21} \tag{112}
\]

Moreover we can set – using astronomical units – again:

\[
GM = 4\pi^2 \tag{113}
\]

The terms

\[
+ \frac{Gm_2}{r_{21}^3} \mathbf{r}_{21} \quad \& \quad - \frac{Gm_1}{r_{21}^3} \mathbf{r}_{21} \tag{114}
\]
The (special)† three-body problem III

can be written with help of mass ratios

\[
\frac{m_2}{M} \quad \& \quad - \frac{m_1}{M} \quad (115)
\]

so that

\[
\text{ratio}[0] = \frac{m_2}{M} GM \quad \& \quad \text{ratio}[1] = -\frac{m_1}{M} GM \quad (116)
\]

The accelerations are then calculated like this:

\[
\begin{align*}
\text{dx} &= x[1] - x[0] \\
\ldots
\text{dr3} &= \text{pow}(\text{dx} * \text{dx} + \text{dy} * \text{dy} , 3./2.) \\
\ldots
\text{ax} &= -GM * x[i] / r3 + \text{ratio}[i] * \text{dx} / \text{dr3} \\
\text{ay} &= -GM * y[i] / r3 + \text{ratio}[i] * \text{dy} / \text{dr3}
\end{align*}
\]
Methods for solving the Newtonian equations of motion
Review → Newtonian equations of motion

\[
\frac{dv}{dt} = a(t) \tag{117}
\]
\[
\frac{dx}{dt} = v(t) \tag{118}
\]

Numerical solution from Taylor expansion:

\[
v_{n+1} = v_n + a_n \Delta t + \mathcal{O}((\Delta t)^2) \tag{119}
\]
\[
x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + \mathcal{O}((\Delta t)^3) \tag{120}
\]

Euler method: account only for $\mathcal{O}(\Delta t)$ (for $\Delta t \to 0$):

\[
v_{n+1} = v_n + a_n \Delta t \tag{121}
\]
\[
x_{n+1} = x_n + v_n \Delta t \tag{122}
\]
therefore, only having $O(\Delta t)$:
→ local truncation error in one time step: $\sim (\Delta t)^2$
→ global error over $t$: $\sim (\Delta t)$, because $n$ steps with $n = \frac{t}{\Delta t} \sim \frac{1}{\Delta t}$, so order of global error reduced by $\frac{1}{\Delta t}$

A method is of $n$th order, if the global error is of the order of $(\Delta t)^n$. The Euler method is of 1st order.

Note, the Euler-Cromer method (semi-implicit Euler method) is also of 1st order, but conserves energy (symplectic integrator):

$$v_{n+1} = v_n + a_n(x_n)\Delta t \quad (123)$$
$$x_{n+1} = x_n + v_{n+1}\Delta t \quad (124)$$

but there is also a 2nd variant of the (semi-implicit) Euler method

$$x_{n+1} = x_n + v_n\Delta t \quad (125)$$
$$v_{n+1} = v_n + a_{n+1}(x_{n+1})\Delta t \quad (126)$$
Possible improvement: use velocity from the *midpoint* of the interval

cf. Heun’s method (Karl Heun, 1859-1929)

\[ v_{n+1} = v_n + a_n \Delta t \quad \text{(as for Euler)} \tag{127} \]

\[ x_{n+1} = x_n + \frac{1}{2} (v_n + v_{n+1}) \Delta t \tag{128} \]

\[ = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2 \tag{129} \]

→ accuracy of position is of 2nd order and velocity is of 1st order (only good for constant acceleration, not more accurate than Euler, as error increases with each time step)
Numerical Integration IV

Better (stable for oscillatory motions with const. $\Delta t \leq 2/\omega$, therefore common, error bounded):

**Halfstep method / Leapfrog integration**

\begin{align}
    v_{n+\frac{1}{2}} &= v_{n-\frac{1}{2}} + a_n \Delta t \quad (130) \\
    x_{n+1} &= x_n + v_{n+\frac{1}{2}} \Delta t \quad (131)
\end{align}
2nd order with same number of steps as Euler (1st order)

But: not self starting, i.e. from Eq. (130) \( \Rightarrow v_{1/2} \)

therefore Euler method for the first half step:

\[
v_{1/2} = v_0 + \frac{1}{2} a_0 \Delta t
\]  
(132)

Moreover, velocity steps can be eliminated by using Eq. (130) & (131):

\[
(x_{n+1} - x_n) - (x_n - x_{n-1}) = (v_{n+1/2} - v_{n-1/2}) \Delta t
\]  
(133)

\[
x_{n+1} - 2x_n + x_{n-1} = a_n \Delta t
\]  
(134)

\[
x_{n+1} = 2x_n - x_{n-1} + a_n \Delta t^2
\]  
(135)

with start values \( x_0, x_1 = x_0 + v_0 + \frac{1}{2} a_0 \Delta t^2 \) (so \( v_0 \) is still required!)
Or, by interpolation of intermediate values as combination of symplectic, semi-implicit Euler methods (Eq. (123)-(126))

\[
\begin{align*}
\nu_{n+\frac{1}{2}} &= \nu_n + a_n \frac{1}{2} \Delta t \\
x_{n+\frac{1}{2}} &= x_n + \nu_n \frac{1}{2} \Delta t
\end{align*}
\] \quad \{ \text{(136)} \}

\[
\begin{align*}
x_{n+1} &= x_{n+\frac{1}{2}} + \nu_{n+\frac{1}{2}} \frac{1}{2} \Delta t \\
\nu_{n+1} &= \nu_{n+\frac{1}{2}} + a_{n+1} \frac{1}{2} \Delta t
\end{align*}
\] \quad \{ \text{(137)} \}

by substituting system (136) into system (137) one obtains Leapfrog with integer steps:

\[
x_{n+1} = x_n + \nu_n \Delta t + \frac{1}{2} a_n \Delta t^2
\] \quad \{ \text{(138)} \}

\[
\nu_{n+1} = \nu_n + \frac{1}{2} (a_n + a_{n+1}) \Delta t
\] \quad \{ \text{(139)} \}

→ so-called Verlet integration (see next slides)
Higher order methods

for that purpose: Taylor expansion of $x_{n-1}$ (negative time step):

\begin{align*}
x_{n-1} &= x_n - v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 - O((\Delta t)^3) \quad (140) \\
+ \ x_{n+1} &= x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O((\Delta t)^3) \quad (141) \\
= \ x_{n+1} + x_{n-1} &= 2x_n + a_n (\Delta t)^2 + O((\Delta t)^4) \quad (142) \\
\Rightarrow \ x_{n+1} &= 2x_n - x_{n-1} + a_n (\Delta t)^2 \quad (143)
\end{align*}
Analogously:

\[
\begin{align*}
    x_{n+1} &= x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O((\Delta t)^3) \quad (144) \\
    - (x_{n-1} &= x_n - v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 - O((\Delta t)^3)) \quad (145) \\
    = x_{n+1} - x_{n-1} &= 2v_n \Delta t + O((\Delta t)^3) \quad (146) \\
    \Rightarrow v_n &= \frac{x_{n+1} - x_{n-1}}{2\Delta t} \quad \text{(Verlet)} \quad (147)
\end{align*}
\]

→ method of 2nd order in \( v \) and 3rd order in \( x \)

But:

- not self starting (needs start values \( x_0, x_1 = x_0 + v_0 + \frac{1}{2} a_0 \Delta t^2 \), see above)

- Eq. (147) contains differences of two values of same order of magnitude and expected \( \Delta x \ll x \) → round-off errors possible (subtractive cancelation)
Therefore, from Eq. (147)

\[ x_{n-1} = x_{n+1} - 2v_n \Delta t \]  
insert in Eq. (143):

\[ x_{n+1} = 2x_n - x_{n+1} + 2v_n \Delta t + a_n(\Delta t)^2 \]  

(148)

(149)

Solve for \( x_{n+1} \), yields:

### Velocity Verlet

\[ x_{n+1} = x_n + v_n \Delta t + \frac{1}{2}a_n(\Delta t)^2 \]  

(150)

\[ v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t \]  

(151)

→ self-starting

→ minimizes round-off errors (no differences)

→ 4th order in \( x \) (why? → Eq. (144) & (145))
Eq. (151) results from Eq. (147) for $v_{n+1}$:

$$v_{n+1} = \frac{x_{n+2} - x_n}{2\Delta t}$$  \hfill (152)

and

$$x_{n+2} = 2x_{n+1} - x_n + a_{n+1}(\Delta t)^2 \text{ from Eq. (143)}$$  \hfill (153)

$$\Rightarrow v_{n+1} = \frac{2x_{n+1} - x_n + a_{n+1}(\Delta t)^2 - x_n}{2\Delta t}$$  \hfill (154)

$$= \frac{x_{n+1} - x_n}{\Delta t} + \frac{1}{2}a_{n+1}(\Delta t)^2 \quad \& \quad x_{n+1} \text{ from Eq. (150)}$$

$$= \frac{x_n + v_n \Delta t + \frac{1}{2}a_n(\Delta t)^2 - x_n}{\Delta t} + \frac{1}{2}a_{n+1}\Delta t$$  \hfill (155)

$$= v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t$$  \hfill (156)
Alternatively:
(developed for liquid particles in a Lennard-Jones potential)

**Beeman method (Schofield 1973; Beeman 1976)**

\[ x_{n+1} = x_n + v_n \Delta t + \frac{1}{6} (4a_n - a_{n-1})(\Delta t)^2 \]  
\[ v_{n+1} = v_n + \frac{1}{6} (2a_{n+1} + 5a_n - a_{n-1})\Delta t \]

→ not self-starting
→ locally: \( O(\Delta t)^4 \) in \( x \) and \( O(\Delta t)^3 \) in \( v \), globally \( O(\Delta t)^3 \)
→ better energy conservation than for Verlet, but more calculation steps

even better: → Runge-Kutta method of 4th order
The Runge-Kutta method
Remember:

**Euler-Richardson method (Euler-halfstep method)**

\[
a_n = \frac{F(x_n, v_n, t_n)}{m} \tag{159}
\]

\[
v_M = v_n + a_n \frac{1}{2} \Delta t \tag{160}
\]

\[
x_M = x_n + v_n \frac{1}{2} \Delta t \tag{161}
\]

\[
a_M = \frac{F \left( x_M, v_M, t_n + \frac{1}{2} \Delta t \right)}{m} \tag{162}
\]

\[
v_{n+1} = v_n + a_M \Delta t \tag{163}
\]

\[
x_{n+1} = x_n + v_M \Delta t \tag{164}
\]

→ calculation of \( F \) or \( a \), respectively, for the whole step at the “midpoint” of the interval, instead of using the values from the beginning

≌ Runge-Kutta method 2nd order
We will refine the halfstep method by using more supporting points:
With the Runge-Kutta method\textsuperscript{†} the initial value problem

\[ \frac{dy}{dx} = y' = f(x, y), \quad y(x_0) = y_0 \]  

(165)

is solved by calculating approximate values $y_i$ at selected supporting points $x_i$ to obtain the wanted $y(x)$. These $y_i$ are calculated with help of the following scheme (cf. Bronstein), where also only linear terms are calculated, but in form of a “polygonal line”:

- supporting point at the beginning and at the end of the interval
- two additional supporting points in the middle of the interval with doubled weight

\textsuperscript{†}Carl Runge (1856-1927), Wilhelm Kutta (1867-1944)
Runge-Kutta method of 4th order IV

Move from $x_0$ to $x_i = x_0 + ih$ (step size $h$, $i = 0, 1, 2, \ldots$) → single step method

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$k = h \cdot f(x, y) = h \cdot dy/dx$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$k_1$</td>
</tr>
<tr>
<td>$x_0 + h/2$</td>
<td>$y_0 + k_1/2$</td>
<td>$k_2$</td>
</tr>
<tr>
<td>$x_0 + h/2$</td>
<td>$y_0 + k_2/2$</td>
<td>$k_3$</td>
</tr>
<tr>
<td>$x_0 + h$</td>
<td>$y_0 + k_3$</td>
<td>$k_4$</td>
</tr>
</tbody>
</table>

$x_1 = x_0 + h$  
$y_1 = y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

Cf.: Simpson’s rule\(^\dagger\) (actually Kepler’s rule, “Keplersche Fassregel”) for integration of $y'(x)$ via a parabola:

$$\int_a^b y'(x)dx \approx \frac{b - a}{6} \left( y'(a) + 4 y' \left( \frac{a + b}{2} \right) + y'(b) \right)$$  \hspace{1cm} (166)

\(^\dagger\)Thomas Simpson (1710-1761)
For the equations of motion this means therefore:

\[ \vec{k}_{1v} = \vec{a}(\vec{x}_n, \vec{v}_n, t) \Delta t \quad (= \vec{a}_{\text{grav.}}(\vec{x}_n) \Delta t \text{ in our case}) \quad (167) \]

\[ \vec{k}_{1x} = \vec{v}_n \Delta t \quad (168) \]

\[ \vec{k}_{2v} = \vec{a} \left( \vec{x}_n + \frac{\vec{k}_{1x}}{2}, \vec{v}_n + \frac{\vec{k}_{1v}}{2}, t_n + \frac{\Delta t}{2} \right) \Delta t \quad (169) \]

\[ \vec{k}_{2x} = \left( \vec{v}_n + \frac{\vec{k}_{1v}}{2} \right) \Delta t \quad (170) \]

\[ \vec{k}_{3v} = \vec{a} \left( \vec{x}_n + \frac{\vec{k}_{2x}}{2}, \vec{v}_n + \frac{\vec{k}_{2v}}{2}, t_n + \frac{\Delta t}{2} \right) \Delta t \quad (171) \]

\[ \vec{k}_{3x} = \left( \vec{v}_n + \frac{\vec{k}_{2v}}{2} \right) \Delta t \quad (172) \]

\[ \vec{k}_{4v} = \vec{a}(\vec{x}_n + \vec{k}_{3x}, \vec{v}_n + \vec{k}_{3v}, t + \Delta t) \Delta t \quad (173) \]

\[ \vec{k}_{4x} = (\vec{v}_n + \vec{k}_{3v}) \Delta t \quad (174) \]
Adaptive stepsize: step doubling

1. calculate new coordinates \((\vec{x}, \vec{v})\) via two Runge-Kutta steps each with \(\Delta t\)
2. calculate new coordinates \((\vec{x}, \vec{v})'\) via one Runge-Kutta step with \(2\Delta t\)

→ calculation overhead increases only by \(11/8\), because of same derivatives on the beginning of the interval

Now, if

\[
\frac{|(x, v) - (x, v)'|}{|(x, v)|} \geq \epsilon_{\text{max}} \tag{177}
\]

with an accuracy criterion \(\epsilon_{\text{max}} \rightarrow \text{decrease stepsize } \Delta t\)

If

\[
\frac{|(x, v) - (x, v)'|}{|(x, v)|} \leq \epsilon_{\text{min}} \quad \text{mit } \epsilon_{\text{min}} < \epsilon_{\text{max}} \tag{178}
\]

→ increase \(\Delta t\)
Predictor-corrector method

First prediction of the new position, e.g.:

\[ x_p = x_{n-1} + 2v_n \Delta t \]  \hfill (179)

→ yields acceleration \( a_p \) → corrected position by trapezoidal rule:

\[
\begin{align*}
    v_{n+1}^0 &= v_n + \frac{1}{2} (a_p + a_n) \Delta t \\
    x_{n+1}^0 &= x_n + \frac{1}{2} (v_{n+1} + v_n) \Delta t 
\end{align*}
\]  \hfill (180, 181)

→ yields better value for \( a_{n+1} \) and hence

\[
\begin{align*}
    v_{n+1}^1 &= v_n + a_{n+1} \Delta t \\
    x_{n+1}^1 &= x_n + v_{n+1} \Delta t 
\end{align*}
\]  \hfill (182, 183)

repeated iteration until \(|x_{n+1}^{k+1} - x_{n+1}^k| < \epsilon \) with intended accuracy \( \epsilon \)
Especially interesting for interactions of several bodys (*few-body problem*):

- resonances in planetary systems
- influence by one-time passage of a star
- influence of the galactic gravitational potential

**Requires:**

- high numerical accuracy
- flexibility
- high computation rate

**Idea:** combination of

- modified midpoint method
- Richardson extrapolation
- extrapolation via rational functions

→ Bulirsch-Stoer method (Stoer & Bulirsch 1980)

cf. Numerical Recipes

†Roland Bulirsch (*1932), Josef Stoer (*1934)
Bulirsch-Stoer method II

Modified midpoint method

For an ODE \( dx/dt = f(t, x) \) over a time step \( H = Nh \) with \( N \) equidistant sub-steps

\[
\begin{align*}
    x_0 &= x(t) \quad \text{(184)} \\
    x_1 &= x_0 + hf(t, x_0) \quad \text{(185)} \\
    &\quad \ldots \quad \text{(186)} \\
    x_n &= x_{n-2} + 2hf(t + [n - 1]h, x_{n-1}) \quad n = 2, \ldots, N \quad \text{(187)} \\
    x(t + H) &\approx \tilde{x} = \frac{1}{2} [x_N + x_{N-1} + hf(t + H, x_N)] \quad \text{(188)}
\end{align*}
\]

→ 2nd order method, but with only one derivative per \( h \)-(sub)step (where 2nd order Runge-Kutta has two derivatives per step)
Gragg\(\dagger\) (1965): error in Eq. (188) \(\rightarrow\) even power series:

\[
\tilde{x} - x(t + H) = \sum_{i=1}^{\infty} \alpha_i h^{2i}
\]

\(\rightarrow\) for even \(N\) all odd error terms cancel out
Let \(x_{N/2}\) the result for \(x(t + H)\) with half the number of steps:

\[
x(t + H) \approx \frac{4\tilde{x}_N - \tilde{x}_{N/2}}{3}
\]

\(\rightarrow\) 4th order accuracy (as for RK4), but only with 1.5 derivatives
(RK4: 4)

\(\dagger\)William B. Gragg (1936-2016)
Richardson extrapolation

Idea: result $x(t + H)$ is an analytic function of $h$ with $h = H/N$:

1. calculate $x_{t+H}(h = 2, 4, 6, \ldots)$
2. fit function $x_{t+H}(h)$ to $x_{t+H}(N = 2)$, $x_{t+H}(N = 4)$, \ldots
3. extrapolate $x_{t+H}(h \to 0)$, corresponding to $N \to \infty$
Extrapolation via polynomial

Compute \( k \)-times \( x_{t+H} \) with \( N = 2, 4, 6, \ldots \):

\[
x_{t+H}(h) = a_0 + a_1 h + a_2 h^2 + \ldots + a_k h^{k-1}
\]

(191)

where following Lagrange

\[
x_{t+H}(h) = \frac{(h - h_2)(h - h_3) \ldots (h - h_k)}{(h_1 - h_2)(h_1 - h_3) \ldots (h_1 - h_k)} x_{t+H}(h_1) + \frac{(h - h_1)(h - h_3) \ldots (h - h_k)}{(h_2 - h_1)(h_2 - h_3) \ldots (h_2 - h_k)} x_{t+H}(h_2) + \ldots + \frac{(h - h_1)(h - h_2) \ldots (h - h_{k-1})}{h_k - h_1)(h_k - h_2) \ldots (h_k - h_{k-1})} x_{t+H}(h_k)
\]

(192, 193, 194)

In the original Bulirsch-Stoer method: rational function \((P(h)/Q(h))\) instead of Lagrange polynomial
Consider an $N$-body system with

$$\frac{d^2 \vec{x}_i}{dt^2} = -\sum_{j=1; j \neq i}^{N} \frac{G m_j (\vec{x}_i - \vec{x}_j)}{|\vec{x}_i - \vec{x}_j|^3}$$

(195)

Problem: $a_{ij} \propto \frac{1}{r_{ij}^2}$ for very small distances $r_{ij}$ (close encounters)

$\rightarrow$ small distances $\rightarrow$ large accelerations $\rightarrow$ requires small $\Delta t$

$\rightarrow$ slows down calculations & increases numerical accumulation error

Possibly uncomplicated for *one time* encounters

But in star clusters:

$\rightarrow$ formation of close binaries $\rightarrow$ periodic

So-called “binary hardening”: transfer of the energy of the binary system to the cluster by consecutive close encounters
M 62 (NGC 6266). *Left*: optical HST. *Right*: X-ray CHANDRA

→ above-average rate of close binary systems (e.g., low-mass X-ray binaries) in globular clusters (Pooley et al. 2003)
obvious (and inaccurate) idea: “softening” term in Eq. (195):

\[
\vec{F}_{ij} = \frac{G m_i m_j (\vec{r}_j - \vec{r}_i)}{(\epsilon^2 + |\vec{r}_i - \vec{r}_j|^2)^{3/2}}
\]

such that

\[
\max |\vec{a}_j| = \frac{2G m_i}{3^{3/2} \epsilon^2} \text{ at } r = \frac{1}{\sqrt{2}} \epsilon
\]

→ adaptive $\Delta t$ not arbitrarily small; but: close binary orbits and passages not resolvable

When is “softening” applicable?

→ if close encounters are irrelevant
→ collisionless systems, e.g., galaxy
Illustration: distances in a galaxy

Galaxy: $\varnothing \approx 10^{23}$ cm with $10^{11}$ stars with $R_* \approx 10^{11}$ cm $\rightarrow d \approx 10^{19}$ cm

$10^{11}$ sand grains $\rightarrow$ width of $100 \times$

$\rightarrow$ average distance between sand grains $\approx 10$ km

$\rightarrow t_{*,\text{coll}} \gg t_{\text{Hubble}} \rightarrow$ collisionless

stars perceive only the average gravitational potential of the galaxy
Better: regularization (technique in physics to avoid $\infty$) with help of transformation of spacetime coordinates.

Consider vector $\vec{R}$ between two particles (center of mass frame):

$$\frac{d^2 \vec{R}}{dt^2} = -G(m_1 + m_2) \frac{\vec{R}}{|\vec{R}|^3} + \vec{F}_{12} \quad (199)$$

with external force $\vec{F}_{12} = \vec{F}_1 - \vec{F}_2$ per mass, by other particles

1. regularized time $\tau$

$$dt = R^n d\tau \quad (200)$$

$$\frac{d^2}{dt^2} = \frac{1}{R^{2n}} \frac{d^2}{d\tau^2} - \frac{n}{R^{2n+1}} \frac{dR}{d\tau} \frac{d}{d\tau} \quad (201)$$

$$\frac{d^2 \vec{R}}{d\tau^2} = \frac{n}{R} \frac{dR}{d\tau} \frac{d\vec{R}}{d\tau} - G(m_1 + m_2) \frac{\vec{R}}{R^{3-2n}} + R^{2n} \vec{F}_{12} \quad (202)$$

for $n = 1 \rightarrow \dot{R} \propto t/\tau$ and without $R^{-2}$- singularity, but with $\vec{R}/R$ term (indefinite for $R \rightarrow 0$)
therefore:

2. regularized distance $u$, initially only for 1 dimension (already known by Euler), without external force (see Aarseth 2003):

$$\frac{d^2}{d\tau^2} = \frac{1}{R} \left( \frac{dR}{d\tau} \right)^2 - G(m_1 + m_2)$$  \hfill (203)

and with conservation of energy, total energy $h$ per reduced mass $\mu = m_1 m_3/(m_1 + m_2)$:

$$h = \frac{1}{2} \left( \frac{dR}{dt} \right)^2 - \frac{G}{R}(m_1 + m_2)$$  \hfill (205)
with

\[ \frac{dR}{dt} = \frac{1}{R} \frac{dR}{d\tau} \] (206)

\[ \Rightarrow \frac{d^2 R}{d\tau^2} = 2hR + G(m_1 + m_2) \] (207)

→ no more singularities. With \( u^2 = R \):

\[ \frac{d^2 u}{d\tau^2} = \frac{1}{2} hu \] (208)

→ easy to integrate

→ method: change from \((x, t)\) to \((u, \tau)\) below some distinct distance (for 1d collision!)
Regularization VIII

in 2 dimensions (Levi-Civita 1904):

\[ x = u_1^2 - u_2^2 \]  
\[ y = 2u_1u_2 \]  
\[ \text{or} \quad \vec{R} = \mathcal{L}\vec{u} \]  

where \( \mathcal{L} = \mathcal{L}(\vec{u}) = \begin{pmatrix} u_1 & -u_2 \\ u_2 & u_1 \end{pmatrix} \)

With the following properties:

\[ \mathcal{L}(\vec{u})^T \mathcal{L}(\vec{u}) = R I \]  
\[ \frac{d}{dt} \mathcal{L}(\vec{u}) = \mathcal{L} \left( \frac{d\vec{u}}{dt} \right) \]  
\[ \mathcal{L}(\vec{u})\vec{v} = \mathcal{L}(\vec{v})\vec{u} \]  
\[ \vec{u} \cdot \vec{u} \mathcal{L}(\vec{v})\vec{v} - 2\vec{u} \cdot \vec{v} \mathcal{L}(\vec{u})\vec{v} + \vec{v} \cdot \vec{v} \mathcal{L}(\vec{u})\vec{u} = 0 \]
With help of Eqn. (214 & 215) coordinates change to

\[
\frac{d \vec{R}}{d \tau} = 2 \mathcal{L}(\vec{u}) \frac{d \vec{u}}{d \tau} \tag{217}
\]

\[
\frac{d^2 \vec{R}}{d \tau^2} = 2 \mathcal{L}(\vec{u}) \frac{d^2 \vec{u}}{d \tau^2} + 2 \mathcal{L} \left( \frac{d \vec{u}}{d \tau} \right) \frac{d \vec{u}}{d \tau} \tag{218}
\]

Hence in Eq. (202) with \( n = 1 \) and with Eq. (216) and some transformations:

\[
2 \vec{u} \cdot \vec{u} \mathcal{L}(\vec{u}) \frac{d^2 \vec{u}}{d \tau^2} - 2 \frac{d \vec{u}}{d \tau} \cdot \frac{d \vec{u}}{d \tau} \mathcal{L}(\vec{u}) \vec{u} + G(m_1 + m_2) \mathcal{L}(\vec{u}) \vec{u} = (\vec{u} \cdot \vec{u})^3 \vec{F}_{12} \tag{219}
\]

Further transformations lead to a form without singularities and indefinitenesses:

\[
\frac{d^2 \vec{u}}{d \tau^2} = \frac{1}{2} h \vec{u} + \frac{1}{2} R \mathcal{L}^T(\vec{u}) \vec{F}_{12} \tag{220}
\]
Binary star without external forces $\vec{F}_{12} \rightarrow$ energy $h$ conserved

Binary star with external forces:

$$h = \left[ 2 \frac{d\vec{u}}{d\tau} \cdot \frac{d\vec{u}}{d\tau} - G(m_1 + m_2) \right] / R$$

(221)

The time evolution in usual coordinates

$$\frac{d}{dt} \left[ \frac{1}{2} \left( \frac{dR}{dt} \right)^2 - \frac{G}{R}(m_1 + m_2) \right] = \frac{d\vec{R}}{dt} \cdot \vec{F}_{12}$$

(222)

after transformation

$$\frac{dh}{d\tau} = 2 \frac{d\vec{u}}{d\tau} \cdot \mathcal{L}(\vec{u}) \vec{F}_{12}$$

(223)

can be solved continuously for $R = 0$
Application of the 2d solution to the so-called Pythagoraian three-body problem ($\vec{L} = 0$) in Szebehely & Peters (1967):

because of $\vec{L} = 0$ three-body collision possible → does not occur
Regularization XII

Regularization for 3 dimensions (Kustaanheimo & Stiefel 1965) requires transformation to 4d coordinates:

\[ R_1 = u_1^2 - u_2^2 - u_3^2 + u_4^2 \]  
\[ R_2 = 2(u_1 u_2 - u_3 u_4) \]  
\[ R_3 = 2(u_1 u_3 + u_2 u_4) \]  
\[ R_4 = 0 \]

and \( \vec{R} = \mathcal{L}(\vec{u})\vec{u} \), such that

\[ \mathcal{L} = \begin{bmatrix} u_1 & -u_2 & -u_3 & u_4 \\ u_2 & u_1 & -u_4 & -u_3 \\ u_3 & u_4 & u_1 & u_2 \\ u_4 & -u_3 & u_2 & -u_1 \end{bmatrix} \]  

→ yields again equations similar to (220) & (223)

see Bodenheimer et al. (2007)
Problems:

1. number of interactions is $N(N - 1)/2 \propto \mathcal{O}(N^2)$

2. multiple timescales for adaptive time steps for each particle $i$:

$$\Delta t_i \simeq k \sqrt{\frac{1}{|\vec{a}_i|}}$$

(229)

with acceleration $\vec{a}_i$ and small factor $k$
possible solutions:

1. Tree method (Barnes & Hut 1986, 1989) → hierarchical structure and calculation of multipoles of the potential → $O(N \log N)$

Holmberg (1941) even $O(N)$ with help of light bulbs on 2d grid (flux $\propto 1/r^2$)

2. Leapfrog method (2nd order integ.):

\begin{align*}
\vec{r}_i^{n+1/2} &= \vec{r}_i^{n-1/2} + \Delta t_i \vec{v}_i^n \quad (230) \\
\vec{v}_i^{n+1} &= \vec{v}_i^n + \Delta t_i \vec{a}_i^{n+1/2} \quad (231)
\end{align*}

The gravitational effect exerted by the star cluster and the single star B on star A can be approximated by a point mass. (from Barnes-Hut Galaxy Simulator)

with time step doubling
\[ \Delta t_i = \Delta t_{\text{max}}/2^{n_i} \text{ for each particle } i \]
Example time step doubling

col. A: time step $\Delta t/2$, col. B: time step $\Delta t$

starting via

\[
\tilde{r}_i^{n+1/2} = \tilde{r}_i^n + \frac{1}{2} \Delta t_i \tilde{v}_i^n + \frac{1}{8} \Delta t_i^2 \tilde{a}_i^n \quad \text{for } i = A, B. \quad (232)
\]

Hence, we get $\tilde{r}_A(\Delta t/4)$ and $\tilde{r}_B(\Delta t/2)$ and from that

$\tilde{a}_A(A[\Delta t/4], B[\Delta t/2])$ and analogously $\tilde{a}_B \rightarrow$ time asymmetry

$\tilde{a}_A \rightarrow \tilde{v}_A(\Delta t/2) \rightarrow \tilde{r}_A(3/4\Delta t)$

$\tilde{a}_A(A[3/4\Delta t], B[\Delta t/2]) \rightarrow \tilde{v}_A(\Delta t) \rightarrow$ reversed time asymmetry

Averaging of $\tilde{r}_A(\Delta t/4), \tilde{r}_A(3/4\Delta t)$ to $\tilde{r}_A(\Delta t/2)$, then

$\rightarrow \tilde{a}_B(A[\Delta t/2], B[\Delta t/2]) \rightarrow \tilde{v}_B(\Delta t)$

from $\tilde{v}_A(\Delta t), \tilde{v}_B(\Delta t) \rightarrow \tilde{r}_A(5/4\Delta t)$

i.e. next cycle starts
Differential equations
One can classify differential equations regarding their 

- order, so the degree of the highest derivative. General form of a first-order differential equation:

\[
\frac{dy}{dt} = f(y, t) \tag{233}
\]

for any arbitrary function \( f \), e.g., \( \frac{dy}{dt} = 2ty^8 - t^5 + \sin(y) \). A second-order differential equation has the form:

\[
\frac{d^2y}{dt^2} + \lambda \frac{dy}{dt} = f(t, \frac{dy}{dt}, y) \tag{234}
\]

and so on.
Reduction

By introducing auxiliary variables/functions, every higher order differential equation can be reduced to a set of \textit{first-order} differential equations

\[ y^{(m)}(x) = f(x, y(x), y^{(1)}(x), \ldots, y^{(m-1)}(x)) \]  
\[ \rightarrow z_1(x) := y(x) \]  
\[ z_2(x) := y^{(1)}(x) \]  
\[ \vdots \]  
\[ z_m(x) := y^{(m-1)}(x) \]

\[
\begin{align*}
\dot{z}' & = \begin{bmatrix}
z'_1 \\
\vdots \\
z'_m
\end{bmatrix} = \begin{bmatrix}
z_2 \\
\vdots \\
f(x, z_1, z_2, \ldots, z_m)
\end{bmatrix}
\end{align*}
\]
One can distinguish

- **ordinary differential equations (ODE)**, where only one independent variable is explicitly involved (typically time or location), e.g.:

\[
\frac{dP}{dr} = -\rho(r) g(r)
\]  
(241)

- **partial differential equations (PDE)**, where the solution depends at least on two variables, e.g.:

\[
\Delta \rho = \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \rho(x, y, z) = f(x, y, z)
\]  
(242)

The theory and (numerical) solution of PDEs is more complicated than for ODE.
Moreover, there are the classes of

- **linear** differential equations: only the first power of $y$ or $d^ny/dt^n$ occurs, e.g.:

$$\left(\frac{1}{c^2} - \Delta\right) u = 0$$  \hspace{1cm} (243)

→ special property: *law of linear superposition*, linear combinations of solutions are also solutions:

$$u_2(x, y, z, t) = au_0(x, y, z, t) + bu_1(x, y, z, t)$$  \hspace{1cm} (244)

→ unperturbed superposition of waves
• *nonlinear* differential equations: contain higher powers or other functions of $y$ or $d^n y/dt^n$, e.g.:

\[
\frac{d\theta^2}{dt^2} = \frac{l}{g} \sin \theta \tag{245}
\]

→ clear: linear combinations of solutions are not automatically solutions too, e.g.

\[
\frac{dy}{dt} = \lambda y(t) - \lambda^2 y^2(t) \tag{246}
\]

\[
y(t) = \frac{a}{1 + be^{-\lambda t}} \text{ one solution} \tag{247}
\]

\[
y_1(t) = \frac{a}{1 + be^{-\lambda t}} + \frac{c}{1 - de^{-\lambda t}} \text{ not a solution} \tag{248}
\]

nonlinear equations in general became feasible with the rise of computers
As general solution of (ordinary) differential equation contains arbitrary constant per order, problems involving differential equations can be characterized by the type of conditions:

1. *Initial* values/conditions must be given: constant for 1st order differential equation (usually time-dependent) fixed by giving \( y(t) \) for some time \( t_0 \), so giving \( y_0 = y(t_0) \); for 2nd order by giving additionally \( y'(t_0) \) and so on (Note, that we solve usually for \( t > t_0 \), but this is not a requirement), e.g., Kepler problem

\[
\ddot{v}(t) = \ddot{r}(t) \quad \& \quad \dddot{a}(t) = \dot{v}(t) = F_G(r)/m \\
x(t_0) = x_0, y(t_0) = 0; \ v_x(t_0) = 0, \ v_y(t_0) = v_{y,0}
\]  

(249)  

(250)

For the initial value problem (Cauchy problem), the theorem by Picard-Lindelöf guarantees a unique solution:
Existence and uniqueness of the solution for the initial value problem

\[ y' = f(y, x), \quad y(x_0) = y_0 \quad (251) \]

If \( f \) is continuous on the stripe \( S := \{(x, y) | a \leq x \leq b, y \in \mathbb{R}^n\} \) with finite \( a, b \) and a constant \( L \), such that

\[ ||f(x, y_1) - f(x, y_2)|| \leq L||y_1 - y_2|| \quad (252) \]

for all \( x \in [a, b] \) and for all \( y_1, y_2 \in \mathbb{R}^n \) (Lipschitz continuous), then exists for all \( x_0 \in [a, b] \) and for all \( y_0 \in \mathbb{R}^n \) a unique function \( y(x) \) for \( x \in [a, b] \) with

1. \( y(x) \) is continuous and continuously differentiable for \( x \in [a, b] \);
2. \( y'(x) = f(x, y(x)) \) for \( x \in [a, b] \);
3. \( y(x_0) = y_0 \)
Note that the Lipschitz condition (bounded slope) of \( f(y, x) \) is required for uniqueness, e.g., \( y'(x) = \sqrt{|x|} \) with \( y(0) = 0 \) is fulfilled by \( y_1(x) \equiv 0 \) and also by \( y_2(x) = \frac{x^2}{4} \), that is because \( f'(y, x) = \frac{1}{\sqrt{|x|}} \) and hence \( \lim_{x \to 0} f' = \infty \). Without Lipschitz condition the Peano existence theorem guarantees the existence of a solution.

Proof idea
Integrating Eq. (251) gives a fixed point equation:

\[
y(x) - y(x_0) = \int_{x_0}^{x} f(s, y(s)) \, ds
\]

(253)

with Picard-Lindelöf iteration

\[
\phi_0(x) = y_0 \quad \text{and} \quad \phi_{k+1} = y_0 + \int_{x_0}^{x} f(s, \phi_k(s)) \, ds
\]

(254)
Boundary values IV

Example: Picard iteration

For the Cauchy problem

\[ y'(x) = 1 + y(x)^2, \quad y(x_0) = y(0) = 0 \] (255)

\[ \phi_0(x) = 0 \] (256)

\[ \phi_1(x) = 0 + \int_0^x (1 + 0^2) \, ds = x \] (257)

\[ \phi_2(x) = 0 + \int_0^x (1 + s^2) \, ds = x + \frac{1}{3}x^3 \] (258)

→ Taylor series expansion of \( y(x) = \tan(x) \)

so following Banach fixed point theorem \( \phi_k \) converges uniquely to the solution \( y(x) \). The existence of \( y(x) \) (Peano) is proven by constructing a piecewise continuous function with help of the Euler method (polygonal curve) that converges uniformly for \( \Delta x \to 0 \).
boundary values/conditions can be given, (additionally to initial conditions) to restrict further the solutions, i.e., constrain it to fixed values at the boundaries of the solution space, usually for 2nd order differential equation

\[ u''(x) = f(u, u', x) \]  \hspace{1cm} (259)

where \( u \) or \( u' \) is given at boundaries, by transformation, e.g.,

\[ x' = (x - x_1)/(x_2 - x_1) \]  \hspace{1cm} (260)

at \( x = 0 \) and \( x = 1 \). Then \( \rightarrow \) 4 possible types of boundary conditions

1. \( u(0) = u_0 \) and \( u(1) = u_1 \)
2. \( u(0) = u_0 \) and \( u'(1) = v_1 \)
3. \( u'(0) = v_0 \) and \( u(1) = u_1 \)
4. \( u'(0) = v_0 \) and \( u'(1) = v_1 \)
Usually: reduce to set of 1st order differential equations and start integration with given $u(0)$ and $v(0)$. But for boundary-value problem: only $u(0)$ or $v(0)$ given, not sufficient for any algorithm.
**Example: Boundary values**

First two equations of stellar structure (e.g., for white dwarf)

\[
\frac{\partial r}{\partial m} = \frac{1}{4\pi r^2 \rho} \quad \text{mass continuity} \quad (261)
\]

\[
\frac{\partial P}{\partial m} = -\frac{G M}{4\pi r^4} \quad \text{hydrostatic equilibrium} \quad (262)
\]

+ equation of state \( P(\rho) \) (e.g., ideal gas \( P = RT\rho/\mu \)), and boundary values

\[
\text{center} \quad m = 0 : r = 0 \quad (263)
\]

\[
\text{surface} \quad m = M : \rho = 0 \rightarrow P = 0 \quad (264)
\]

→ solve for \( r(m) \), specifically for \( R_* = r(m = M_*) \)
eigenvalue problems: solution for selected parameters (\(\lambda\)) in the equations; usually even more complicated and solution not always exist, sometimes trial-and-error search necessary.

E.g.,

\[
\frac{d^2 u}{dx^2} = f(u, u', x, \lambda) \quad (265)
\]

for eigenvalue lambda plus a set of boundary conditions. Eigenvalue \(\lambda\) can only have some selected values for valid solution.

E.g., Schrödinger equation for particle confined in a potential: eigenfunctions \(\rightarrow\) wavefunction; eigenvalues \(\rightarrow\) discrete energies
Eigenvalue problem: Stationary elastic waves

Displacement $u(x)$ by

$$u'' = -k^2 u \quad (266)$$

Allowed values of wavevector $k = \omega / c \rightarrow$ eigenvalues of the problem
both ends fixed: $u(0) = u(1) = 0$ or one end fixed, other end free:
$u(0) = 0$ and $u'(1) = 0$. Fortunately, analytical solutions:

$$u_n(x) = \sqrt{2} \sin k_n x \quad \& \quad k_n = n\pi \quad n = \pm 1, \pm 2, \ldots \quad (267)$$

Moreover, complete solution of longitudinal waves along elastic rod: linear
combination of all eigenfunctions with their initial solutions (fixing $c_n$)

$$u(x, t) = \sum_{n=-\infty}^{\infty} c_n u_n(x) e^{i n\pi ct} \quad (268)$$
e.g., for boundary-value problem $u'' = f(u, u', x)$ with $y_1 \equiv u$ and $y_2 \equiv u'$
\[
\frac{dy_1}{dx} = y_2 \quad (269)
\]
\[
\frac{dy_2}{dx} = f(y_1, y_2, x) \quad (270)
\]
plus boundary conditions, e.g., $u(0) = y_1(0) = u_0$ and $u(1) = y_1(1) = u_1$.
Idea: introduce adjustable parameter, so that we have an initial value problem. E.g., $u'(0) = \delta \rightarrow$ together with given $u(0) = u_0$; integrate for given intial values up to $x = 1$ with result $u(1) = u_\delta(1)$, so that
\[
F(\delta) = u_\delta(1) - u_1 \overset{!}{=} 0 \quad (271)
\]
$\rightarrow$ use root search algorithm to determine (approximative) $\delta$
The shooting method II

Shooting method for boundary value problem (Stoer & Bulirsch 2005)

\[ u''(x) = \frac{3}{2} u^2, \quad u(0) = 4, \quad u(1) = 1 \]  
(272)

set \( y_1(0) = 4, \quad y_2(0) = \delta = -1, \ldots -70 \)  
(273)

\[ \rightarrow y_{1,k+1} = y_{1,k} + \Delta x \cdot y_{2,k} \]  
(274)

\[ y_{2,k_1} = y_{2,k} + \Delta x \cdot 3./2. \cdot y_{1,k}^2 \]  
(275)

plot \( F(\delta) = y_1,n - u(1) \), roots give missing initial values \( u'(0) \)
Similarly, for given

- \( u'(0) = v_0 \) and \( u(1) = u_1 \) \( \rightarrow \) \( u(0) = \delta \), find root of \( F(\delta) = u_\delta(1) - u_1 \)
- \( u'(0) = v_0 \) and \( u'(1) = v_1 \) \( \rightarrow \) \( F(\delta) = u'_\delta(1) - v_1 \)

Moreover, for eigenvalue problem:

- if \( u(0) = u_0 \) and \( u(1) = u_1 \) given, start integration with \( u'(0) = \delta \)
  with small \( \delta \)
- search root \( F(\lambda) = u_\lambda(1) - u_1 \) \( \rightarrow \) approximated eigenvalue \( \lambda \) and
eigenvector from normalized solution \( u_\lambda(x) \) \( \rightarrow \) \( \delta \) automatically
modified to be correct \( u'(0) \) through normalization of eigenfunctions
Direct solution of 2nd order ODE I

Although, always possible \( \rightarrow \) reduce 2nd order ODE to set of coupled 1st order ODEs, however, sometimes direct solution has advantages

Example: Radiative Transfer Equation

For the 1d case:

\[
\frac{dl^{\pm}}{d\tau} = \pm(S - l^{\pm}), \quad d\tau = \kappa dz
\] (276)

with inward \((-\)) and outward \(+(\) intensities \(l = \frac{dE}{d\Omega dAdtd\nu}\), optical depth \(\tau\) and source function \(S = \frac{\eta}{\kappa}\).

Introducing Feautrier variables (Schuster 1905; Feautrier 1964):

\[
u = \frac{1}{2}(l^+ + l^-) \quad \text{(intensity-like)} \] (277)

\[
u = \frac{1}{2}(l^+ - l^-) \quad \text{(flux-like)} \] (278)
we get system of two coupled 1st order ODE:

\[
\frac{du}{d\tau} = v \quad \text{and} \quad \frac{dv}{d\tau} = u - S
\]  

or, combining them:

\[
\frac{d^2u}{d\tau^2} = u - S
\]  

discretization on a \( \tau \) grid with numerical derivatives (see below):

\[
\left. \frac{d^2u}{d\tau^2} \right|_{\tau_i} \approx \frac{\frac{du}{d\tau}\big|_{\tau_{i+1/2}} - \frac{du}{d\tau}\big|_{\tau_{i-1/2}}}{\tau_{i+1/2} - \tau_{i-1/2}} \approx \frac{\frac{u_{i+1} - u_i}{\tau_{i+1} - \tau_{i}} - \frac{u_i - u_{i-1}}{\tau_{i} - \tau_{i-1}}}{\frac{1}{2}(\tau_{i+1} - \tau_{i}) - \frac{1}{2}(\tau_{i} - \tau_{i-1})}
\]
Direct solution of 2nd order ODE III

→ set of linear equations for $u_i$:

$$A_i u_{i-1} + B_i u_i + C_i u_{i+1} = S_i$$

(282)

with the coefficients

$$A_i = - \left( \frac{1}{2} (\tau_{i+1} - \tau_{i-1})(\tau_i - \tau_{i-1}) \right)^{-1}$$

(283)

$$C_i = - \left( \frac{1}{2} (\tau_{i+1} - \tau_{i-1})(\tau_{i+1} - \tau_i) \right)^{-1}$$

(284)

$$A_i = -1 - A_i - C_i$$

(285)

→ tridiagonal matrix, efficiently solvable by standard linear algebra solvers (e.g., Gauß-Seidel elimination)
Advantage of Feautrier scheme

- direct solution of 2nd order ODE saves memory
- at large optical depths $I^+ \approx I^- \rightarrow$ radiative flux $\sim I^+ - I^-$ inaccurate because of roundoff error, Feautrier scheme uses instead averaged quantities $u, v$ for higher accuracy ($\rightarrow$ stability in an iterative scheme for $S(I), \tau(I))$
Root finding – Iterative techniques
Problem: Finding roots for equations that cannot be solved analytically, i.e. finding $x_0$ for $f(x_0) = 0$

Transcendent equation: quantum states in a square well

The 1d potential $V(x)$ for the Schrödinger equation

$$V(x) = \begin{cases} -V_0, & |x| \leq a \\ 0, & |x| \geq a \end{cases}$$

has bound states with energies $E = -E_B < 0$

$$\sqrt{2m(V_0 - E_B)} \tan \left[ a \sqrt{2m(V_0 - E_B)} \right] = \sqrt{2mE_B}$$

→ e.g., for $2m = 1$, $a = 1$ we want to find the roots $E_B$ of

$$f(E_B) = \sqrt{V_0 - E_B} \tan \left( \sqrt{V_0 - E_B} \right) - \sqrt{E_B} = 0$$
Transcendent equations II

Roots of numerically derived functions

Some functions cannot even be written analytically, e.g.

- $x(t)$ for the Kepler problem
- solutions of the Lane-Emden equation $\theta_n(\xi)$ for $n \neq \{0, 1, 5\}$

→ roots can be found numerically by trial-and-error algorithms, i.e. iteratively until some specified level of precision is reached
Bisection I

→ very stable, but also very slow iterative procedure
→ needs two start values \([x_1, x_2]\) for estimating \(x_0\)
If \(f(x)\) continuous on \([a, b]\) and \(f(a) \cdot f(b) < 0\), then the intermediate value theorem guarantees the existence of an \(x_0 \in [a, b]\) with \(f(x_0) = 0\).

Bisection algorithm

1. start with interval \([x_1, x_2]\) on which \(f(x)\) changes sign (so \(f(x_1) \cdot f(x_2) < 0\) → contains root)
2. choose new \(x_3\) as the midpoint of the interval \(x_3 = \frac{x_1 + x_2}{2}\)
3. calculate \(f(x_3)\): either \(f(x_3)\) is sufficiently close to 0 → root is \(x_3\) or \(x_3\) is a new interval endpoint:
   - if \(f(x_3) \cdot f(x_1) > 0\) → new interval is \([x_3, x_2]\)
   - or if \(f(x_3) \cdot f(x_1) \leq 0\) → new interval is \([x_1, x_3]\)
4. goto step 2

→ nested intervals enclosing the root
The secant method is similar to Newton’s method (see below).

Requirement: \( f(x) \) continuous and
\[ \exists x_0 \in [a, b] \text{ with } f(x_0) = 0. \]

Then: line through \((x_0, f(x_0))\) and \((x_1, f(x_1))\), so that
\[
y = \frac{f(x_1) - f(x_0)}{x_1 - x_0} (x - x_1) + f(x_1)
\]
with root
\[ x = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)} \]

→ new point \((x_2, f(x_2))\) repeat with \(x_1, x_2\) instead of \(x_0, x_1\).
1. start with interval \( x_1 \neq x_2 \) close to the root

2. iterate

\[
    x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)
\]

\( \rightarrow \) superlinear convergence, per iteration about 1.6 more correct digits

\( \rightarrow \) convergence not assured (especially as \( f(x_n) \cdot f(x_{n-1}) \) is not necessary 0)

\( \rightarrow \) numerically limited by subtractive cancellation, as fraction \( \rightarrow 0/0 \)
## Regula falsi method I

→ refinement of bisection by combining it with the secant method

### Regula falsi (False position method)

1. as for bisection: start with interval \([x_1, x_2]\) with \(f(x_1) \cdot f(x_2) < 0\)
2. calculate the zero of the secant
   \[
   x_3 = x_1 - \frac{x_2 - x_1}{f(x_2) - f(x_1)} f(x_1) = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)}
   \]  
   \(\text{(290)}\)
3. if \(f(x_3) = 0\) → stop, else
4. if \(f(x_1) \cdot f(x_3) > 0\) → replace \(x_1 = x_3\)
   if \(f(x_2) \cdot f(x_3) > 0\) → replace \(x_2 = x_3\)
5. goto 2

→ convergence is superlinear (more than one significant digit per iteration)
   as for secant method
→ advantage: numerically stable, no evaluation of derivatives required, computation of function values is reused
→ preferred method for 1d problems
Newton’s method I

or *Newton-Raphson method* (Newton 1669, Raphson 1690) to solve numerically non-linear equations or systems of equations

→ quicker than bisection, but sometimes problematic

Idea: start with approximation \( x_0 \), draw tangent at \((x_0, f(x_0))\), determine intersection with x-axis → new approximation for root

Derivation: evaluate function \( f(x) \) around \( x_0 \) (Taylor expansion)

\[
f(x_0 + \Delta x) \simeq f(x_0) + f'(x_0) \cdot \Delta x
\]  

(291)  

(linear approximation = tangent on \( x_0 \) shall vanish)  

(292)  

\[
\rightarrow f(x_0) + f'(x_0) \cdot \Delta x = 0
\]  

(293)  

\[
\rightarrow \Delta x = -\frac{f(x_0)}{f'(x_0)}
\]  

(294)  

the correction \( \Delta x \) added on \( x_0 \) gives improved guess for root
Newton’s method

\[ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \]  

(295)

Convergence:

If \( f : [a, b] \rightarrow \mathbb{R} \) is a \( C^2 \) function with

1. \( f \) has a root \( \xi \) in \([a, b]\)
2. \( f'(x) \neq 0 \) for \( x \in [a, b] \)
3. \( f \) is convex (\( f'' \geq 0 \)) or concave (\( f'' \leq 0 \)) in \([a, b]\)
4. the iterated \( x_1 \) for \( x_0 = a \) and \( x_0 = b \) are in \([a, b]\)

Then: For any \( x_0 \in [a, b] \) the values \( x_1, x_2, \ldots \) from Eq. (295) are in \([a, b]\)
and the sequence converges monotonically to \( \xi \).
Remarks:

- only locally convergent,

  i.e. result depends on start approximation for $x_0$
  $\to$ Newton fractal for $z^3 - 1 = 0$

- in some situations Newton method may fail (see requirements):
  - if $x_n$ is at local extremum with $f(x_n) \neq 0 \to$ tangent with slope 0, i.e. $f'(x_n) = 0$
    $\to$ infinite correction
    $\to$ solution: start over with different $x_0$
Newton’s method IV

- infinite loop,

\[ f(x) = x^3 - 2x + 2 \]
with \( x_0 = 0 \to f(0) = 2 \),
\( f'(0) = -2 \to x_1 = 0 - \frac{2}{-2} = 1 \)
and
for \( x_0 = 1 \to f(1) = 1 \),
\( f'(1) = 1 \to x_1 = 1 - \frac{1}{1} = 0 \)

\( \rightarrow \) happens if \( x_0 \) in region where \( f(x) \) not “linear enough”
(vizualization may help to find better initial guess)

- convergence is quadratic, i.e. with every step two more significant
digits

- instead of analytic \( f'(x) \) numeric approximation
  \( f'(x_n) \approx \frac{f(x_n+h) - f(x_n)}{h} \)
sufficient \( \rightarrow \) even rough (or constant!) approximation may be sufficient

- if convergent, method is stable
Backtracing
→ solution to some problems (i.e. infinite loop) with large corrections
So: if for new guess \( x_0 + \Delta x \)

\[ |f(x_0 + \Delta x)|^2 > |f(x_0)|^2 \]  \hspace{1cm} (296)

→ backtrack, try smaller guess, e.g., \( x_0 + \Delta x/2 \), if still condition (296), try \( x_0 + \Delta x/4 \) and so on
→ because tangent line will lead to decrease in \( f(x) \), even small step \( \Delta x \) sufficient
Extension to multidimensional case
for multidimensional function \( f : \mathbb{R}^n \rightarrow \mathbb{R}^n \)

\[
f(x + h) = f(x) + J(x) \cdot h + \mathcal{O}(||h||^2)
\]  

(297)

where \( J(x) = f'(x) = \frac{\partial f}{\partial x}(x) \) the Jacobi matrix, the matrix of the partial derivatives w.r.t. \( x \):

\[
J(x) := \left( \frac{\partial f_i}{\partial x_j}(x) \right)_{ij} = \begin{pmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n}
\end{pmatrix}
\]

(298)

Therefore

\[
x_{n+1} = x_n - J^{-1}(x_n) f(x_n) \quad \Delta x_n = -J^{-1}(x_n) f(x_n)
\]  

(299)
Newton’s method VII

As inversion of $J$ is expensive, usually solve system of linear equations:

$$J(x_n)\Delta x_n = -f(x_n)$$  \hspace{1cm} (300)

to get $\Delta x_n$ and then $x_{n+1} = x_n + \Delta x_n$

→ Newton-Raphson method in $n$ dimensions (i.e. system of equations) is expensive, therefore often used: \emph{quasi Newton methods}

**Example: statistical equilibrium**

In the non-LTE case population numbers of ions $n$ from statistical equations with transition rates $P_{ij}$, stationary:

$$\sum_{i \neq j} n_i \ P_{ij} = \sum_{j \neq i} n_j \ P_{ji}$$

with $P_{ij} := - \sum_{j} P_{ji}$ → $n \cdot P(n, J, T_e) = 0$ together with $J = \Lambda S(n)$

→ non-linear system of $N$ equations → $N^3$ derivatives ($N$ derivatives for $N \times N$ rates)
Instead of calculating \( n^3 \) derivatives use modified *secant* equation

\[
x_{k+1} = x_k - f(x_k)B_k^{-1}
\]

with slope \( B_{k+1} = \frac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k} = \frac{\Delta y_k}{\Delta x_k} \)

But: Eq. (302) defines \( B \) only as \( n - 1 \) dimensional subspace \( \rightarrow \) need further constraints. Broyden (1965): use updating algorithm

\[
B_{k+1} = B_k + \frac{\Delta x_k^T \otimes (\Delta y_k - \Delta x_k B_k)}{|\Delta x_k|^2}
\]

with dyadic product of two vectors (columns \( \times \) rows) yielding matrix elements: \( (u^T \otimes v)_{ij} = u_i v_j \)

Advantage: Broyden’s formula (303) can be inverted analytically by help of
Sherman-Morrison-Woodbury lemma

\[(A + u^T \otimes v)^{-1} = A^{-1} - \frac{A^{-1} u^T \otimes v A^{-1}}{1 + v A^{-1} u^T}\]  

(304)

with row-vectors \(u, v\) and an invertible matrix \(A\) the required \(B_{k+1}^{-1}\) can be directly obtained from previous \(B_k^{-1}\):

\[B_{k+1}^{-1} = B_k^{-1} + \frac{(B_k^{-1} \Delta x_k^T) \otimes (\Delta x_k - \Delta y_k B_k^{-1})}{(\Delta y_k B_k^{-1}) \cdot \Delta x_k^T}\]  

(305)

→ no operations between full matrices involved  → only \(\sim N^2\) multiplications
Quasi Newton methods: Broyden method III

Broyden method

1. select starting point $x_0$ (e.g., initial guess on $n$ from LTE population numbers) and starting matrix $B_0^{-1} = f'^{-1}$ (Newton step)

2. $x_{k+1} = x_k - f(x_k)B_k^{-1}$

3. stop if $|\Delta x| < \epsilon$

4. else update Broyden matrix Eq. (305)

5. $k = k + 1$ goto 2
Interpolation
Interpolating data I

Consider following measurement of a cross section

<table>
<thead>
<tr>
<th>$E_i$ [MeV]</th>
<th>0</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>125</th>
<th>150</th>
<th>175</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma(E_i)$ [Mb]</td>
<td>10.6</td>
<td>16.0</td>
<td>45.0</td>
<td>83.5</td>
<td>52.8</td>
<td>19.9</td>
<td>10.8</td>
<td>8.25</td>
<td>4.7</td>
</tr>
</tbody>
</table>

The cross section can be described by Breit-Wigner formula

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4}$$  (306)
Interpolating data II

Interpolation problem

Task: Determine $\sigma(E)$ for values of $E$ which lie between measured values of $E$

By, e.g.,

- numerical interpolation (assumption of data representation by polynomial in $E$):
  - piecewise constant $\rightarrow$ step function (easy to implement, error goes as $\sim y'_i(x_{i+1} - x_i)$)
  - piecewise linear (special case of polynomial)
  - polynomial (Lagrange)
  - piecewise Lagrange, cubic spline
  $\rightarrow$ ignores errors in measurement (noise)
- fitting parameters of an underlying model, e.g., Breit-Wigner with $f_r$, $E_r$, $\Gamma$, (taking errors into account), i.e., minimizing $\chi^2$
- Fourier analysis
Interpolating data III

Linear interpolation

Tabulated function $y_i = y(x_i)$, $i = 1 \ldots N$, e.g., for interval $x_i, x_{i+1}$, linear interpolation in this interval is by

$$y = A(x)y_i + B(x)y_{i+1}$$  \hspace{1cm} (307)

where

$$A \equiv \frac{x_{i+1} - x}{x_{i+1} - x_i} \hspace{1cm} B \equiv 1 - A = \frac{x - x_i}{x_{i+1} - x_i}$$  \hspace{1cm} (308)

or:

$$y = y_i + (y_{i+1} - y_i) \frac{x - x_i}{x_{i+1} - x_i}$$

Disadvantages:

- not differentiable at nodes $x_i$
- error $\sim y_i''(x_{i+1} - x_i)^2$
Cosine interpolation

A smoother transition between intervals can be achieved by piecewise cosine functions:

\[
t = \frac{x - x_i}{x_{i+1} - x_i} \quad \text{(mapping on unit interval } [0, 1]) \tag{309}
\]

\[
B = \frac{(y_{i+1} + y_i)}{2} \; ; \quad A = y_i - B \tag{310}
\]

\[
y = A \cos(\pi t) + B \tag{311}
\]

Note, that \( y_i' = 0 \)
Lagrange interpolation (global)

- fit \((n - 1)\) polynomial through \(n\) data points

\[
p(x) = y_1\lambda_1(x) + y_2\lambda_2(x) + \ldots + y_n\lambda_n(x) \tag{312}
\]

\[
\lambda_i(x) = \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_i - x_1} \frac{x - x_2}{x_i - x_2} \ldots \frac{x - x_n}{x_i - x_n} \tag{313}
\]

where \(\sum_{i=1}^{n} \lambda_i(x) = 1\)

- practical: the \(\lambda_i\) are independent from the values of the function values \(f_i\) → for same nodes \(x_i\) → same \(\lambda_i\)s

- so, for \(n = 9\) → \((n - 1) = 8\)th degree polynomial

- note that \(\lambda_i(x_j) = \delta_{ij}\)
Interpolating data VI

Example: Lagrange interpolation polynomial \( n = 3 \)

\( n = 3 \) datapoints \( \rightarrow n - 1 = 2 \) degree polynomial, e.g., for points \( P_1 = (-1; 4), P_2 = (0; 1), P_3 = (2; 5) \)

\[
\lambda_1 = \frac{x - x_2}{x_1 - x_2} \cdot \frac{x - x_3}{x_1 - x_3} = \frac{(x - 0)}{(-1 - 0)} \cdot \frac{(x - 2)}{(-1 - 2)} = \frac{x^2 - 2x}{3} \quad (314)
\]

\[
\lambda_2 = \frac{x - x_1}{x_2 - x_1} \cdot \frac{x - x_3}{x_2 - x_3} = \frac{(x - (-1))}{(0 - (-1))} \cdot \frac{(x - 2)}{(0 - 2)} = \frac{x^2 - 2 - x}{-2} \quad (315)
\]

\[
\lambda_3 = \frac{x - x_1}{x_3 - x_1} \cdot \frac{x - x_2}{x_3 - x_2} = \frac{(x - (-1))}{(-2 - (-1))} \cdot \frac{(x - 0)}{(2 - 0)} = \frac{x^2 + x}{6} \quad (316)
\]

\[
p(x) = y_1 \cdot \lambda_1 + y_2 \cdot \lambda_2 + y_3 \cdot \lambda_3 = 4 \cdot \frac{x^2 - 2x}{3} + 1 \cdot \frac{x^2 - 2 - x}{-2} + 5 \cdot \frac{x^2 + x}{6} \quad (317)
\]

\[
= \frac{5}{3}x^2 - \frac{4}{3}x + 1 \quad (318)
\]

Check: \( \lambda_1 + \lambda_2 + \lambda_3 = \frac{x^2 - 2x}{3} + \frac{x^2 - 2 - x}{-2} + \frac{x^2 + x}{6} = 1 \)
Interpolating data VII

Application: Newton-Cotes formula for integration

Idea: interpolate $f(x)$ in $\int_a^b f(x)\,dx$ with polynomial of degree $n$ and integrate this polynomial exactly (note: now $n =$degree, start with $j = 0$):

\[ \int_a^b f(x)\,dx \approx \int_a^b p_n(x)\,dx = \int_a^b \sum_{i=0}^{n} f(x_i) \cdot \lambda_i(x) \quad (319) \]

\[ \lambda_i(x) = \prod_{\substack{j = 0 \\to i \\neq j}}^{n} \frac{x - x_j}{x_i - x_j} \quad x = a + ht \xrightarrow{\text{to}} \quad \phi_i(t) \prod_{\substack{j = 0 \\to i \\neq j}}^{n} \frac{t - j}{i - j} \quad (320) \]

Note that the transformation $x = a + ht$ means that $x_0 = a + h \cdot 0$, $x_1 = a + h \cdot 1$, ... Therefore the integration of $p_n(x)$ yields

\[ \int_a^b \sum_{i=0}^{n} f(x_i) \cdot \lambda_i(x) = h \sum_{i=0}^{n} f_i \int_0^{n} \phi_i(t)\,dt = h \sum_{i=0}^{n} f_i w_i \quad (321) \]
Interpolating data VIII

Example: Newton-Cotes formula \( n = 1 \)

\[
\begin{align*}
    w_0 &= \int_0^1 \phi_0(t) \, dt = \int_0^1 \frac{t - 1}{0 - 1} \, dt = \int_0^1 (1 - t) \, dt = \frac{1}{2} \quad (322) \\
    w_1 &= \int_0^1 \phi_1(t) \, dt = \int_0^1 \frac{t - 0}{1 - 0} \, dt = \int_0^1 t \, dt = \frac{1}{2} \quad (323)
\end{align*}
\]

\[
\int_a^b p_1(x) \, dx = h \sum_{i=0}^1 f_i w_i = h \left( f_0 \frac{1}{2} + f_1 \frac{1}{2} \right) = \frac{h}{2} (f_0 + f_1) \quad (324)
\]

→ trapezoid rule

Analogously for \( n = 2 \), e.g.,

\[
\begin{align*}
    w_0 &= \int_0^2 \frac{t - 1}{0 - 1} \cdot \frac{t - 2}{0 - 2} \, dt = \frac{1}{2} \int_0^2 (t^2 - 3t + 2) \, dt = \frac{1}{3} \quad (325) \\
    w_1 &= \frac{4}{3}, \quad w_2 = \frac{1}{3} \rightarrow \int_a^b p_2(x) \, dx = \frac{h}{3} (f_0 + 4f_1 + f_2) \rightarrow \text{Simpson's rule}
\end{align*}
\]
Interpolating data IX

→ closed Newton-Cotes formulae with nodes $t_i$ on $[0, 1]$:
$t_0 = 0, t_i = \frac{i}{n}, t_n = 1$, use mapping $x_i = a + t_i(b − a)$, so

$$\int_a^b f(x)dx = \int_a^b p_n(x)dx + E_f = (b − a) \sum_{i=0}^{n} w_i f(x_i) + E_f$$

(326)

<table>
<thead>
<tr>
<th>$n$</th>
<th>name</th>
<th>nodes $t_i$</th>
<th>weights $w_i$</th>
<th>$E_f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>trapezoid rule</td>
<td>0 1</td>
<td>1/2 1/2</td>
<td>$-\frac{(b-a)^3}{12} f''$</td>
</tr>
<tr>
<td>2</td>
<td>Simpson’s rule</td>
<td>0 1/2 1</td>
<td>1/6 4/6 1/6</td>
<td>$-\frac{(b-a)^5}{2880} f(4)$</td>
</tr>
<tr>
<td>3</td>
<td>3/8 rule</td>
<td>0 1/3 2/3 1</td>
<td>1/8 3/8 3/8 1/8</td>
<td>$-\frac{(b-a)^5}{6480} f(4)$</td>
</tr>
<tr>
<td>4</td>
<td>Milne rule</td>
<td>0 1/4 2/4 3/4 1</td>
<td>7/90 32/90 12/90 32/90 7/90</td>
<td>$-\frac{(b-a)^7}{1935360} f(6)$</td>
</tr>
</tbody>
</table>

for $n \geq 8$ some weights $w_i$ are also negative → subtractive cancellation

Note, again: $\sum w_i = 1$. The error: $E_f = h^{p+1} \cdot K \cdot f^{(p)}(\xi)$, $\xi \in (a, b)$
Trick: Neville’s algorithm (sometimes confused with Aitken’s method)

Instead of computing the whole Lagrange polynomial: nested linear interpolations

\[
\begin{align*}
  & f_1 \\
  & f_{12} \\
  & f_{123} \\
  & f_{23} \\
  & f_3
\end{align*}
\]

Where the \( f_{i...j} \) are recursively computed, e.g.,

\[
\begin{align*}
  f_{i...j} &= \frac{x - x_j}{x_i - x_j} f_{i...j-1} + \frac{x - x_i}{x_j - x_i} f_{i+1...j} \\
  f_{123} &= \frac{x - x_3}{x_1 - x_3} f_{12} + \frac{x - x_1}{x_3 - x_1} f_{23}
\end{align*}
\]

\( n - 1 \) degree

sequence of \ldots linear interpolations = interpolation with polynomial of

error can be estimated from

\[
\left| f_{i...j} - f_{i...j-1} \right| + \left| f_{i...j} - f_{i+1...j} \right|
\]

e.g.,

\[
\frac{\left| f_{12345} - f_{1234} \right| + \left| f_{12345} - f_{2345} \right|}{2}
\]
Neville’s algorithm: code

```c
// input : xi[], fi[], x; output: f(x), df
for (i = 1 ; i <= n ; ++i) ft[i] = fi[i] ;
for (i = 1 ; i <= n-1 ; ++i) {
    for (j = 1 ; j <= n-i ; ++j) {
        x1 = xi[j] ; x2 = xi[j+1] ;
        f1 = ft[j] ; f2 = ft[j+1] ;
        ft[j] = (x - x1)/(x2 - x1) * f2 + (x - x2)/(x1 - x2) * f1
    }
}
f = ft[1] ;
df = fabs(f-f1)+fabs(f-f2))/2. ;
```
Runge’s phenomenon: polynomials → ±∞ for x → ±∞. If function has different behavior (e.g., asymptotically constant) ⇒ oscillations at interval limits (e.g., for Runge’s function $\frac{1}{1+x^2}$)
Interpolating data XIII

one possible solution for the problem of Runge’s phenomenon: piecewise polynomials
here: 2nd degree polynomials (parabola, requires 3 points)

Problem: not differentiable at $x_i$
better: **Cubic Hermite spline**

- **Remember:** piecewise linear interpolation with functions

\[
A(x) = \frac{x_{i+1} - x}{x_{i+1} - x_i} \quad B(x) = 1 - A = \frac{x - x_i}{x_{i+1} - x_i} \quad (329)
\]

\[
\rightarrow y(x) = Ay_i + By_{i+1} \quad (330)
\]

- 2nd derivative=0 in interval and undefined/infinite at interval points

- Idea: get interpolation with smooth 1st derivative and continuous in 2nd derivative

A flat spline (lath) with fixed points (ducks) has minimum energy of bending

Burmester stencils are splines of 3rd degree
if (assume!): not only \( y_i \) given, but also \( y_i'' \) → add cubic polynomial with 2nd derivative varying linearly between \( y_i'' \) to \( y_{i+1}'' \) and zero values for \( x_i \) and \( x_{i+1} \) (so \( y_i, y_{i+1} \) unchanged):

\[
y(x) = Ay_i + By_{i+1} + Cy_i'' + Dy_{i+1}''
\] (331)

\[
C \equiv \frac{1}{6}(A^3 - A)(x_{i+1} - x_i)^2 \quad D \equiv \frac{1}{6}(B^3 - B)(x_{i+1} - x_i)^2
\] (332)

→ \( x \) dependence only through \( A, B \) → cubic \( x \)-dependence in \( C, D \)
check: now $y_i''$ is 2nd derivative of interpolating polynomial (calculating $dA/dx$, ...):

$$\frac{dy}{dx} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{3A^2 - 1}{6}(x_{i+1} - x_i)y_i'' + \frac{3B^2 - 1}{6}(x_{i+1} - x_i)y_{i+1}''$$

\hspace{1cm} (333)

$$\frac{d^2y}{dx^2} = Ay_i'' + By_{i+1}''$$

\hspace{1cm} (334)

note that $A = 1$ and $B = 0$ at $x_i$; and $A = 0$ and $B = 1$ at $x_{i+1}$, so ok (√)

however: in most cases $y_i''$ not known
idea → 1st derivative shall be continuous across interval boundaries → gives equation for 2nd derivatives
so: Eq. (333) shall be same for \( x_i \) on \([x_{i-1}, x_i]\) and on \([x_i, x_{i+1}]\) (for \( i = 2, \ldots, N - 1 \)) yielding \( N - 2 \) equations

\[
\frac{x_i - x_{i-1}}{6} y''_{i-1} + \frac{x_{i+1} - x_{i-1}}{3} y''_i + \frac{x_{i+1} - x_i}{6} y''_{i+1} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}}
\]

(335)

with \( N \) unknown \( y''_i \) \( \rightarrow \) need further constraint

- often: \( y''_1 \) and \( y''_N \) set to 0 \( \rightarrow \) natural cubic spline

- advantage of cubic splines: linear set of equations and also tridiagonal, each \( y''_i \) couples only to nearest neighbors
hence with mapping \( t = (x - x_i)/(x_{i+1} - x_i) \) on unit interval \([0, 1]\)

\[
p(t) = T M_h C = (t^3 \ t^2 \ t) \begin{pmatrix}
2 & -2 & 1 & 1 \\
-3 & 3 & -2 & -1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix} \begin{pmatrix}
y_i \\
y_{i+1} \\
m_i \\
m_{i+1}
\end{pmatrix}
\]

(336)

\[
p(t) = (2t^3 - 3t^2 + 1)y_i + (-2t^3 + 3t^2)y_{i+1} \\
+ (t^3 - 2t^2 + t)m_i + (t^3 - t^2)m_{i+1}
\]

(337)

with the numerical 1st derivatives \( m_i, m_{i+1} \)
Interpolating data XX

Catmull-Rom splines

The “width” of the curve segment can be controlled by a parameter $T_k$ according to (for $k = 2, \ldots, n - 2$):

$$m_k = T_k \frac{y_{k+1} - y_{k-1}}{x_{k+1} - x_{k-1}}$$

(338)
Simplest method on a rectilinear 2D grid: bilinear interpolation, i.e., linear interpolation in one direction, then again in another direction → as for Neville’s algorithm $2 \times \text{linear} = \text{quadratic order}$

If four $f$ values are given as follows: $f_1 : Q_{11} = (x_1, y_1)$, $f_2 : Q_{12} = (x_1, y_2)$, $f_3 : Q_{21} = (x_2, y_1)$, $f_4 : Q_{22} = (x_2, y_2)$ then

1. linear interpolation in $x$-direction:

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$ (339)

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$ (340)
2. linear interpolation in y-direction:

\[
f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)
\]

\[
= \frac{y_2 - y}{y_2 - y_1} \left( \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left( \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right)
\]

\[
= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \left( f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1) \right)
\]

→ same result as for 1. y-direction + 2. x direction
So:

\[ f(x, y) = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \cdot \left( f_1(x_2 - x)(y_2 - y) + f_3(x - x_1)(y_2 - y) + f_2(x_2 - x)(y - y_1) + f_4(x - x_1)(y - y_1) \right) \] (342)

Example, here: rgb colors on corner points \( f_{11} = b, f_{12} = f_{21} = r, f_{22} = g \)
Interpolating in 2D IV

As the interpolation can also be written as:

\[
    f(x, y) = \sum_{i=0}^{1} \sum_{j=0}^{1} a_{ij} x^i y^j = a_{00} + a_{10}x + a_{01}y + a_{11}xy 
\]  

(343)

\[
    a_{00} = f(0,0), 
\]

(344)

\[
    a_{10} = f(1,0) - f(0,0), 
\]

(345)

\[
    a_{01} = f(0,1) - f(0,0), 
\]

(346)

\[
    a_{11} = f(1,1) + f(0,0) - (f(1,0) + f(0,1)). 
\]

(347)

→ interpolation only linear along lines of const. \(x\) or const. \(y\), any other direction: quadratic in position (but linear in \(f\))

→ other method: bicubic interpolation \(f(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j\) with 16 coefficients

→ extension to 3D: trilinear interpolation, tricubic interpolation (64 coefficients)
Numerical Integration and Differentiation
Computing integrals

Often integrals have to be evaluated numerically. Examples:

- measured $dN(t)/dt$, the rate of some events, e.g., photons per unit time interval. Task: Determine the number of photons in the first second:

$$N(1) = \int_0^1 \frac{dN(t)}{dt} dt$$  \hspace{1cm} (348)

- radiative rates in the statistical equations for non-LTE population numbers (stellar atmospheres, photoionized nebulae)

$$R_{\ell u} = \int \frac{4\pi}{h\nu} \sigma_{\ell u}(\nu) J_\nu d\nu \quad \text{where} \quad J_\nu = \frac{1}{2} \int_{-1}^1 l_\nu d(\cos \theta)$$  \hspace{1cm} (349)
Also, *analytical* integration sometimes difficult or impossible (e.g., elliptic integrals), but numerically straightforward. So, Riemann definition

\[ \int_{a}^{b} f(x) \, dx = \lim_{h \to 0} \left[ h \sum_{i=1}^{(b-a)/h} f(x_i) \right] \] (350)

summing up areas of boxes of height \( f(x) \) and width \( h \to \) numerical quadrature

\[ \int_{a}^{b} f(x) \, dx \approx \sum_{i=1}^{N} f(x_i) w_i \] (351)

→ problem: find appropriate sampling \( f_i \equiv f(x_i) \),
generally: result improves with \( N \)
some hints

- remove singularities before integration
- sometimes splitting of interval is helpful, e.g.,

\[
\int_{-1}^{1} f(|x|) \, dx = \int_{-1}^{0} f(-x) \, dx + \int_{0}^{1} f(x) \, dx \tag{352}
\]

- or transformation/substitution

\[
\int_{0}^{1} x^{1/3} \, dx = \int_{y(0)}^{y(1)} y \, 3y^2 \, dy \quad \left( y(x) = x^{1/3} \right) \tag{353}
\]
The Trapezoid rule

- uses values $f(x)$ at evenly spaced $x_i \ (i = 1, \ldots, N)$ with step size $h$ on integration region $[a, b]$, including endpoints
- so, $N - 1$ intervals of length $h$:

$$h = \frac{b - a}{N - 1} \quad x_i = a + (i - 1)h$$ (354)

- so construct trapezoid on interval $i$ of width $h \rightarrow f(x)$ approximated by straight line
with average height \((f_i + f_{i+1})/2\):

\[
\int_{x_i}^{x_i+h} f(x) \, dx \simeq \frac{h(f_i + f_{i+1})}{2} = \frac{1}{2} hf_i + \frac{1}{2} hf_{i+1}
\]  

(355)

i.e. Eq. (351) for \(N = 2\) and \(w_i = \frac{1}{2} h\)

- hence for full interval

\[
\int_a^b f(x) \, dx \simeq \frac{h}{2} f_1 + hf_2 + hf_3 + \ldots + hf_{N-1} + \frac{h}{2} f_N
\]  

(356)

i.e. \(w_i = \{h/2, h, \ldots, h, h/2\}\)
Simpson’s rule I

- similar to Trapezoid rule with \textit{odd} number of points \( N \)
- for each interval: \( f(x) \) approximated by parabola

\[
f(x) = \alpha x^2 + \beta x + \gamma \quad (357)
\]

hence area for each interval:

\[
\int_{x_i}^{x_i + h} \left( \alpha x^2 + \beta x + \gamma \right) dx
\]

\[
\rightarrow \text{like integrating the corresponding Taylor series up to quadratic term (358)}
\]
need to determine $\alpha, \beta, \gamma$ for $f(x)$, so consider interval $[-1, 1]$

$$
\int_{-1}^{1} (\alpha x^2 + \beta x + \gamma) \, dx = \frac{1}{3} \alpha x^3 + \frac{1}{2} \beta x^2 + \gamma x \bigg|_{-1}^{1} = \frac{2\alpha}{3} + 2\gamma \quad (359)
$$

and $f(-1) = \alpha - \beta + \gamma$, $f(0) = \gamma$, $f(1) = \alpha + \beta + \gamma$, therefore:

$$
\Rightarrow \alpha = \frac{f(1) + f(-1)}{2} - f(0), \quad \beta = \frac{f(1) - f(-1)}{2}, \quad \gamma = f(0) \quad (360)
$$

so insert Eqn. (360) into Eq. (359)

$$
\int_{-1}^{1} (\alpha x^2 + \beta x + \gamma) \, dx = \frac{f(-1)}{3} + \frac{4f(0)}{3} + \frac{f(1)}{3} \quad (361)
$$
Simpson’s rule III

- or more general:

\[
\int_{x_i-h}^{x_i+h} f(x) \, dx = \int_{x_i-h}^{x_i} f(x) \, dx + \int_{x_i}^{x_i+h} f(x) \, dx \tag{362}
\]

\[
\simeq \frac{h}{3} f_{i-1} + \frac{4h}{3} f_i + \frac{h}{3} f_{i+1} \tag{363}
\]

→ pairs of intervals (so: odd \( N \))

- so for total interval \([a, b]\)

\[
\int_{a}^{b} f(x) \, dx \simeq \frac{h}{3} f_1 + \frac{4h}{3} f_2 + \frac{2h}{3} f_3 + \frac{4h}{3} f_4 + \ldots \frac{2h}{3} f_{N-2} + \frac{4h}{3} f_{N-1} + \frac{h}{3} f_N \tag{364}
\]

with \( w_i = \left\{ \frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \ldots, \frac{4h}{3}, \frac{h}{3} \right\} \) → check: \( \sum_{i=1}^{N} w_i \overset{!}{=} (N - 1)h \)
numerical integration: use algorithm with least number of integration points for accurate answer

estimate error from Taylor expansion at midpoint of each subinterval, e.g., for trapezoid rule \( hf^{(2)} \frac{h^2}{12} \), \( \times \) number of subintervals \( N = \frac{b - a}{h} \):

\[
E_{\text{trap}} = O \left( \frac{[b - a]^3}{12 N^2} \right) f^{(2)}, \quad E_{\text{Simps}} = O \left( \frac{[b - a]^5}{180 N^4} \right) f^{(4)} \quad (365)
\]

\[
\epsilon_{\text{trap}, \text{Simps}} = \frac{E_{\text{trap, Simps}}}{f} \quad (366)
\]

Note that for Simpson’s rule 3rd derivate cancels

\( \rightarrow \) Simpson’s rule should converge faster
check: find $N$ for minimum total error (usually for $\epsilon_{ro} \approx \epsilon_{appr}$):

$$
\epsilon_{tot} = \epsilon_{ro} + \epsilon_{approx} \approx \sqrt{N}\epsilon_m + \epsilon_{trap, \text{Simps}}
$$

(367)

$$
\rightarrow \quad \epsilon_{ro} = \epsilon_{trap, \text{Simps}} = \frac{E_{trap, \text{Simps}}}{f}
$$

(368)

Assuming some scale:

$$
\frac{f^{(n)}}{f} \approx 1 \quad b - a = 1 \quad \Rightarrow \quad h = \frac{1}{N}
$$

(369)
For double precision ($\epsilon_m \approx 10^{-15}$) and trapezoid rule:

\[
\sqrt{N}\epsilon_m \approx \frac{f^{(2)}(b-a)^3}{fN^2} = \frac{1}{N^2} \tag{370}
\]

\[\Rightarrow N \approx \frac{1}{(\epsilon_m)^{2/5}} = \left(\frac{1}{10^{-15}}\right)^{2/5} = 10^6 \tag{371}\]

\[\Rightarrow \epsilon_{ro} \approx \sqrt{N}\epsilon_m = 10^{-12} \tag{372}\]

For double precision ($\epsilon_m \approx 10^{-15}$) and Simpson’s rule:

\[
\sqrt{N}\epsilon_m \approx \frac{f^{(4)}(b-a)^5}{fN^4} = \frac{1}{N^4} \tag{373}
\]

\[\Rightarrow N \approx \frac{1}{(\epsilon_m)^{2/9}} = \left(\frac{1}{10^{-15}}\right)^{2/9} = 2154 \tag{374}\]

\[\Rightarrow \epsilon_{ro} \approx \sqrt{N}\epsilon_m = 5 \times 10^{-14} \tag{375}\]
We conclude:

- Simpson’s rule is better
- Simpson’s rule gives errors close to $\epsilon_m$ (in general for higher order integration algorithms, e.g., RK4)
- best numerical approximation not for $N \to \infty$, but small $N \leq 1000$
- however, as $\epsilon_{\text{Simps}} \sim f^{(4)}$ → only for sufficiently smooth functions, i.e., for narrow peak-like functions trapezoidal rule might be more efficient
Gaussian quadrature I

So far: improvement by aptly choosing weights $w_i$, but still equally space points $x_i$ for integral evaluation (cf. Eq. (351)), now: additional freedom of choosing $x_i$ so that order is twice that of Newton-Cotes formula for same number of nodes $N \rightarrow$ compute $N \times f(x_i)$. → choose $w_i$ and $x_i$ such that integral exact for orthogonal polynomials $\times$ specific weight function $W(x)$

$$
\int_a^b g(x)dx = \int_a^b W(x)f(x)dx \approx \int_a^b W(x)p_n(x)dx = \sum_{i=1}^{N} f(x_i) w_i \quad (376)
$$

Note that the integration of the orthogonal polynomials is on $[-1; +1]$, hence a transformation of the variables is usually necessary (e.g., for $W(x) \equiv 1$):

$$
\int_a^b f(x)dx \approx \frac{b-a}{2} \sum_{i=1}^{N} f \left( \frac{b-a}{2} x_i + \frac{a+b}{2} \right) w_i \quad (377)
$$
Example: Chebyshev-Gauß quadrature

The weight function is \( W(x) = \frac{1}{\sqrt{1-x^2}} \), i.e,

\[
\int_{-1}^{+1} g(x) dx = \int_{-1}^{+1} \frac{f(x)}{\sqrt{1-x^2}} dx \approx \sum_{i=1}^{N} w_i f(x_i) \quad (378)
\]

where \( w_i = \frac{\pi}{N}, \) and \( x_i = \cos \left( \frac{2i-1}{2N} \pi \right) \) are the zeros of associated Chebyshev polynomials of 1st kind \( T_n(x) \), with \( T_{n+1}(x) = 2xT_n(x) - T_{n-1} \), \( T_0(x) = 1, \ T_1(x) = x \) and

\[
\int_{-1}^{+1} T_n(x) W(x) T_m(x) dx = \delta_{nm} \quad (379)
\]

And for the Chebyshev polynomials of 2nd kind \( U_n(x) \) analogously:

\( W(x) = \sqrt{1-x^2}, \ w_i = \frac{\pi}{N+1} \sin^2 \left( \frac{i}{N+1} \pi \right), \ x_i = \cos \left( \frac{i}{N+1} \pi \right) \)
Most often: \( W(x) \equiv 1 \) \( \rightarrow \) Legendre-Gauß quadrature with Legendre Polynomials \( P_n(x) \), which are the solutions to Legendre’s differential equation (a special case of the Sturm-Liouville differential equation) \( \rightarrow \) Laplace equation in 3D for spherical coordinates \( \rightarrow \) QM

\[
\frac{d}{dx} \left[ (1 - x^2) \frac{dP_n(x)}{dx} \right] + n(n + 1)P_n(x) = 0 \quad (380)
\]

\[
\rightarrow P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n \quad \text{(Rodrigues’ formula)} \quad (381)
\]

so, \( P_0(x) = 1, \ P_1(x) = x, \ P_2(x) = \frac{1}{2}(3x^2 - 1), \ldots \) Then, weights

\[
w_i = \frac{2}{(1 - x_i^2)[P_n'(x)]^2} \quad (382)
\]
The $n$ zeros of $P_n(x)$ need to be computed, e.g., via Newton’s method. One can use as start approximation ($i = 1, \ldots, n$):

$$x_i \approx \cos \left( \frac{4i - 1}{4n + 2} \pi \right)$$ (383)

The values of $P_n(x)$ and $P'_n(x)$ for Newton’s method can be obtained via recursion:

$$nP_n(x) = (2n - 1)xP_{n-1}(x) - (n - 1)P_{n-2}(x)$$ (384)

$$\rightarrow P_n(x) = \left[ (2n - 1)xP_{n-1}(x) - (n - 1)P_{n-2}(x) \right]/n$$ (385)

$$(x^2 - 1)P'_n(x) = nxP_n(x) - nP_{n-1}(x)$$ (386)

$$\rightarrow P'_n(x) = (nxP_n(x) - nP_{n-1}(x))/(x^2 - 1)$$ (387)
Finally, the transformation from $t \in [-1; +1] \rightarrow x \in [a; b]$ can be done via
the midpoint $\frac{a+b}{2}$

\begin{equation}
    x = t \frac{b-a}{2} + \frac{a+b}{2} \tag{388}
\end{equation}

\begin{equation}
    w_{i,x} = w_{i,t} \frac{b-a}{2} \tag{389}
\end{equation}
Alternatively, other mappings are possible, allowing for integration of improper integrals with the Gauß-Legendre quadrature.

<table>
<thead>
<tr>
<th>interval</th>
<th>midpoint</th>
<th>$x_i(t_i)$</th>
<th>$w_{i,x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0, \infty]$</td>
<td>$a$</td>
<td>$a \frac{1 + t_i}{1 - t_i}$</td>
<td>$\frac{2a}{(1 - t_i)^2} w_{i,t}$</td>
</tr>
<tr>
<td>$[-\infty, +\infty]$</td>
<td>scale $a$</td>
<td>$a \frac{t_i}{1 - t_i}$</td>
<td>$\frac{a(1 + t_i^2)}{(1 - t_i)^2} w_{i,t}$</td>
</tr>
<tr>
<td>$[b, +\infty]$</td>
<td>$a + 2b$</td>
<td>$a + 2b + at_i$</td>
<td>$\frac{2(b + a)}{(1 - t_i)^2} w_{i,t}$</td>
</tr>
<tr>
<td>$[0, b]$</td>
<td>$ab/(b + a)$</td>
<td>$\frac{ab(1 + t_i)}{b + a - (b - a)t_i}$</td>
<td>$\frac{2ab^2}{(b + a - (b - a)t_i)^2} w_{i,t}$</td>
</tr>
</tbody>
</table>
Moreover, there exist even more orthogonal polynomials useful for Gauß quadrature

<table>
<thead>
<tr>
<th>interval</th>
<th>polynomials</th>
<th>$W(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[-1, 1]$</td>
<td>Legendre</td>
<td>1</td>
</tr>
<tr>
<td>$[-1, 1]$</td>
<td>Chebyshev 1st kind</td>
<td>$\frac{1}{\sqrt{1-x^2}}$</td>
</tr>
<tr>
<td>$[-1, 1]$</td>
<td>Chebyshev 2nd kind</td>
<td>$\sqrt{1-x^2}$</td>
</tr>
<tr>
<td>$(-1, 1)$</td>
<td>Jacobi</td>
<td>$(1-t)^\alpha(1+x)^\beta$, $\alpha, \beta &gt; -1$</td>
</tr>
<tr>
<td>$[0, +\infty)$</td>
<td>Laguerre</td>
<td>$e^{-x}$</td>
</tr>
<tr>
<td>$[0, +\infty)$</td>
<td>Generalized Laguerre</td>
<td>$x^\alpha e^{-x}$, $\alpha &gt; -1$</td>
</tr>
<tr>
<td>$(-\infty, +\infty)$</td>
<td>Hermite</td>
<td>$e^{-x^2}$</td>
</tr>
</tbody>
</table>
In general, the Gauß quadrature is constructed from orthogonal polynomials $p_n(x)$ with

$$\int_a^b p_n(x) W(x) p_n'(x) dx = \langle p_n | p_n' \rangle = N_n \delta_{nn'} \quad (390)$$

where $N_n$ is a normalization constant. If we choose the roots $x_i$ of $p_n(x) = 0$ and

$$w_i = -\frac{a_n N_n}{p_n'(x_i) p_{n+1}(x_i)} \quad (391)$$

with $i = 1, \ldots, n$, then the error in the quadrature is

$$\int_a^b g(x) dx - \sum_{i=1}^n f(x_i) w_i = \frac{N_n}{A_n^2(2n)!} f^{(2n)}(x_0) \quad (392)$$

where $x_0$ is some value in $[a, b]$, $A_n$ a coefficient of the $x^n$ term in the polynomial $p_n(x)$, $a_n = A_{n+1}/A_n$, e.g., for the Legendre polynomials $a_n = (2n + 1)/(n + 1)$ and $N_n = 2/(2n + 1)$. 
Gaussian quadrature IX

Gaussian quadrature is superior to simple methods with fixed integration steps

H. Todt (UP) Computational Astrophysics SoSe 2020 317 / 647
Romberg integration

Ideally: choose required accuracy $\epsilon \rightarrow$ know $n$ for Gauß quadrature (e.g., from Eq. (392)). Unfortunately, usually impossible. Therefore: increase $n$ until $\epsilon$ small enough, recalculate all $f(x_i)$ for new degree $n \rightarrow$ disadvantage of Gauß quadrature

Often better: trapezoid rule for different $N$ (or $h = (b-a)/N$) + extrapolation for $h \rightarrow 0$ (cf. Richardson extrapolation) $\rightarrow$ Romberg integration

\[
\int_0^1 e^{-x} dx
\]

Note that $h^2$ is plotted, although $h$ is used for the trapezoid rule $\rightarrow$ extrapolate polynomial in $h^2$
Sometimes numerical derivative needed, e.g., for minimization algorithms, Newton method for root finding, so

\[ f' = \frac{df(x)}{dx} := \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \tag{393} \]

Problem: for \( h \to 0 \to f(x + h) \approx f(x) \)

\( \to \) subtractive cancelation for numerator & machine precision limit for denominator
Numerical differentiation II

Forward difference
Taylor series with step size $h$

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \ldots$$  \hspace{1cm} (394)

$\rightarrow$ forward difference by solving Eq. (394) for $f'$

$$f_{fd}'(x) := \frac{f(x + h) - f(x)}{h} \simeq f'(x) + \frac{h}{2}f''(x) + \ldots$$  \hspace{1cm} (395)

approximate function by straight line through two points, error $\sim h$, e.g., consider $f(x) = a + bx^2$

$$f_{fd}'(x) \simeq \frac{f(x + h) - f(x)}{h} = 2bx + bh \phantom{.} \text{vs. exact } f' = 2bx$$  \hspace{1cm} (396)

$\rightarrow$ only good for small $h \ll 2x$
Numerical differentiation III

Central difference
modify Eq. (393) by stepping forward $h/2$ and backward $h/2$

$$f'_{cd} := \frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h}$$  \hspace{1cm} (397)$$

So, if we insert Taylor series for $f(x \pm \frac{h}{2})$ in to Eq. (397)

$$f'_{cd} \simeq f'(x) + \frac{1}{24}h^2f^{(3)}(x) + \ldots$$ \hspace{1cm} (398)$$

→ all terms with odd power of $h$ cancel → accuracy is of order $h^2$
if function well behaved, i.e., $f^{(3)}h^2/24 \ll f^{(2)}h/2$ → error for central difference method $\ll$ forward difference method, e.g., for $f(x) = a + bx^2$

$$f'_{cd}(x) \approx \frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h} = 2bx \quad \text{vs. exact} \quad f' = 2bx$$ \hspace{1cm} (399)$$
Numerical differentiation IV

Forward difference (solid line) and central difference (dashed) → central difference more accurate
Extrapolated difference
try to make also $h^2$ vanish by algebraic extrapolation

$$f_{ed}'(x) \approx \lim_{h \to 0} f_{cd}'$$  \hspace{1cm} (400)

→ need additional information for extrapolation by central difference with step size $h/2$:

$$f_{cd}'(x, h/2) = \frac{f(x + h/4) - f(x - h/4)}{h/2} \approx f'(x) + \frac{h^2 f^{(3)}(x)}{96} + \ldots$$  \hspace{1cm} (401)

We eliminate linear and quadratic error term by forming

$$f_{ed}'(x) := \frac{4 \frac{f(x+h/4) - f(x-h/4)}{h/2} - \frac{f(x+h/2) - f(x-h/2)}{h}}{3}$$

$$\approx f'(x) - \frac{h^4 f^{(5)}(x)}{4 \cdot 16 \cdot 120} + \ldots$$  \hspace{1cm} (402)
for $h = 0.4$ and $f^{(5)} \simeq 1 \rightarrow$ approximation error close to $\epsilon_m$. To minimize subtractive cancelation write Eq. (402) as

$$f'_{ed}(x) = \frac{1}{3h} \left( 8 \left[ f(x + \frac{h}{4}) - f(x - \frac{x}{4}) \right] - \left[ f(x + \frac{h}{2}) - f(x - \frac{h}{2}) \right] \right)$$

(404)
Error analysis

→ decreasing $h$ reduces approximation error but increases roundoff error (more steps $N$ needed), moreover: subtractive cancelation

\[ f' \approx \frac{f(x + h) - f(x)}{h} \approx \frac{\epsilon_m}{h} \approx \epsilon_{ro} \quad (405) \]

and

\[ \epsilon_{fd\text{ approx}} \approx \frac{f^{(2)} h}{2}, \quad \epsilon_{cd\text{ approx}} \approx \frac{f^{(3)} h^2}{24} \quad (406) \]

Therefore $\epsilon_{ro} \approx \epsilon_{approx}$ for
Numerical differentiation VIII

\[
\frac{\epsilon_m}{h} \approx \epsilon_{\text{approx}}^{\text{fd}} = \frac{f^{(2)}h}{2}, \quad \frac{\epsilon_m}{h} \approx \epsilon_{\text{approx}}^{\text{cd}} = \frac{f^{(3)}h}{24}
\]

\[
\Rightarrow h_{\text{fd}}^2 = \frac{2\epsilon_m}{f^{(2)}} \quad \Rightarrow h_{\text{cd}}^3 = \frac{24\epsilon_m}{f^{(3)}}
\]

(407)

(408)

for \( f' \approx f^{(2)} \approx f^{(3)} \) (e.g., \( \exp(x) \), \( \cos(x) \)) and double precision \( (\epsilon_m \approx 10^{-15}) \):

\[
h_{\text{fd}} \approx 4 \times 10^{-8} \quad \text{&} \quad h_{\text{cd}} \approx 3^{-5}
\]

\[
\Rightarrow \epsilon_{\text{fd}} \approx \frac{\epsilon_m}{h_{\text{cd}}} \approx 3 \times 10^{-8}, \quad \Rightarrow \epsilon_{\text{cd}} \approx \frac{\epsilon_m}{h_{\text{cd}}} \approx 3 \times 10^{-11}
\]

(409)

(410)

→ can choose \( 1000 \times \) larger \( h \) for central difference → error is \( 1000 \times \)
smaller for central difference
Second derivative
starting from first derivative with \textit{central difference} method

\[ f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h} \quad (411) \]

the 2nd derivative \( f^{(2)}(x) \) is central difference from 1st derivative

\[ f^{(2)}(x) \approx \frac{f'(x + h/2) - f'(x - h/2)}{h} , \quad (412) \]

\[ \approx \frac{[f(x + h) - f(x)] - [f(x) - f(x - h)]}{h^2} \quad (413) \]

\[ \approx \frac{f(x + h) + f(x - h) - 2f(x)}{h^2} \quad (414) \]

→ Eq. (413) better in terms of subtractive cancelation
Applications: The Lane-Emden equation
We remember:

**Example: Boundary values**

First two equations of stellar structure (e.g., for white dwarf)

\[
\frac{\partial r}{\partial m} = \frac{1}{4\pi r^2 \rho} \quad \text{mass continuity} \quad (415)
\]

\[
\frac{\partial P}{\partial m} = -\frac{G M}{4\pi r^4} \quad \text{hydrostatic equilibrium} \quad (416)
\]

+ equation of state \( P(\rho) \) (e.g., ideal gas \( P(\rho, T) = RT \rho/\mu \)), and boundary values

center \( m = 0 : r = 0 \) \hspace{1cm} (417)

surface \( m = M : \rho = 0 \rightarrow P = 0 \) \hspace{1cm} (418)

→ solve for \( r(m) \), specifically for \( R_* = r(m = M_*) \)
Derivation
(see also Hansen & Kawaler 1994)

→ if equation of state (EOS) for pressure is only function of density, e.g.,
completely degenerate, nonrelativistic, electron gas

\[ P_e = 1.004 \times 10^{13} \left( \frac{\rho[g \text{ cm}^{-3}]}{\mu_e} \right)^{5/3} \text{ dyn cm}^{-2} \]  
(419)

so, \( P \propto \left( \frac{\rho}{\mu_e} \right)^{5/3} \) power law ...

\( \mu_e = \left[ \sum Z_i X_i y_i / A_i \right]^{-1} \) mean molecular weight per electron, e.g.,
\( \mu_e \approx \left( 1 \cdot 0.7 \cdot 1 + 4 \cdot 0.3 \cdot 1 \right) \approx 1.2 \) for fully ionized H-He plasma

Polytropes are pseudo-stellar models where a power law for \( P(\rho) \) is assumed a priori without reference to heat transfer/thermal balance

→ only hydrostatic and mass continuity equation taken into account
define a polytrope as

\[ P(r) = K \rho^{1+1/n}(r) \]  

(420)

with some constant \( K \) and the polytropic index \( n \).

A polytrope must be in hydrostatic equilibrium, so hydrostatic equation

\[
\frac{dP}{dr} = -\frac{GM_r}{r^2} \rho \frac{\rho}{r^2} \cdot r^2 \frac{d}{dr} \]  

(421)

\[
\frac{d}{dr} \left( \frac{r^2 dP}{\rho dr} \right) = -G \frac{dM_r}{dr} = -4\pi G r^2 \rho 
\]  

(422)

with the continuity equation \( \frac{dM_r}{dr} = 4\pi r^2 \rho \) and the mass variable \( M_r = \int_0^r dm(r) \), i.e., \( M_r = 0 \rightarrow \) center \( (r = 0, \rho = \rho_c) \) and \( M_r = M_* \rightarrow \) surface \( (r = R_*, \rho = 0) \)

so
The Lane-Emden-Equation IV

\[ \frac{1}{r^2} \frac{d}{dr} \left( \frac{r^2 dP}{\rho \frac{d\rho}{dr}} \right) = -4\pi G \rho \]  

→ Poisson’s equation of gravitation with \( g(r) = \frac{d\Phi}{dr} = \frac{GM_r}{r^2} \), and
\[ \frac{dP}{dr} = -\frac{GM_r}{r^2} \rho \rightarrow \nabla^2 \Phi = 4\pi G \rho \] in spherical coordinates

find transformations to make Eq. (423) dimensionless. Define dimensionless variable \( \theta \)

\[ \rho(r) = \rho_c \theta^n(r) \]  

→ then, power law for pressure from Eq. (420)

\[ P(r) = K \rho^{1+1/n}(r) = K \rho_c^{1+1/n} \theta^{n+1}(r) = P_c \theta^{1+n}(r) \]  

→ \( P_c = K \rho_c^{1+1/n} \)
inserting Eqs. (424) & (426) into Eq. (423)

\[
\frac{(n+1)P_c}{4\pi G \rho_c^2} \frac{1}{r^2} \frac{d}{dr} \left( r^2 \frac{d\theta}{dr} \right) = -\theta^n \tag{427}
\]

together with dimensionless radial coordinate $\xi$

\[
r = r_n \xi \quad \text{with scale length} \quad r_n^2 = \frac{(n+1)P_c}{4\pi G \rho_c^2} \tag{428}
\]

our Poisson’s equation (423) becomes

→ so called
The Lane-Emden-Equation VI

Lane-Emden equation (Lane 1870; Emden 1907)

\[
\frac{1}{\xi^2} \frac{d}{d\xi} \left( \xi^2 \frac{d\theta}{d\xi} \right) = -\theta^n
\]  

(429)

with solutions “polytropes of index \( n \)” \( \theta_n(\xi) \)

applications:

- describe i.g. self-gravitating spheres (of plasma)
- Bonnor-Ebert sphere \( (n \to \infty, \text{so } u, e^{-u} \text{ instead of } \theta, \theta^n) \): stable, finite-sized, finite-mass isothermal cloud with \( P \neq 0 \) at outer boundary \( \rightarrow \) Bonnor-Ebert mass (Ebert 1955; Bonnor 1956)
- characterize (full) stellar structure models, e.g., Bestenlehner (2020)
- composite polytropic models for modeling of massive interstellar clouds with a hot ionized core, stellar systems with compact, massive object (BH) at centre
Remarks:

if EOS is ideal gas $P = \rho N_A k T / \mu$

$$P(r) = K' T^{n+1}(r), \quad T(r) = T_c \theta(r) \quad (430)$$

with $K' = \left( \frac{N_A k}{\mu} \right)^{n+1} K^{-n}, \quad T_c = K \rho_c^{1/n} \left( \frac{N_A k}{\mu} \right)^{-1} \quad (431)$

→ polytrope with EOS of ideal gas and mean molecular weight $\mu$ gives temperature profile, radial scale factor is

$$r_n^2 = \left( \frac{N_A k}{\mu} \right)^2 \frac{(n+1) T_c^2}{4\pi G \rho_c} = \frac{(n+1) K \rho_c^{1/n-1}}{4\pi G} \quad (432)$$
Requirements for physical solutions:

central density $\rho_c \rightarrow \theta(\xi = 0) = 1$

spherical symmetry at center $(dP/dr|_{r=0}) \rightarrow \theta' = d\theta/d\xi = 0$ at $\xi = 0$

$\rightarrow$ suppresses divergent solutions of the 2nd order system $\rightarrow$ regular solutions (E-solutions)

surface $P = \rho = 0 \rightarrow \theta_n = 0$ (first occurrence of that!) at $\xi_1$

Boundary conditions for polytropic model

$\theta(0) = 1, \quad \theta'(0) = 0 \quad$ at $\xi = 0$ (center)

$\theta(\xi_1) = 0 \quad$ at $\xi = \xi_1$ (surface)

So stellar radius

$$R = r_n \xi_1 = \sqrt{\frac{(n + 1)P_c}{4\pi G \rho_c^2}} \xi_1$$  \hspace{1cm} (433)

for given $K, n$, and either $\rho_c$ or $P_c$ ($P_c = K \rho_c^{\frac{1+1/n}{n}}$)
Analytic E-solutions

→ analytic regular solutions exist for $n = 0, 1, 5$

$n = 0$ constant density sphere, $\rho(r) = \rho_c$, and

$$\theta_0(\xi) = 1 - \frac{\xi^2}{6} \rightarrow \xi_1 = \sqrt{6}$$ (434)

so $P(\xi) = P_c \theta(\xi) = P_c \left[1 - \left(\frac{\xi}{\xi_1}\right)^2\right]$. For $P_c$ we need $M, R$ from Eq. (433): $P_c = (3/8\pi)(GM^2/R^4)$

$n = 1$ solution $\theta_1$ is sinc function

$$\theta_1 = \frac{\sin \xi}{\xi} \text{ with } \xi_1 = \pi$$ (435)

$\rightarrow \rho = \rho_c \theta$ and $P = P_c \theta^2$
The Lane-Emden-Equation X

\( n = 5 \) finite central density \( \rho_c \) but infinite radius \( \xi_1 \to \infty \): 

\[
\theta_5(\xi) = \frac{1}{\sqrt{1 + \frac{\xi^2}{3}}} 
\]  

(436)

contains finite mass

solutions with \( n > 5 \) have also infinite radius, but also infinite mass
For the interesting cases $0 \leq n \leq 5 \rightarrow$ numerical solution

\[
\frac{1}{\xi^2} \frac{d}{d\xi} \left( \xi^2 \frac{d\theta}{d\xi} \right) = \frac{2}{\xi} \frac{d\theta}{d\xi} + \frac{d}{d\xi} \frac{d\theta}{d\xi} = -\theta^n \quad (437)
\]

Reduction: set $x = \xi$, $y = \theta$, $z = \left( \frac{d\theta}{d\xi} \right) = \left( \frac{dy}{dx} \right)$

\[
y' = \frac{dy}{dx} = z, \quad (438)
\]
\[
z' = \frac{dz}{dx} = -y^n - \frac{2}{x}z \quad (439)
\]

Assume that we have values $y_i$, $z_i$ at a point $x_i$, so that we can get with some step size $h$: $y_{i+1}$ & $z_{i+1}$ at $x_{i+1} = x_i + h$
Then with RK4:

\[ k_1 = h \cdot y'(x_i, y_i, z_i) = h \cdot (z_i) \]  \hspace{1cm} (440)

\[ \ell_1 = h \cdot z'(x_i, y_i, z_i) = h \cdot (-y_i^n - \frac{2}{x_i} z_i) \]  \hspace{1cm} (441)

\[ k_2 = h \cdot y'(x_i + h/2, y_i + k_1/2, z_i + \ell_1/2) = h \cdot (z_i + \ell_1/2) \]  \hspace{1cm} (442)

\[ \ell_2 = h \cdot z'(x_i + h/2, y_i + k_1/2, z_i + \ell_1/2) = h \cdot \left( -(y_i + k_1/2)^n - \frac{2}{x_i + h/2} (z_i + \ell_1/2) \right) \]  \hspace{1cm} (443)

\[ k_3 = h \cdot y'(x_i + h/2, y_i + k_2/2, z_i + \ell_2/2) \]  \hspace{1cm} (444)

\[ \ell_3 = h \cdot z'(x_i + h/2, y_i + k_2/2, z_i + \ell_2/2) \]  \hspace{1cm} (445)

\[ k_4 = h \cdot y'(x_i + h, y_i + k_3, z_i + \ell_3) \]  \hspace{1cm} (446)

\[ \ell_4 = h \cdot z'(x_i + h, y_i + k_3, z_i + \ell_3) \]  \hspace{1cm} (447)
Integration could be started for $\xi = 0$, as

\[
\theta_n(\xi) = 1 - \frac{\xi^2}{6} + \frac{n}{120} \xi^4 - \frac{n(8n-5)}{15120} \xi^6 + \ldots
\]  

(450)

\[
\rightarrow \theta'_n(\xi) = -\frac{1}{3} + \frac{n}{40} \xi^3 - \frac{n(8n-5)}{2520} \xi^5 + \ldots
\]  

(451)

So for $\xi \to 0$ then $y' \to -1/3$

but there is a irregularity in $\xi = 0$ in $z' = -y^n - \frac{2}{x}z$ (Eq. (439))

better: choose $0 < \xi \ll 1$ and compute with help of Eq. (450)

$y, y'(= z), z'$
construct polytropes for \( n < 5 \) and given \( M, R \)
\[ \rightarrow \] possible as long as \( K \) not fixed
because of definition of \( \theta \) from \( \rho(r) = \rho_c \theta^n(r) \) (Eq. (424)) and \( r = r_n \xi \)
(Eq. 428)

\[ m(r) = \int_0^r 4\pi \rho r^2 dr = 4\pi \rho_c \int_0^r \theta^n r^2 dr = 4\pi \rho_c \frac{r^3}{\xi^3} \int_0^\xi \theta^n \xi^2 d\xi \quad (452) \]

Note that \( r^3/\xi^3 = r_n^3 \) is constant. From Lane-Emden equation (429)

\[ \frac{1}{\xi^2} \frac{d}{d\xi} \left( \xi^2 \frac{d\theta}{d\xi} \right) = -\theta^n \rightarrow \theta^n \xi^2 = -\frac{d}{d\xi} \left( \xi^2 \frac{d\theta}{d\xi} \right) \]

follows direct integration, so

\[ m(r) = 4\pi \rho_c r^3 \left( -\frac{1}{\xi} \frac{d\theta}{d\xi} \right) \quad (453) \]

\[ \rightarrow \] Eq. (453) contains \( \xi \) and \( r \), related by Eq. (428): \( r/\xi = r_n = R/\xi_1 \), so
for the surface:
Applying the Lane-Emden equation to stars II

\[ M = 4\pi \rho_c R^3 \left( -\frac{1}{\xi} \frac{d\theta}{d\xi} \right) \xi=\xi_1 \]  \hspace{1cm} (454)

With help of the mean density \( \bar{\rho} := M/(\frac{4}{3}\pi R^3) \) this can be written as

\[ \frac{\bar{\rho}}{\rho_c} = \left( -\frac{3}{\xi} \frac{d\theta}{d\xi} \right) \xi=\xi_1 \]  \hspace{1cm} (455)

Note the right hand side depends only on \( n \), can be computed. E.g., for \( n = 1 \rightarrow \left( -\frac{3}{\xi} \frac{d\theta}{d\xi} \right) \xi=\xi_1 = 1 \)

the larger \( n \rightarrow \) the smaller \( \frac{\bar{\rho}}{\rho_c} \rightarrow \) the higher the density concentration
Random numbers and Monte-Carlo methods
Motivation

Many physical process can be described in two pictures:

- microscopic, individual, e.g., particle-particle interactions are considered realization usually with help of → Monte-Carlo methods
- macroscopic, only the effective coaction is described → usually analytical equations

Example: Yukawa potential

microscopic: quarks-gluons and the strong interaction
effective theory: e.g. Yukawa potential

Monte-Carlo simulation

Computer algorithm based on a large number of repeated random experiments to obtain a representative sample of the possible configurations.
Motivation II

Example: radiative transfer

- **microscopic**: interaction of light (photons) with atoms/ions/molecules
  → MOCASSIN for Monte-Carlo simulation of photon propagation in gaseous nebulae
  → MCRH (Noebauer 2015) MC radiation hydrodynamics for stellar winds
  - **advantage**: arbitrary geometries (e.g., torus) and processes
  - **disadvantage**: feedback on matter (often iteratively calculated) hard to implement because of MC noise

- **macroscopic**: radiative transfer equation = effective theory, i.e. light (intensity) instead of single photons
  → Cloudy spectral synthesis code for astrophysical plasmas
  → PoWR for emergent spectra of stellar atmospheres
  - **advantage**: feedback on matter (non-LTE) via iteration (boundary conditions, e.g., conservation of energy)
  - **disadvantage**: hard to program (numerical stability, only some geometries, e.g., spherical symmetry)
For MC methods we need *good* and *many* random numbers. Usual base are uniformly distributed random numbers. Humans are not a good source for random numbers:

Figure: random numbers, created by colleagues

→ not uniformly distributed, too few

→ direct, severe consequence: don’t make up your own passwords!
Random numbers II

Other sources: rolling dices, tossing coins

most programming languages have a builtin random function, which gives pseudo-random numbers, e.g., in C/C++ integers (!) from $[0, \text{RAND\_MAX}]$:

```c
#include <cstdlib>
...
int i = rand();
```

- output of next random number of a sequence
- restart by `srand(i)`;

To get uniformly distributed random numbers $\in [0; 1]$:  

```c
r = rand()/double(RAND\_MAX);
```
Definition

A result (a state) is random, if it was not predictable.

Quality tests for random numbers:

- **uniform distribution**: random numbers should be fair
- **sequential tests**: for \( n \)tuple repetitions (usually only for \( n = 2 \) und \( n = 3 \))
- **run tests**: for monotonically increasing/decreasing sequences, and duration of stay in a certain interval
- and more . . .
Non-uniform distributions
random number generators give \textit{uniform} (pseudo) random numbers \( \in [0, \text{RAND\_MAX}] \rightarrow r \in [0, 1] \)

we need often different distributions, e.g., normal (Gaussian) distributions or uniform distributions \( x \in [a, b] \)

i.e., we need a transformation that maps \( r \) to \( x \), so

\begin{equation}
    \hat{x} = P^{-1}(r)
\end{equation}
First, for the case of discrete numbers

- e.g., two events (1,2) with probabilities $p_1$ and $p_2$, such that
  \[ p_1 + p_2 = 1 \]  

  How can we choose with help of $r$?

  - obvious choice: for $r < p_1$ event 1, otherwise 2
    
    \[ \begin{array}{cccccc}
    * & * & * & * & * & * \\
    p_1 & * & * & * & p_2 & * \\
    \end{array} \]

  - for the case of 3 possible events with $p_1$, $p_2$, $p_3$: $r < p_1 \rightarrow$ event 1, $r < p_1 + p_2 \rightarrow$ 2, else 3
    
    \[ \begin{array}{cccccc}
    * & * & * & * & * & * \\
    p_1 & * & * & * & p_2 & * \\
    \end{array} \]

  - in general for $n$ events, event $i$ is selected if for $r$:
    
    \[
    \sum_{j=0}^{i-1} p_j \leq r \leq \sum_{j=0}^{i} p_j \quad \text{where } p_0 \equiv 0
    \]
For continuous distributions:

- need the *probability density function* \( p(x) \), where \( p(x) \cdot dx \) is probability that \( x \) is in the interval \([x, x + dx]\)
- moreover, \( p(x) \) is normalized:

\[
\int_{-\infty}^{+\infty} dx \ p(x) = 1 \tag{459}
\]

**Example: uniform distribution**

\[
p_u(r) = \begin{cases} 
1, & \text{if } 0 \leq r \leq 1 \\
0, & \text{else}
\end{cases} \tag{460}
\]
for the continuous case (continuum limit $i \to x$) in the Eqn. (458)

\[
\sum_{j=0}^{i-1} p_j \leq r \leq \sum_{j=0}^{i} p_j \quad \text{where } p_0 \equiv 0
\]

both sums are equal and become the integral:

\[
P(x) = \int_{-\infty}^{x} p(x') \, dx' = r \quad (461)
\]
This corresponds to the cumulated distribution function

\[ P(x) = \int_{-\infty}^{x} p(x') \, dx' \]  \hspace{1cm} (462)

i.e. the probability to get a random number smaller or equal \( x \).
Geometrically: fraction of the area left of (smaller than) \( x \). We state:

\[ P(x) = r \]  \hspace{1cm} (463)
\[ \Rightarrow x = P^{-1}(r) \]  \hspace{1cm} (464)

i.e. exactly as \( r \) also \( P(x) \) is uniformly distributed.
Therefore, the probability to find \( P(x) \) in the interval \([P(x), P(x) + dP(x)]\)
is \( dP(x) = dr \) (Eq. 463).
The relation between $dP(x)$ and $dx$ is obtained by derivating Eq. (462)

\[ \frac{dP(x)}{dx} = p(x) \]  \hspace{1cm} (465)

for $0 \leq r \leq 1$ it is also:

\[ dP(x) = p(x) \, dx = p_u(r) \, dr \]  \hspace{1cm} (466)

I.e., because of Eq. (463) is $x$ distributed according to $p(x)$
To obtain such \( p(x) \) distributed random numbers, one has to solve Eq. (464)

**Inverse transformation**

1. Insert the required distribution \( p(x) \) into:

\[
    r = P(x) = \int_{-\infty}^{x} p(x') \, dx'
\]

2. Solve for \( x \), i.e. find

\[
    P^{-1}(x) = r
\]

Not for all \( p(x) \) are the corresponding conditions fulfilled (solvable integral and invertibility)
Example for inverse transformation

Let

\[ p(x) = \begin{cases} a e^{-ax}, & \text{if } 0 \leq x \leq \infty \\ 0, & x < 0 \end{cases} \]  

(469)

\[ P(x) = \int_0^x a e^{-ax'} dx' = 1 - e^{-ax} \]  

(470)

\[ \Rightarrow x = -a^{-1} \ln(1 - r) \]  

(471)

as \((1 - r)\) is exactly distributed as \(r\):

\[ x = P^{-1}(r) = -a^{-1} \ln r \]  

(472)

The evaluation of \(\ln\) on a computer is relatively time consuming

\(\rightarrow\) inverse transformation not always the best method
Non-uniform distributions II
non-uniform distribution:

- with help of the *inversion* method we can get non-uniform random numbers from uniform random numbers \( \rightarrow \) condition: \( P(x) \) invertable
- for the Gaussian normal distribution:

\[
p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \quad (473)
\]

\( P(x) \) is not analytical representable (error function)

- idea: 2d-transformation where:

\[
p(x, y) \, dx \, dy = \frac{1}{2\pi\sigma^2} e^{-\left(x^2+y^2\right)/2\sigma^2} \, dx \, dy \quad (474)
\]

- change to polar coordinates:

\[
r = \sqrt{x^2+y^2} \quad \theta = \tan^{-1}\frac{y}{x} \quad (475)
\]
Box-Muller method II

- let \( \rho = \frac{r^2}{2} \) and set \( \sigma = 1 \):

\[
p(x, y) dx \, dy = p(\rho, \theta) \, d\rho \, d\theta = \frac{1}{2\pi} e^{-\rho} \, d\rho \, d\theta
\]  

(476)

- now generate random numbers \( \rho \) according to exponential distribution, so \( \rho = -\ln u \) (\( u \) standard uniform distributed) and \( \theta \) uniform distributed on \([0, 2\pi]\), then

\[
x = \sqrt{-2 \ln u} \cos \theta \quad \text{und} \quad y = \sqrt{-2 \ln u} \sin \theta
\]  

(477)

are each according to Eq. (473) with \( \sigma = 1 \) and \( \mu = 0 \) distributed
Example: Neutron transport
Application for non-uniform random numbers!
Transport of neutrons through matter – one of the first MC applications!

- consider a plate of thickness $t$
- plate is infinite in $z$ and $y$ direction, $x$-axis perpendicular to the plate
- at each point within the plate: probability $p_c$, that neutron gets absorbed (captured) and probability $p_s$ that neutron is scattered
- after each scattering: find scattering angle $\theta$ in $xy$ plane
- as motion in $y, z$ direction irrelevant: azimutal angle $\phi$ irrelevant
Determine scattering angle & scattering path length

1. Isotropic scattering:

\[ p(\theta, \phi) \, d\theta \, d\phi = d\Omega / 4\pi \]  \hspace{1cm} (478)

because of \( d\Omega = \sin \theta \, d\theta \, d\phi \) :

\[ p(\theta, \phi) = \frac{\sin \theta}{4\pi} \]  \hspace{1cm} (480)

obtain \( p(\theta) \) and \( p(\phi) \) by integration over the complementary angle:

\[ p(\theta) = \int_0^{2\pi} p(\theta, \phi) \, d\phi = 2\pi \frac{\sin \theta}{4\pi} = \frac{1}{2} \sin \theta \]  \hspace{1cm} (481)

\[ p(\phi) = \int_0^{\pi} p(\theta, \phi) \, d\theta = \frac{1}{4\pi}(-\cos \pi + \cos 0) = \frac{1}{2\pi} \]  \hspace{1cm} (482)

i.e. \( p(\theta, \phi) = p(\theta)p(\phi) \rightarrow \) independent variables
If random variable $\phi$ is wanted ($p(\phi) \equiv \text{const.}$):

$$\phi = 2\pi r \quad (483)$$

To get random $\theta$ according to Eq. (481) $\rightarrow$ inversion method:

$$r = P(\theta) = \int_0^\theta \frac{1}{2} \sin x \, dx = -\frac{1}{2} (\cos \theta - \cos 0) \quad (484)$$

$$\cos \theta = 1 - 2r \quad (485)$$

I.e. $\cos \theta$ is uniformly distributed on $[-1; 1]$ and $\phi$ on $[0; 2\pi]$. Solve for $\theta$ possible, but unnecessary, as required for $x$ component of the path $\rightarrow$ 2. scattering path length:

$$x = \ell \cos \theta \quad (486)$$

where $\ell$ from $p(\ell) \sim e^{-\ell/\lambda}$ (see example for inversion method):

$$\ell = -\lambda \ln r \quad (487)$$

$\lambda \rightarrow$ mean free path (e.g., $\lambda = (\sigma n)^{-1}$)
Algorithm, start at $x = 0$:

1. determine, if neutron is scattered or captured. If captured: increment number of absorbed neutrons, go to 5 step

2. scattering: “dice” $\cos \theta$ and $\ell$, move to $x$ position by $\ell \cos \theta$

3. if $x < 0$: increment number of reflected neutrons, if $x > t$: increment number of transmitted neutrons; go to 5

4. repeat step 1 - 3 until final result is achieved for all neutrons

5. repeat step 1 - 4 with more incident neutrons
Monte-Carlo integration
Idea: Can the area of a pool (irregular!) be measured by throwing stones?

- pool with area $F_n$ in a field with area $A$
fraction of the \textit{randomly} thrown stones which fall into the pool:

\[ \frac{n_p}{n} = \frac{F_n}{A} \quad (488) \]

\( (n \ \text{stones, } n_p \ \text{hit pool}) \)

determine \( F_n \) with help of the \textit{hit-or-miss method}:

\[ F_n = A \frac{n_p}{n} \quad (489) \]
- rectangle of height $h$, width $(b - a)$, area $A = h \cdot (b - a)$, such that $f(x)$ within the rectangle
- generate $n$ pairs of random variables $x_i, y_i$ with $a \leq x_i \leq b$ and $0 \leq y_i \leq h$
- fraction $n_t$ of the points, which fulfill $y_i \leq f(x_i)$ gives estimate for area under $f(x)$ (integral)
Buffon’s needle problem – determine $\pi$ by throwing matches

Buffon’s question (1773): What is the probability that a needle or a match of length $\ell$ will lie across a line between two strips on a floor made of parallel strips, each of same width $t$?

$\rightarrow x$ is distance from center of the needle to closest line, $\theta$ angle between needle and lines, hence the uniform probability density functions are

$$p(x) = \begin{cases} \frac{2}{t} : & 0 \leq x \leq \frac{t}{2} \\ 0 : & \text{elsewhere} \end{cases} \quad p(\theta) = \begin{cases} \frac{2}{\pi} : & 0 \leq \theta \leq \frac{\pi}{2} \\ 0 : & \text{elsewhere} \end{cases}$$

$x, \theta$ independent $\rightarrow p(x, \theta) = \frac{4}{t\pi}$ with condition $x \leq \frac{\ell}{2} \sin \theta$. If $\ell \leq t$ (short needle):

$$P(\text{hit}) = \int_{\theta=0}^{\frac{\pi}{2}} \int_{x=0}^{\frac{\ell}{2} \sin \theta} \frac{4}{t\pi} dx d\theta = \frac{2\ell}{t\pi}$$

$\rightarrow$ count hits and misses and then:

$$\pi = \frac{2\ell}{t} \frac{n_{\text{hit}} + n_{\text{miss}}}{n_{\text{hit}}}$$
**Sample-mean method**

- the integral

$$F(x) = \int_a^b f(x) \, dx$$  \hspace{1cm} (490)

is given in the interval \([a, b]\) by the mean \(\langle f(x) \rangle\) (mean value theorem for integration)

- choose arbitrary \(x_i\) (instead of regular intervals) and calculate

$$F_n = (b - a)\langle f(x) \rangle = (b - a) \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$  \hspace{1cm} (491)

where \(x_i\) are uniform random numbers in \([a, b]\)

\[
\left(\text{cf. rectangle rule} \quad F_n = \sum_{i=1}^{n-1} f(x_i) \Delta x \quad \text{with fixed } x_i, \Delta x\right)  \hspace{1cm} (492)
\]
Error estimation
Numerical integration (exact or MC) gives approximation

\[ \int_{a}^{b} f(x) \, dx = Q(f) + E(f) \]  

(493)

Q(f) so-called quadrature formula,
E(f) error → unknown (obvious)

Aim: estimate magnitude of error

so far: error calculated from our knowledge of the exact result
Obvious: for constant integrand $f$ is $E = 0$, i.e. $F_n$ is independent of $n$ (and always the same)

Idea: try to estimate the error with help of the standard deviation $\sigma$:

$$\sigma^2 = \langle f(x)^2 \rangle - \langle f(x) \rangle^2$$ (494)

\[
\langle f(x) \rangle = \frac{1}{n} \sum_{i=1}^{n} f(x_i)
\] (495)

\[
\langle f(x)^2 \rangle = \frac{1}{n} \sum_{i=1}^{n} f(x_i)^2
\] (496)

if $f$ constant $\rightarrow \sigma = 0$
consider the example \( f(x) = 4\pi \sqrt{1 - x^2} \) with \( F = \int_0^1 f(x) \, dx = \pi \)

Calculate \( \sigma \) for different \( n \) (cf. Gould et al. 1996)

| \( F_n \)   | \( n \) | \( E = |F_n - \pi| \) | \( \sigma \) |
|------------|--------|----------------------------|---------|
| 3.271771   | \( 10^1 \) | 0.13017                     | 0.78091 |
| 3.100276   | \( 10^2 \) | 0.04131                     | 0.91441 |
| 3.173442   | \( 10^3 \) | 0.03185                     | 0.85013 |
| 3.135863   | \( 10^4 \) | 0.00572                     | 0.90317 |
| 3.142189   | \( 10^5 \) | 0.00059                     | 0.89051 |
| 3.141798   | \( 10^6 \) | 0.00020                     | 0.89236 |

\( \sigma \) almost constant and much larger than \( E \)

but: decrease of \( E \) from \( n = 10^2 \) to \( n = 10^4 \) by a factor of 10 
\( \rightarrow \sim 1/n^{1/2} \)

therefore: \( \sigma \) says how much \( f \) varies in \([a, b]\)
idea: estimate $E$ by several runs $\alpha$ for constant $n = 10^4$, each with result $M_\alpha$:

<table>
<thead>
<tr>
<th>$M_\alpha$</th>
<th>$\alpha$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14892</td>
<td>1</td>
<td>0.00735</td>
</tr>
<tr>
<td>3.13255</td>
<td>2</td>
<td>0.00904</td>
</tr>
<tr>
<td>3.14042</td>
<td>3</td>
<td>0.00117</td>
</tr>
<tr>
<td>3.14600</td>
<td>4</td>
<td>0.00441</td>
</tr>
<tr>
<td>3.15257</td>
<td>5</td>
<td>0.01098</td>
</tr>
<tr>
<td>3.13972</td>
<td>6</td>
<td>0.00187</td>
</tr>
<tr>
<td>3.13107</td>
<td>7</td>
<td>0.01052</td>
</tr>
<tr>
<td>3.13585</td>
<td>8</td>
<td>0.00574</td>
</tr>
<tr>
<td>3.13442</td>
<td>9</td>
<td>0.00717</td>
</tr>
<tr>
<td>3.14047</td>
<td>10</td>
<td>0.00112</td>
</tr>
</tbody>
</table>

$E$ varies, differences $|M_\alpha - M_\beta|_{\alpha \neq \beta}$ between results comparable with $E$, therefore:
define standard deviation $\sigma_m$ of the means:

$$\sigma_m^2 = \langle M^2 \rangle - \langle M \rangle^2 \quad (497)$$

$$\langle M \rangle = \frac{1}{m} \sum_{\alpha=1}^{m} M_\alpha \quad \rightarrow \quad \langle M^2 \rangle = \frac{1}{m} \sum_{\alpha=1}^{m} M^2_\alpha \quad (498)$$

$$\sum_{\alpha=1}^{m} \rightarrow \quad (499)$$

for the runs 1 till 10 one gets $\sigma_m = 0.006762 \rightarrow$ comparable with $E$

exact: one run has the chance of 68% that $M_\alpha$ is in in the range $\pi \pm \sigma_m$

however method not very usefull, as several runs are required
Numerical integration and error VI

- actually for large $n$ holds:

$$\sigma_m = \frac{\sigma}{\sqrt{n-1}} \approx \frac{\sigma}{\sqrt{n}}$$

(500)

- e.g., for $n = 10^4$ is $\sigma_m = 0.90317/100 \approx 0.009$, i.e., consistent with our estimate $\sigma_m = 0.007$ and the error $E = 0.006$

How can we get $\sigma$ without $\alpha$ runs?
Hence, split one run, e.g., in $s = 10$ subsets $k$ such that each contains $n/s = 1000$ trials and has result $S_k$

Then, with the mean $\langle S \rangle$ is also

$$\sigma_s^2 = \langle S^2 \rangle - \langle S \rangle^2$$

and

$$\sigma_m = \sigma_s / \sqrt{s}$$
Numerical integration and error VIII

derivation:
- random variable $x$
- $m$ runs with each $n$ trials ($= mn$ trials in total)
- index $\alpha$ labels a run, $i$ a single trial

result from one run (= measurement):

$$M_{\alpha} = \frac{1}{n} \sum_{i=1}^{n} x_{\alpha,i} \quad (503)$$

the arithmetic mean of all $mn$ trials is:

$$\overline{M} = \frac{1}{m} \sum_{\alpha}^{m} M_{\alpha} = \frac{1}{nm} \sum_{\alpha=1}^{m} \sum_{i=1}^{n} x_{\alpha,i} \quad (504)$$
Numerical integration and error IX

difference of a one run $\alpha$ and the total mean

$$e_\alpha = M_\alpha - \overline{M}$$  \hspace{1cm} (505)

Hence the variance (standard deviation$^2$) can be written for the runs as:

$$\sigma_m^2 = \frac{1}{m} \sum_{\alpha=1}^{m} e_\alpha^2$$  \hspace{1cm} (506)

Now finding the relation between $\sigma_m$ and $\sigma$ of the individual $mn$ trials.

Difference between one trial and the mean of one run:

$$d_{\alpha,i} = x_{\alpha,i} - \overline{M}$$  \hspace{1cm} (507)

Therefore the variance for all $mn$ trials:

$$\sigma^2 = \frac{1}{mn} \sum_{\alpha=1}^{m} \sum_{i=1}^{n} d_{\alpha,i}^2$$  \hspace{1cm} (508)
With help of Eq. (507) the Eq. (505) can be rewritten as:

\[ e_\alpha = M_\alpha - \bar{M} = \frac{1}{n} \sum_{i=1}^{n} (x_{\alpha,i} - \bar{M}) = \frac{1}{n} \sum_{i=1}^{n} d_{\alpha,i} \]  

(509)

Insert Eq. (509) into Eq. (506):

\[ \sigma^2_m = \frac{1}{m} \sum_{\alpha=1}^{m} \left( \frac{1}{n} \sum_{i=1}^{n} d_{\alpha,i} \right) \left( \frac{1}{n} \sum_{j=1}^{n} d_{\alpha,j} \right) \]  

(510)
The products in Eq. (510) consist of terms $i = j$ and terms $i \neq j$. As the trials are independent of each other, for large $n$ the differences $d_{\alpha,i}$ and $d_{\alpha,j}$ are on average as often negative as positive, i.e., the terms $i \neq j$ cancel out on average. What remains are the terms for $i = j$:

$$\sigma_m^2 = \frac{1}{mn^2} \sum_{\alpha=1}^{m} \sum_{i=1}^{n} d_{\alpha,i}^2$$  \hspace{1cm} (511)$$

By comparison with Eq. (508) one gets:

$$\sigma_m^2 = \frac{\sigma^2}{n} \Rightarrow \sigma_m = \frac{\sigma}{\sqrt{n}}$$  \hspace{1cm} (512)$$

$\square$
Importance sampling I

Idea: improve MC integration by a better sampling \( \rightarrow \) introduce a positive function \( p(x) \) with

\[
\int_a^b p(x) \, dx = 1 \quad (513)
\]

and rewrite integral \( \int_a^b f(x) \, dx \) as

\[
F = \int_a^b \left[ \frac{f(x)}{p(x)} \right] p(x) \, dx \quad (514)
\]

this integral can be evaluated by \textit{sampling according to} \( p(x) \):

\[
F_n = \frac{1}{n} \sum_{i=1}^{n} \frac{f(x)}{p(x)} \quad (515)
\]
Note that for the uniform case $p(x) = 1/(b - a)$ the sample mean method is recovered.

Now, try to minimize variance $\sigma^2$ of integrand $\frac{f(x)}{p(x)}$ by choosing $p(x) \approx f(x)$, especially for large $f(x)$

→ slowly varying integrand $f(x)/p(x)$

→ smaller variance $\sigma^2$
Example: Normal distribution

Evaluate integral \( F = \int_{a}^{b} f(x)dx = \int_{0}^{1} e^{-x^2}dx \) (error function)

<table>
<thead>
<tr>
<th></th>
<th>( p(x) = 1 )</th>
<th>( p(x) = Ae^{-x} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( (b - a) \ast r + a )</td>
<td>(- \log(e^{-a} - \frac{r}{A}))</td>
</tr>
<tr>
<td>( n )</td>
<td>( 4 \times 10^5 )</td>
<td>( 8 \times 10^3 )</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>( 0.0404 )</td>
<td>( 0.0031 )</td>
</tr>
<tr>
<td>( \sigma/\sqrt{n} )</td>
<td>( 6 \times 10^{-5} )</td>
<td>( 3 \times 10^{-5} )</td>
</tr>
<tr>
<td>total CPU time(^\dagger)</td>
<td>( 19 \text{ ms} )</td>
<td>( 0.8 \text{ ms} )</td>
</tr>
<tr>
<td>CPU time / trial</td>
<td>( 50 \text{ ns} )</td>
<td>( 100 \text{ ns} )</td>
</tr>
</tbody>
</table>

\(^\dagger\text{CPU time on a Intel Core i7-4771 3.5 GHz}\)

\( \rightarrow \) the extra time needed per trial for getting \( x \) from uniform \( r \) is usually overcompensated by the smaller number of necessary trials for same \( \sigma/\sqrt{n} \)
Importance sampling IV

Similar: **Metropolis algorithm** (Metropolis, Rosenbluth, Rosenbluth, Teller & Teller 1953) useful for averages of the form

\[
\langle f \rangle = \frac{\int p(x)f(x)dx}{\int p(x)dx}
\]  

(516)

Metropolis algorithm produces *random walk* (see below) of points \( \{x_i\} \) (1D or higher) with asymptotic probability distribution approaching \( p(x) \) for \( n \gg 1 \). Random walk from *transition probability* \( T(x_i \rightarrow x_j) \), such that

\[
p(x_i)T(x_i \rightarrow x_j) = p(x_j)T(x_j \rightarrow x_i) \quad \text{(detailed balance)} \quad (517)
\]

\[
e.g., \quad T(x_i \rightarrow x_j) = \min \left[ 1, \frac{p(x_j)}{p(x_i)} \right] \quad (518)
\]
Metropolis algorithm

1. choose trial position $x_{\text{trial}} = x_i + \delta_i$ with random $\delta_i \in [-\delta, +\delta]$
2. calculate $w = p(x_{\text{trial}})/p(x_i)$
3. if $w \geq 1$, accept and $x_{i+1} = x_{\text{trial}}$
4. if $w < 1$, generate random $r \in [0; 1]$
5. if $r \leq w$, accept and $x_{i+1} = x_{\text{trial}}$
6. if not, $x_{i+1} = x_i$

→ problem: optimum choice of $\delta$; if too large, only small number of accepted trials → inefficient sampling
if too small, only slow sampling of $p(x)$. Hence, rule of thumb: choose $\delta$ for which $\frac{1}{3} \ldots \frac{1}{2}$ trials accepted
also: choose $x_0$ for which $p(x_0)$ is largest → faster approach of $\{x_i\}$ to $p(x)$
Typical applications for Metropolis algorithm: computation of integrals with weight functions $p(x) \sim e^{-x}$, e.g.,

$$
\langle x \rangle = \frac{\int_0^\infty xe^{-x} \, dx}{\int_0^\infty e^{-x} \, dx} \quad (519)
$$

$$
\langle A \rangle = \frac{\int A(\vec{X}) e^{-U(\vec{X})/k_BT} \, d\vec{X}}{\int e^{-U(\vec{X})/k_BT} \, d\vec{X}} \quad (520)
$$

where the latter is the average of a physical quantity $A$ in a liquid system with good contact to a thermal bath, fixed number of particles (with $\vec{X} = (\vec{x}_1, \vec{x}_2, \ldots)$ of all particles) and volume $\rightarrow$ canonical ensemble, e.g.,

$$
\left\langle \frac{mv_{ik}^2}{2} \right\rangle = \frac{1}{2} k_B T \quad (521)
$$
Rejection sampling
(acceptance-rejection method)
Rejection sampling (acceptance-rejection method)

Problem: get random $x$ for any $p(x)$, also if $P(r)^{-1}$ not (easily) computable

Idea:

- area under $p(x)$ in $[x, x + dx]$ is probability of getting $x$ in that range
- if we can choose a random point in \textit{two dimensions} with uniform probability in the area under $p(x)$, then $x$ component of that \textit{point} is distributed according to $p(x)$
- so, on same graph draw $f(x)$ with $f(x) > p(x)$ $\forall x$
- if we can uniformly distribute points in the area under curve $f(x)$, then all points $(x, y)$ with $y < p(x)$ are uniform under $p(x)$
creation of arbitrary probability distributions with help of rejection sampling (especially for compact intervals \([a, b]\)):

- **let** \(p(x)\) **be the required distribution in** \([a, b]\)
- **choose a** \(f(x)\) **such that** \(p(x) < f(x)\) **in** \([a, b]\), **e.g.**, \(f(x) = c \cdot \max(p(x)) = \text{const.} \) where \(c > 1\)
- **it is** \(A := \int_a^b f(x)dx\), **i.e.** \(A(x)\) **must exist and be invertible**
- **generate** *uniform* random number in \([0, A]\) **and get the corresponding** \(x(A)\)
- **generate 2nd** *uniform* random number \(y\) **in** \([0, f(x)]\), **so** \(x, y\) **are uniformly distributed on** \(A\) **(area under** \(f(x)\))
- **accept** this point if \(y < p(x)\), **otherwise** **reject** it
Rejection sampling (acceptance-rejection method) III

Example: normal distribution $p(x)$ sampled by $f(x) = (x^2 + 1)^{-1}$

\[
\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad \text{(blue solid line)} \quad \text{sampled with help of the function} \quad \frac{1}{x^2+1} \quad \text{(red dashed)}
\]

whose integral is $\arctan(x)$ \text{(thick dashed red)} and hence $F(x)^{-1} = \tan(x)$
Rejection sampling (acceptance-rejection method) IV

Requirements:

- $p(x)$ must be computable for every $x$ in the interval
- $f(x) > p(x)$ always possible, as $\int_{-\infty}^{+\infty} p(x)dx = 1$ (i.e. $A > 1$)
- to get $x_0$ for a chosen value in $[0, A]$ requires usually: $\int f(x)dx = F$ is analytically invertible, i.e. $F(x)^{-1}$ exists

→ this is easy for compact $[a, b]$, e.g.,

$$F(x) = c \cdot \max(p(x)) \cdot (x - a) = k(x - a) \rightarrow x = F/k - a$$

for randomly chosen $F$ in $[0, A]$, where $A = k \cdot (b - a)$
Example: acceptance-rejection for normal distribution

double p(double x){ return exp(-0.5*x*x)/sqrt(2.*M_PI); }
double f(double x){ return 1./(x*x+1.); }
double inv_int_f(double ax){ return tan(ax - M_PI/2.); }
...
for (int i = 0; i < nmax; ++i){
    // get random value between 0 and A:
    ax = A * double(rand())/double(RAND_MAX);
    // obtain the corresponding x value:
    x = inv_int_f(ax);
    // get random y value in interval [0,f(x)]:
    y = f(x) * double(rand())/double(RAND_MAX);
    // test for y <= p(x) for acceptance:
    if ( y <= p(x) ) { cout << x << endl ;}
In our example:

- it is \( p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \) the standard normal distribution; normal distributions with \( \sigma \neq 1, \mu \neq 0 \) can be obtained by transformation
- the comparison function \( f(x) = \frac{1}{x^2+1} \) is always \( f(x) > p(x) \), moreover:
  - \( F(x) = \int_{-\infty}^{x} f(x') dx' = \arctan(x) - \arctan(-\infty) = \arctan(x) - \left( -\frac{\pi}{2} \right) \)
    \( \rightarrow F(x) = \arctan(x) + \frac{\pi}{2} \)
  - the total area \( A \) under \( f(x) \) is
    \( \int_{-\infty}^{x} f(x') dx' = \arctan(+\infty) - \arctan(-\infty) = \pi \)
  - the inverse \( F(x)^{-1} \), which returns \( x \) for a given value \( F \in [0, A] \) simply
    \( x = \tan \left( F - \frac{\pi}{2} \right) \)
  - efficiency of the acceptance is \( N_{\text{accepted}}/N_{\text{MAX}} = \int p(x)/\int f(x) = 1/\pi \approx 0.32 \), i.e. efficiency can be increased by choosing
    \( f(x) = \frac{1}{2} \frac{1}{x^2+1} \), then \( x = \tan \left( 2F - \frac{\pi}{2} \right) \rightarrow 63\% \) acceptance
Alternative choice I: $f(x) = \exp(-x)$ only for $x \geq 0$, then

- the integral $F(x)$ is $\int_0^x = -\exp(-x) + 1$
- the total area $\int_0^\infty \exp(-x)\,dx = 1 > 0.5 = \int_0^\infty p(x)$
- the inverse is $x = -\log(-x + 1)$
- to obtain also negative $x \to$ add random sign $\pm$
Alternative choice II: \( f(x) = 1.1 \cdot \max(p(x)) \) in the compact interval \([0,3]\), then

- it is \( \max(p(x)) = \frac{1}{\sqrt{2\pi}} \) in \([0,3]\)
  \[ \rightarrow f(x) = \frac{1.1}{\sqrt{2\pi}} \text{ in } [0,3] \]
- hence \( F(x)^{-1} \) is \( x = \frac{F\sqrt{2\pi}}{1.1} - 0 \).
- the total area \( A \) is \( \frac{1.1}{\sqrt{2\pi}} \cdot (3 - 0) \)

→ clear: this choice (const. function) works only for compact intervals, otherwise \( A \) is infinite and \( F(x)^{-1} \) does not exist.
Random walk
Random walk I

Idea: Brownian motion, e.g., dust in water (lab course: determination of diffusion coefficient $D = \frac{\langle x^2 \rangle}{2t}$)

frequent collisions between dust particles and water molecules
→ frequent change of direction
→ trajectory not predictable even for few collisions
→ motion of dust particle into any direction with same probability

→ Random walk
like “drunken sailor”: $N$ steps of equal length in arbitrary direction will lead to which distance from start point?
In one dimension:

- let’s start at $x = 0$, each step with length $\ell$
- for each step: probability $p$ for step to the right and $q = 1 - p$ to the left (independent from previous step)
- displacement after $N$ steps

$$x(N) = \sum_{i=1}^{N} s_i \quad \text{where } s_i = \pm \ell \quad \rightarrow \quad x^2(N) = \left( \sum_{i=1}^{N} s_i \right)^2 \quad (522)$$

- for $p = q = 1/2 \rightarrow$ coin flipping
- for large $N$: $\langle x(N) \rangle = 0$ expected
but for $\langle x^2(N) \rangle$? \rightarrow rewrite Eq. (522)

$$x^2(N) = \sum_{i=1}^{N} s_i^2 + \sum_{i \neq j=1}^{N} s_i s_j \quad (523)$$

where (for $i \neq j$) $s_i s_j = \pm \ell^2$ with same probability, so: $\sum_{i \neq j}^{N} s_i s_j = 0$

because of $s_i^2 = \ell^2 \rightarrow \sum_{i=1}^{N} s_i^2 = N\ell^2$:

$$\langle x^2(N) \rangle = \ell^2 N \quad (524)$$

especially for constant time intervals of the random walk

$$\langle x^2(t) \rangle = \frac{\ell^2}{\Delta t} N\Delta t \quad \left( = \frac{\ell^2}{\Delta t} t \right) \quad (525)$$

generally if $p \neq 1/2$ and $p$ for $+\ell$

$$\langle x(N) \rangle = (p - q)\ell N \quad (526)$$

\rightarrow linear dependance on $N$
Random walk IV

Example: Diffusion of photons in the Sun

Simplification: constant density, only Thomson scattering (free e\(^-\)) with cross section \(\sigma_T = 6.652 \times 10^{-25} \text{ cm}^2\)

mean free path length:

\[
\ell = \frac{1}{n\sigma_T} = \left( \frac{\rho}{m_H} \sigma_T \right)^{-1}
\]

(527)

one dimension \(\rightarrow\) only \(R = R_\odot\), total time \(t = N \Delta t\)

\[
\Rightarrow t = 9 \times 10^{10} \text{ s} = 2900 \text{ a} \ll t_{KH}(= 3 \times 10^7 \text{ a})
\]
Importance of the random walk model

many processes can be described by differential equation similar to diffusion equation (e.g., heat equation)

\[
\frac{\partial P(x, t)}{\partial t} = D \frac{\partial^2 P(x, t)}{\partial x^2}
\]  

(528)

with diffusion coefficient \( D \) and probability \( P(x, t)\,dx \) to find particle at time \( t \) in \([x, dx]\)
in 3 dimensions: \( \frac{\partial^2}{\partial x^2} \equiv \nabla^2 \)
Moments: mean value of a function \( f(x) \)

\[
\langle f(x, t) \rangle = \int_{-\infty}^{+\infty} f(x, t)P(x, t)\,dx
\]

(529)

\[
\Rightarrow \quad \langle x(t) \rangle = \int_{-\infty}^{+\infty} xP(x, t)\,dx
\]

(530)
Compute integral in Eq. (530) \( \rightarrow \) multiply Eq. (528) by \( x \) and integrate over \( x \)

\[
\int_{-\infty}^{+\infty} x \frac{\partial P(x, t)}{\partial t} \, dx = D \int_{-\infty}^{+\infty} x \frac{\partial^2 P(x, t)}{\partial x^2} \, dx \tag{531}
\]

left hand side

\[
\int_{-\infty}^{+\infty} x \frac{\partial P(x, t)}{\partial t} \, dx = \frac{\partial}{\partial t} \int_{-\infty}^{+\infty} xP(x, t) \, dx = \frac{\partial}{\partial t} \langle x \rangle \tag{532}
\]
right hand side via integration by parts, note that \( P(x = \pm \infty, t) = 0 \), as well as all spatial derivatives:

\[
D \int_{-\infty}^{+\infty} x \frac{\partial^2 P(x, t)}{\partial x^2} dx = D x \frac{\partial P(x, t)}{\partial x} \bigg|_{x = +\infty}^{x = -\infty} - D \int_{-\infty}^{+\infty} \frac{\partial P(x, t)}{\partial x} dx
\]

\[
= 0 - D P(x, t) \big|_{x = +\infty}^{x = -\infty} = 0
\]

\[
\Rightarrow \frac{\partial}{\partial t} \langle x \rangle = 0
\]

i.e. \( \langle x \rangle \equiv \text{const.} \) for all \( t \). For \( x(t = 0) = 0 \rightarrow \langle x \rangle = 0 \) for all \( t \).
Analogously for $\langle x^2(t) \rangle$: integration by parts twice

$$\frac{\partial}{\partial t} = 2D \int_{-\infty}^{+\infty} P(x, t)dx = 2D \quad (536)$$

$$\rightarrow \langle x^2(t) \rangle = 2D \, t \quad (537)$$

compare with Eq. (525) $\langle x^2(t) \rangle = \frac{\ell^2}{\Delta t} N \Delta t = \frac{\ell^2}{\Delta t} \, t$

$\rightarrow$ random walk and diffusion equation have same time dependence (linear)

(with $2D = \frac{\ell^2}{\Delta t}$)
Random numbers
Pseudorandom numbers I

for scientific purposes

- fast method to generate huge number of “random numbers”
- sequence should be reproducible

→ use deterministic algorithm to generate pseudorandom numbers

Linear congruential method

start with a seed $x_0$, use one-dimensional map

$$x_n = (ax_{n-1} + c) \mod m \quad (538)$$

- with integers $a$ (multiplier), $c$ (increment), $m$ (modulus)
- $m$ largest possible integer from Eq. (538) → maximum possible period is $m$ → obtain $r \in [0, 1)$ by $x_n/m$
- real period depends on $a$, $c$, $m$, e.g.,
  $a = 3$, $c = 4$, $m = 32$, $x_0 = 1$ → $1, 7, 25, 15, 17, 23, 9, 31, 1, 7, 25, \ldots$
  → period is 8 not 32
Better randomness can be obtained from physical processes:

- nuclear decay (*real* randomness!), e.g., → measure $\Delta t$ (difficult to implement)
- image noise, thermal noise (Johnson-Nyquist noise), e.g., → darkend USB camera (simple), special expansion cards with a diode
- “activity noise” in Unix:
  
  
  ```
  /dev/random
  /dev/urandom
  ```

  → random *bit* patterns from input/output streams (entropy pool) of the computer
  
  ```
  /dev/random blocks, if entropy pool is exhausted
  ```

For readout of Unix random devices need to interpret random bits(!) as numbers
Reading from urandom

E.g., by using fstream and union

```cpp
ifstream fin("/dev/urandom/") ;
union {unsigned int num ;
    char buf[sizeof(unsigned int)]; } u ;
fin.read(u.buf, sizeof(u.buf)) ;
cout << u.num ;
```

→ fstream reads only char, buf and num are at the same address → read bits in as char output as unsigned int
quality check for uniformly distributed random numbers

- **equal distribution**: random numbers should be fair
- **entropy**: bits of information per byte of a sequence of random numbers (same as equal distribution)
- **serial tests**: for $n$-tuple repetitions (often only for $n = 2, n = 3$)
- **run test**: for monotonically increasing/decreasing sequences, also for length of stay for a distinct interval
- and more …

Be careful!

There is no necessary or sufficient test for the randomness of a finite sequence of numbers.

→ can only check if it is “apparently” random
Correlation tests I

→ testing for “clumping” of numbers

Test for doublets

- define a square lattice $L \times L$ and fill each cell at random:
- array $n(x, y)$ with discrete coordinates
- choose random $1 \leq x_i, y_i \leq L$ where $x_i, y_i$ consecutive numbers of random number sequence
- fill cell $n(x_i, y_i)$ (e.g. set boolean to true)
- repeat procedure $t \cdot L^2$ times, $t$ is MC time step
- → similar to nuclear decay, therefore expected: fraction of empty cells $\propto \exp(-t)$
Simple correlation test

- just plot $x_{i+1}$ over $x_i$ → look for suspicious patterns

Correlation plot for linear congruential method with bad parameters

Same plot but for C++ `rand()` function
testing for randomness (also: numbers or detections)  
→ $\chi^2$ test

- let $y_i$ the number of events in bin $i$ and $E_i$ the expectation value
- e.g., $N = 10^4$ random numbers, $M = 100$ bins $\rightarrow E_i = 100$ (numbers/bin)
- the $\chi^2$ value (with $y_i$ measured number of random numbers in bin $i$):

\[
\chi^2 = \sum_{i=1}^{M} \frac{(y_i - E_i)^2}{E_i}
\]

(539)

measures the conformity of the measured and the expected distribution

- the individual terms in Eq. (539) should be $\leq 1$, so for $M$ terms $\chi^2 \leq M \rightarrow \text{reduced } \chi^2$ by deviding by $M \rightarrow \text{“minimum” red. } \chi^2 = 1$
- e.g., 5 independent runs yield $\chi^2 \approx 92, 124, 85, 91, 99 \rightarrow$ as expected, in general: $\chi^2$ should be small (but $\chi^2 = 0$ is suspicious)
Confidence

- need a quantitative measure that shows normal distribution of the “error” \((y_i - E_i)\) \(\rightarrow\) chi-squared distribution

\[
p(x, \nu) = \frac{1}{2^{\nu/2} \Gamma(\nu/2)} x^{(\nu-2)/2} e^{-x/2}
\]  

(540)

where \(\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt\) and \(\Gamma(z + 1) = z!\)

(541)

\(\rightarrow\) cumulated \(\chi^2\) distribution \(P(x, \nu)\):

\[
P(x, \nu) = \frac{1}{2^{\nu/2} \Gamma(\nu/2)} \int_0^x t^{(\nu-2)/2} e^{-t/2} dt
\]

(542)

with degrees of freedom \(\nu\), here: \(\nu = M - 1 = 99\), because of constraint \(\sum_{i=1}^{M} E_i = N\)
chi-square distribution

chi-square PDF for different degrees of freedom $\nu$

for $\nu > 30$ is $\sqrt{2x} - \sqrt{2\nu} - 1$ approximately normally distributed, for $\nu > 100$ is $x$ approximately normally distributed with $E = \nu$ and $\sigma = \sqrt{2\nu}$
• function \( Q(x, \nu) = 1 - P(x, \nu) \)

\( \rightarrow \) probability that \( \chi^2 > x \)

• we want to check: How likely to get a \( \chi^2 \) of, e.g., 124 (our largest measured \( \chi^2 \))? \( \rightarrow \) solve \( Q(x, \nu) = q \) (probability \( \chi^2 > x \) for given \( x, \nu \)) for \( x \), or look it up in tables

for \( \nu = M - 1 = 99 \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>138.9</th>
<th>134.6</th>
<th>123.2</th>
<th>110.6</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q )</td>
<td>0.005</td>
<td>0.01</td>
<td>0.05</td>
<td>0.2</td>
<td>0.5</td>
</tr>
</tbody>
</table>

• for our case: 1 out of 5 runs (20%) had \( y_2 = 124 \), but \( Q(x, \nu) \) implies for \( x = 123 \) only 5%, i.e., 1 out of 20 runs with \( \chi^2 \geq 123 \)

• therefore: confidence level < 95%, rather 80% (because of \( q = 0.2 \) for \( x = 111 \))

• try to increase confidence level: more runs \( \rightarrow \) if still only 1 out 20 with \( \chi^2 > 123 \) \( \rightarrow \) confidence level at 95%
Why Monte-Carlo integration?
Already seen: for 1d integration, dependence of truncation error on number of intervals ($\sim$ samples)

<table>
<thead>
<tr>
<th>method</th>
<th>$\sigma(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>rectangular rule</td>
<td>$N^{-1}$</td>
</tr>
<tr>
<td>trapezoid rule</td>
<td>$N^{-2}$</td>
</tr>
<tr>
<td>Simpson’s rule</td>
<td>$N^{-4}$</td>
</tr>
<tr>
<td>MC sample-mean method</td>
<td>$N^{-1/2}$</td>
</tr>
</tbody>
</table>

→ for 1d MC sample-mean inefficient integration method

Truncation error derived from Taylor series expansion of integrand $f(x)$:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i)(x - x_i)^2 + \ldots $$  \hspace{1cm} (543)

$$\int_{x_i}^{x_i+1} f(x) \, dx = f(x_i) \Delta x + \frac{1}{2} f'(x_i)(\Delta x)^2 + \frac{1}{6} f''(x_i)(\Delta x)^3 + \ldots $$ \hspace{1cm} (544)
Performance of integration techniques II

For the rectangular rule \((f(x_i)\Delta x)\), error \(\Delta_i\) in leading order for \([x_i, x_{i+1}]\) is

\[
\Delta_i = \left[ \int_{x_i}^{x_{i+1}} f(x) \, dx \right] - f(x_i)\Delta x \approx \frac{1}{2} f'(x_i)(\Delta x)^2
\] (545)

→ error per interval; as there are \(N\) intervals in total and \(\Delta x = (b - a)/N\)

→ total error for rectangular rule \(N \Delta_i \sim N(\Delta x)^2 \sim N^{-1}\)

Analogously for trapezoid rule, where we estimate \(f(x_{i+1})\) by Eq. (543):

\[
\Delta_i = \left[ \int_{x_i}^{x_{i+1}} f(x) \, dx \right] - \frac{1}{2} [f(x_i) + f(x_{i+1})] \Delta x
\] (546)

\[
= \left[ f(x_i)\Delta x + \frac{1}{2} f'(x_i)(\Delta x)^2 + \frac{1}{6} f''(x_i)(\Delta x)^3 + \ldots \right]
\] (547)

\[
- \frac{1}{2} \Delta x \left[ f(x_i) + f(x_i) + f'(x_i)\Delta x + \frac{1}{2} f''(x_i)(\Delta x)^2 + \ldots \right]
\] (548)

\[
\approx -\frac{1}{12} f''(x_i)(\Delta x)^3 \quad \rightarrow \quad \text{total error} \sim N^{-2}
\] (549)
For Simpson’s rule $f(x)$ is approximated as parabola on $[x_{i-1}, x_{i+1}]$ → terms $\sim f''$ cancel, moreover because of symmetry terms $\sim f'''(\Delta x)^4$ cancel → error for interval $[x_i, x_{i+1}]$ is $\sim f^{(4)}(x_i)(\Delta x)^5$ and total error for $[a, b]$ is $\sim N^{-4}$

Integration error in 2d extend previous estimates for rectangular rule in 2d, so for $f(x, y)$: integral → sum of volumes of parallelograms with cross section area $\Delta x \Delta y$ and height $f(x, y)$ at one corner

Taylor series expansion of $f(x, y)$

$$f(x, y) = f(x_i, y_i) + \frac{\partial f(x_i, y_i)}{\partial x}(x - x_i) + \frac{\partial f(x_i, y_i)}{\partial y}(y - y_i) + \ldots$$  \hspace{1cm} (550)

$$\Delta_i = \left[ \int \int f(x, y) \, dx \, dy \right] - f(x_i, y_i) \Delta x \Delta y$$  \hspace{1cm} (551)
Now, substitute Taylor expansion Eq. (550) into error estimate Eq. (551), integrate each term → term $\sim f$ cancels out and $\int (x - x_i)dx = \frac{1}{2}(\Delta x)^2 \rightarrow \int dy$ gives another factor $\Delta y$; similar for $(y - y_i)$
As $O(\Delta y) = O(\Delta x)$, error for interval $[x_i, x_{i+1}]$ and $[y_i, y_{i+1}]$ is

$$\Delta_i \approx \frac{1}{2} [f'_x(x_i, y_i) + f'_y(x_i, y_i)](\Delta x)^3$$ \hspace{1cm} (552)

→ error for one parallelogram $\sim (\Delta x)^3$, for $N$ parallelograms $N \cdot (\Delta x)^3$
But in 2d: $N = A/(\Delta x)^2 \rightarrow$ total error $N^{-1/2}$ (whereas in 1d: $N^{-1}$)
Analogously for trapezoid rule in 2d $N^{-1}$, for Simpson’s rule in 2d $N^{-2}$
In general: if in 1d integration error $\sim N^{-p} \rightarrow$ integration error in $d$ dimensions $\sim N^{-p/d}$ (curse of dimensionality)
In contrast: MC integration error $\sim N^{-1/2}$ independent of $d \rightarrow$ superior for large $d$
(think about integrals $\int_V \int_V f dp^3 dx^3$ in statistical mechanics)
Integrals of functions of more than 1 variable, over regions with $d > 1$, are difficult!

1. **Function evaluation:** if $n$ function calls required for some accuracy in 1d $\rightarrow \sim n^d$ samples needed for $d$ dimensions (e.g., 30 calls in 1d vs. approx. 30,000 in 3d)

2. **Integration region in $d$ dimensions defined by $d - 1$ dimensional boundary $\rightarrow$ can be very complicated for $d > 1$ (e.g., not convex, not simply connected)

Ad 1.) $\rightarrow$ try to reduce integral to lower dimensions by exploiting symmetry of function and boundary and changing coordinates. E.g., spherically symmetric function over spherical region $\rightarrow$ in polar coordinates 1d integral
Example: PoWR code for expanding atmospheres

- non-LTE (i.e. $\vec{n}(\vec{J})$ from statistical equations + ALI $\rightarrow$ Newton’s method) radiative transfer in wind (i.e. CMF RT with Mio. of frequency points $K$, coarsend $\vec{J}(\vec{n})$ for $\vec{n} \rightarrow K \approx 1000$) $\rightarrow$ iteratively solved

- assuming spherical symmetry with, e.g., $L = 50$ depth-points, typically for each iteration $\approx 5$ s, in total $\approx 1000$ iterations $\rightarrow \sim h$

- in 3D: $2500 \times$ more “depthpoints” $\rightarrow$ each iteration now $3.5$ h (!) $\rightarrow$ total $\frac{1}{2}$ a
Ad 2.)

- if boundary complicated, integrand not strongly peaked in very small regions, relatively low accuracy required → MC integration! (see below)
- if boundary simple, smooth integrand, (+ high accuracy required) → repeated 1d integrals or multidimensional quadrature
- if integrand peaks in certain regions → split integral into several “smooth” regions (requires knowledge of behaviour of integrand)
Repeated 1d integration

Let $d = 3$ with $x, y, z$ and boundaries $[x_1, x_2]$, $[y_1(x), y_2(x)]$, $[z_1(x, y), z_2(x, y)] \rightarrow$ find $x_1, x_2$ and functions $y_1(x), y_2(x), z_1(x, y), z_2(x, y)$ such that

$$
\int \int \int dx \, dy \, dz \, f(x, y, z) = \int_{x_1}^{x_2} dx \, \int_{y_1(x)}^{y_2(x)} dy \, \int_{z_1(x, y)}^{z_2(x, y)} dz \, f(x, y, z)
$$

(553)

Example: 2d integral over circle $r = 1$ centered on $(0, 0)$

$$
\int_{x_1=-1}^{x_2=+1} dx \, \int_{y_1(x)=\sqrt{1-x^2}}^{y_2(x)=\sqrt{1-x^2}} dy \, f(x, y)
$$

(554)
Innermost integration yields a function $G(x, y)$:

$$G(x, y) := \int_{z_1(x, y)}^{z_2(x, y)} f(x, y, z) \, dz \quad (555)$$

then integration over $y$ yields $H(x)$:

$$H(x) := \int_{y_1(x)}^{y_2(x)} G(x, y) \, dy \quad (556)$$

finally the overall integral $I$ is

$$I = \int_{x_1}^{x_2} H(x) \, dx \quad (557)$$
How to integrate in higher dimensions VI

instead of using fixed Cartesian mesh of points, better evaluate function at suitable $x$ locations (along $y$-axis);
→ integration call many more times than outer

Implementation of Eq. (555)-(557) requires 3 separate copies of some 1d integration routine, so one for each $x, y, z$ integration or recursive calls of this routine
How to integrate in higher dimensions VII

Example: Code snippet for 3d iterated integration

```fortran
! user provides func(x,y,z), y1(x),
! y2(x), z1(x,y), z2(x,y)

SUBROUTINE quad3d(x1,x2,ss)
REAL ss,x1,x2,h
call quadx(h,x1,x2,ss)
return
END

FUNCTION f(zz)
REAL f,zz,func,x,y,z
COMMON /xyz/ x,y,z
z=zz
f=func(x,y,z)
return
END

FUNCTION g(yy)
REAL g,yy,f,z1,z2,x,y,z
COMMON /xyz/ x,y,z
REAL ss
y=yy
call quadz(f,z1(x,y),z2(x,y),ss)
g=ss
return
END

FUNCTION h(xx)
REAL h,xx,g,y1,y2,x,y,z
COMMON /xyz/ x,y,z
REAL ss
x=xx
call quady(g,y1(x),y2(x),ss)
h=ss
return
END
```
MC integration in higher dimensions I

Example: Mass and center of mass of cut torus

Section of a torus with radius $R$ and cross section radius $r$

$$z^2 + (\sqrt{x^2 + y^2} - R)^2 \leq r$$  \hspace{1cm} (558)

section defined by

$$x \geq a \quad y \geq b$$  \hspace{1cm} (559)

Need to evaluate following integrals

$$M = \int \rho \, dx \, dy \, dz \quad M_x = \int x \rho \, dx \, dy \, dz$$  \hspace{1cm} (560)

$$M_y = \int y \rho \, dx \, dy \, dz \quad M_z = \int z \rho \, dx \, dy \, dz$$  \hspace{1cm} (561)

i.e., $x$-coordinate of center of mass is $x = M_x / M$ and so on
MC integration for a torus (centered on origin, outer radius = 4, inner radius = 2) section, where $x \leq 1$ and $y \leq -3$, i.e., bounds given by intersection of two planes. Integration limits cannot be easily given in analytically closed form.

Choose region that encloses torus section, e.g., rectangular box with $1 \leq x \leq 4$, $-3 \leq y \leq 4$, and $-1 \leq z \leq 1$, hence total volume of box is $V = 3 \times 7 \times 2$.
Example: code sniplet for MC integration of torus section

```c
int N = 1000; // sample points
double den = 1.; // density rho
double sw = 0., varw = 0.; // mass and variance
double swx = 0., varx = 0.; // x-coordinate and var. for center of mass
...
for (i = 0; i < N; ++i) {
    x = 1. + 3. * rand();
y = -3. + 7. * rand();
z = -1. + 2. * rand();
    if (pow(z*z + (sqrt(x*x + y*y) -3.), 2.) <= 1. ) {
        sw = sw + den; varw = varw + den*den;
        swx = swx + x * den; varx = varx + (x*den)*(x*den);
        ...
    }
} }
w = V * sw / n; // mass of torus
x = V * swx / n; // x-coordinate
dw = vol * sqrt((varw / N - (sw/N)*(sw/N)) / N);
dx = vol * sqrt((varx / N - (swx/N)*(swx/N)) / N);
```
Conclusions about advantage of MC integration

1. MC integration error decreases independent of dimension with $\sim N^{-1/2}$ → superior for integrals with many integration variables (e.g., phase space integrals, QM)

2. MC integration easy to implement for any geometry → superior for 3d models without simple symmetry (e.g., spherical symmetry)
Techniques of parallelization
Neutron transport with packets I

So far: single neutron $n^0$

Improvement/speed up: consider “neutron packets”, i.e. we follow an ensemble of neutrons (which advances with random $\ell, \cos \theta$ as before)

$\rightarrow$ determine fraction of the scattered and captured neutrons

1. scattering: fraction of scattered $n^0$: $p_s$, fraction of absorbed $n^0$: $p_c$
2. scattering: fraction of scattered $n^0$: $p_s^2$, fraction of absorbed $n^0$: $p_c p_s$

$m$th scattering: fraction of scattered $n^0$: $p_s^m$, fraction of absorbed $n^0$: $p_c p_s^{m-1}$

so, after $m$th scattering:

$\rightarrow$ total fraction of captured neutrons:

$f_c = p_c + p_c p_s + p_c p_s^2 + \ldots + p_c p_s^{m-1}$

$\rightarrow$ total fraction of scattered neutrons:

$f_s = p_s^m$

$\rightarrow$ if position $x < 0$: add $f_s$ to $f_{\text{refl}}$

$\rightarrow$ if position $x > t$: add $f_s$ to $f_{\text{trans}}$
instead of individual photons, use energy packets of same frequency $\nu$ ($\epsilon(\nu) = n\hbar\nu$) and same energy $\epsilon_0 \rightarrow$ different n

scattering: $\nu_e = \nu_a$

absorption leads to re-emission following: $\epsilon(\nu_e) = \epsilon(\nu_a)$, no packet (= energy) lost or created $\rightarrow$ divergence-free radiation field

macro-atoms with discrete internal states, activation via r-packet (radiative) of appropriate CMF frequency or k-packet (kinetic); active macro-atom performs internal transitions and gets inactive by emission of r- or k-packet
Parallelization
Many runs in MC simulations required for reliable conclusions

Often: Result of one run (e.g., path of a neutron through a plate) independent from other runs

→ **Idea**: acceleration by parallelization

**Problem**: concurrent access to memory resources, i.e. variables (e.g., $n_s$, $f_{\text{refl}}$)

**Solution**: special libraries that enable multithreading (e.g., OpenMP) or multiple processes (e.g., MPI) for one program

→ **insert**: pipelining, vectorization, parallelization
What influences the performance of a CPU (= runtime of your code)?

- architecture/design: out-of-order execution (all x86 except for Intel Atom), pipelining (stages), vectorization units (width)
- cache sizes (kB ... MB) and location: L1 cache for each core, L3 for processor
- clock rate (\( \sim \)GHz): only within a processor family usable for comparison due to different number of instruction per clock (IPC) of design, even more complicated because of variable clock rates (base, peak) to exploit TDP (thermal design power) → impact on single-thread performance
- number of cores (1 ...): → impact on multi-thread performance
splitting machine instruction into a sequence

-independent execution of instructions, each consisting of
  - instruction fetching (IF)
  - instruction decoding (ID) + register fetch
  - execution (EX)
  - write back (WB)

operations of instructions are processed at the same time → quasi parallel execution, higher throughput
NetBurst disaster

Pentium 4 (2000-2008) developed to achieve > 4GHz (goal: 10 GHz) clockrate by a several techniques, i.a., long pipeline:

- 20 stages (Pentium III: 10) up to 31 stages (Prescott core)
- smaller number of instructions per clock (IPC) (!)
- increased branch misprediction (also only 10%, improved by 33% for Pentium III)
- larger penalty for misprediction

→ compensated by higher clock rate

higher clock rate → higher power dissipation, especially for 65 (Presler, Pentium D), 90 (Prescott) up to 180 nm (Williamette) structures
→ power barriere at 3.8 GHz (Prescott)
SSE and AVX I

- **SSE** - Streaming SIMD Extensions (formerly: ISSE - Internet SSE)
- **SIMD** - Single Instruction Multiple Data (→ cf. Multivec, AMD3Dnow!), introduced with Pentium III (Katamai, Feb. 1999)
SSE and AVX II

- enables vectorization of instructions (not to be confused with pipelining or parallelization), often new, complex machine instructions required,
  e.g., PANDN → bitwise NOT + AND on packed integers
- comprises 70 different instructions, e.g., ADDPS – add packed single-precision floats (two “vectors” each with 4 32 bit) into a 128 bit register
- works with 128 bit registers (3Dnow! only 64 bit), but first execution units (before Core architecture) only with 64 bit
- AVX - Advanced Vector Extensions with 256 bit registers, theoretically doubled speed! since Sandy Bridge (Intel Core 2nd generation, e.g., i7-2600 K) and Bulldozer (AMD)
  → AVX-512 with 512 bit registers in Skylake (6th generation, e.g., Core i7-6700)
SSE and AVX III

- supported by all common compilers, e.g.,
  ifort -sse4.2
  g++ -msse4.1
- very easy (automatic) optimization, e.g., for unrolled loops → vectorization

Caution!

Different precisions for SSE-doubles (e.g., 64 bit) and FPU-doubles (80 bit), especially for buffering, so results of doubles, e.g.,
xx = pow(x,2) ;
sqrt( xx - x*x) ;
usually not predictable
Multi-cores

- originally one core per processor, sometimes several processors per machine/board (supercomputer)
- many units (ALU, register) already multiply existing in one processor
- first multi-core processors: IBM POWER4 (2001); desktop → Smithfield (2005), e.g., Pentium D
- Hyper-threading (HT): introduced in Intel Pentium 4 → for better workload of the computing units by simulation of another, logical processor core (compare: AMD Bulldozer design with modules)
- today: up to 64 cores for desktop (AMD Zen: Ryzen Threadripper 2990WX) or server (AMD Epyc 75...), or even more, 72 (Xeon Phi 7290)
- arms race of cores instead of clock rate (NetBurst disaster)
parallelization done by multithreading (from \textit{thread})

because of “The free lunch is over” $\rightarrow$ no simple acceleration more of a \textit{single-thread} program (exceptions: Turbo Boost, Turbo Core, in some ways larger caches may help) by pure increase in clock rate

supported by, e.g., \texttt{OpenMP} (shared memory), see below

different from: multiprocessing parallelization via MPI $\rightarrow$ distributed computing (cf. Co-array Fortran) but can be combined
General-purpose computing on graphics processing units → further development of graphic cards

- e.g., Tesla, Fermi (Nvidia)
  → *Piz Daint* (Swiss, 10th in 2020 Top500) with 5,272 Nvidia Tesla K20X processors (each with 2,688 CUDA cores) + 5,272 × 8 Xeon Cores reaches more than 25 petaflops

- so-called shaders → highly specialized ALUs, often only with single precision (opposite concept: Intel’s Larrabee)

- programming (not only graphics) via CUDA (Nvidia) or OpenCL (more general)

- OpenCL → parallel programming for arbitrary systems, also NUMA (non-uniform memory access), but very abstract and complex concept and also complicated C-syntax

- CUDA support, e.g., by PGI Fortran compiler → simple acceleration without code modifications
OpenMP
OpenMP - Open Multi-Processing

- for shared-memory systems (e.g., multi core) per node
- directly available in g++, gfortran, and Intel compilers
- insertion of so-called OpenMP (pragma) directives:

Example: for loop

### C++
```cpp
#include <omp.h>
...
#pragma omp parallel for
for ( ... )
{ ... }
```

### Fortran
```fortran
USE omp_lib ! ifort
 !$OMP PARALLEL DO
 DO i = 1, n
  ....
 ENDDO
 !$OMP END PARALLEL DO
```

instructs parallel execution of the `for` loop, i.e., there are copies of the loop (different iterations) which run in parallel → only the labeled section runs in parallel
pragma directives are syntactically seen comments, i.e., invisible for compilers without OpenMP support

- realization during runtime by threads
- number of used threads can be set, e.g., by environment variable

```
export OMP_NUM_THREADS=4
```

obvious: per core only one thread can run at the same time (but: Intel’s hyper-threading, AMD’s Bulldozer design) → reasonable: number of threads = number of cores

Caution!

Distributing and joining of threads produces some overhead in CPU / computing time (e.g., copying data) and is therefore only efficient for complex tasks within each thread. Otherwise multithreading can slow down program execution.
Including the OpenMP library:

<table>
<thead>
<tr>
<th>C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>#ifdef _OPENMP</td>
<td>! only needed for ifort:</td>
</tr>
<tr>
<td>#include &lt;omp.h&gt;</td>
<td>!$ use omp_lib</td>
</tr>
<tr>
<td>#endif</td>
<td></td>
</tr>
</tbody>
</table>

→ instructions between `#ifdef _OPENMP` and `#endif` (Fortran: following `!$`) are only executed if compiler invokes OpenMP

Compile via

- `g++ -fopenmp`
- `gfortran -fopenmp`
- `ifort -openmp`
useful: functions specific for OpenMP, e.g., for number of available cores, generated (maximum) number of threads, and current number of threads:

- `omp_get_num_procs()` // number of processor cores
- `omp_get_max_threads()` // max. number of threads generated
- `omp_get_num_threads()` // number of the current threads
- `omp_get_thread_num()` // number of the current thread
Very important: organization of the visibility of the involved data, i.e. assign attributes \texttt{shared} or \texttt{private} to thread variables

**shared**

\rightarrow \texttt{default}

data are visible in all threads and can be modified

in contrast to:

**private**

each thread has its own copy of the data, which are invisible for other threads, especially from outside of the parallel section

moreover, there are further so-called data clauses, e.g., \texttt{firstprivate} (initialization before the parallel section), \texttt{lastprivate} (last completed thread determines the value of the variable after the parallel section) and many more ...
Example private

C++:

```c++
int j, m = 4;
#pragma omp parallel for private (i,j)
for (int i = 0 ; i < max ; i++)
{ j = i + m;
  ...
}
```

Fortran:

```fortran
!$OMP OMP PARALLEL DO PRIVATE (i,j)
DO i = 0, max
  j = i + m
  ...
ENDDO
!$OMP END PARALLEL DO
```

→ loop variable i and variable j as “local” copies in each thread
→ variable m implicitly shared
Some more OpenMP directives:

```plaintext
#pragma omp parallel
→ parallel section also possible without a loop, section is executed per thread, `{ }` block required:

C++:
```plaintext
#pragma omp parallel
{
  cout << "Hi!" << endl ;
}
```

Fortran:
```plaintext
!$OMP PARALLEL
  print *, 'Hi!'
!$OMP END PARALLEL
```

```plaintext
#pragma omp critical
→ within a parallel section
is executed by each thread, but never at the same time (avoiding race conditions for shared resources)
```
schedule(runtime)

e.g.,

```bash
#pragma omp parallel for schedule (runtime)
→ way of distributing the parallel section to threads is defined at runtime, e.g., by (bash)

export OMP_SCHEDULE "dynamic,1"
→ each thread gets a *chunk* of size 1 (e.g., one iteration) as soon as it is ready

export OMP_SCHEDULE "static"
→ the parallel section (e.g., loop iterations) is divided by the number of threads (e.g., 4) and each thread gets a chunk of the same size
```
Useful for performance measurement:

```c
omp_get_wtime() // → returns the so-called wall clock time (not the cpu time)
```

```c
omp_get_thread_num() // → returns the number of the current thread
```

Weblinks:

http://www.openmp.org/
especially the documentation of the specifications:
Programming in Fortran - Part 1

Introduction
Fortran - History

Fortran = Formula translation

- first high level language (John Backus\(^2\) 1957) – in contrast to Assembler
- contains a lot of builtin features, like power "**" and a data type for complex numbers
- initially on punch[ed] cards with → 80 characters per line

Advantages of punch cards (K. Ganzhorn 1966)

- machine and visual readability
- mechanical dublica-, mixa- and sortability
- outstandingly superior signal-to-noise ratio (> 10\(^6\))
- cheap
- universal suitability for machine data input and output

\(^2\)Turing Award 1977, also ALGOL 58
Punch card
Fortran versions/standards

- FORTRAN I (1954-57)
- FORTRAN II (1958)
- FORTRAN IV (1961)
- FORTRAN 66 (ANSI/ASA)
- FORTRAN 77 (structured, before: GOTO)
- Fortran 90 (free form)
- Fortran 95
- Fortran 2000
- Fortran 2003 (polymorphism, inheritance, object-oriented)
- Fortran 2008 (Coarray Fortran)

Conventions

In the following we will only consider FORTRAN 77 with some elements from Fortran 90/95.
Why Fortran? I

Fortran can be easier (and therefore better) optimized by *Compilers* (since Backus) and is also *easy to learn*. This is amongst others due to:

- The Fortran programming language is more strict (=less flexible), e.g., loop variables (counter) cannot be changed within the loop:

```fortran
DO I = 1 , 10
   I = 10
ENDDO
```

*not permitted!* (Error, e.g., *gfortran Error: Variable 'I' cannot be redefined inside loop*)

→ real loops, (max.) number of iterations determined before loop entry
Why Fortran? II

- static data types: data types are known at compile time
- *but*: no explicit declaration (type checking) of arguments of procedures:
  
  ```fortran
  SUBROUTINE VEC_X_MATRIX (VEC, MATRIX, FLAG)
  
  therefore also possible: passing multi-dimensional arrays without giving the dimensions (e.g. re-dimensioning in procedure possible)
  
  no pointer needed (but they exist, e.g., Cray-pointer), procedures get only the start address of the arrays/variables
  
  ```fortran
  CALL ADDZ (ARG1,ARG2(2:4,2:5))
  
  ```fortran
  SUBROUTINE ADDZ (ARG1,ARG2)
  DIMENSION ARG2(3,4)
  
  ```
Fortran is a *procedural* language without templates, header files, etc. → short compile time, highly optimizable by compilers

**“Weaknesses” of Fortran**

On the other hand: no type checking when passing variables to procedures (exception: optional INTERFACE).

Many Fortran programmers use GOTO instructions, one of the main reasons of “Spaghetti” code

There is no general exception handling, no good graphics library, etc.

---

but: modules
## Economical aspects

### Table: Survey among IT people

<table>
<thead>
<tr>
<th>Skill</th>
<th>% of the respondent</th>
<th>annual income / €</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>16.8</td>
<td>55 304</td>
</tr>
<tr>
<td>C++</td>
<td>19.2</td>
<td>53 568</td>
</tr>
<tr>
<td>Cobol</td>
<td>1.7</td>
<td>65 813</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.6*</td>
<td>67 185</td>
</tr>
<tr>
<td>Java</td>
<td>28.8</td>
<td>54 164</td>
</tr>
<tr>
<td>SQL</td>
<td>43.0</td>
<td>50 881</td>
</tr>
</tbody>
</table>

3446 respondents, c’t 2011, 6
* = 21 respondents
Spreading/applications of Fortran

- LINPACK → benchmarking of supercomputers (PFLOPS) & LAPACK → systems of linear equations
- astrophysics:
  - radiative transfer (e.g., CMFGEN, PoWR, FASTWIND, TMAP, MOCASSIN),
  - CMBFAST (but outdated),
  - hydrodynamics (ZEUS),
  - data analysis (e.g., ABSOLUT for absorption lines)
- particle physics: SIMDET (simulation of detectors of colliders),
  - PYTHIA (until V6.4, MC simulation of particle decays)
- but also application software: WRplot
Fortran compilers

Current Fortran compilers (July 2018):

- gfortran (GCC) 8.1 (May 2018)
- ifort (Intel) 17.0 (25.2.2016)
- PGI CUDA Fortran (PGI) 18.5 (31.5.2018)
- g95 (Andy) 0.93 (Oct 2012!)
- openf90/openf95 (AMD x86 Open64) 4.5.2 (2014)
- and many others . . .

each of these compilers has specific advantages and disadvantages, some compilers (fortran, g95) do not support all features (e.g., ENCODE).

Moreover, gfortran is only a front end for gcc and translates Fortran into an abstract syntax tree (AST) → less optimization potential.

The PGI compiler and ifort are commercial products (but can be obtained for free under specific conditions), the other compilers are for free and mostly open source.
Calling the compiler

**gfortran**

- `gfortran myprogram.f` → creates executable program `a.out` from source file `myprogram.f`
- `gfortran -o myprogram myprogram.f` → creates executable program `myprogram`
- `gfortran -c myprogram.f` → creates object file `myprogram.o`
- `gfortran -o myprogram myprogram.o` → links the object file against the runtime libraries to create the program

**ifort**

- first step: set compiler path etc., e.g.,
  
  `source /opt/intel/[...]/bin/ifortvars.sh intel64`
- `ifort -o myprogram.exe myprogram.f` → as for gfortran
- Intel Fortran Language Reference, about 900 pages, extensive reference containing many examples
- Fortran 90/95 explained, Michael Metcalf & John Reid, 1996
...what a developer usually has to know:

1. Where to put the semicolons?

2. How to insert a comment?
In Fortran 77 (default source format):

1. one line = one instruction
   (end of line usually after column 72, continuation lines possible via a
    character in column 6 or via & in the previous line).
   A semicolon separates multiple instructions within a single line.

   1234567 ... 72 column number
   WRITE (*,’(A,A,A)’) > "Hello"
   & " world!" ; END

→ columns 1 - 5 for label (integer number) reserved, e.g.,

   123456
   IF (BERROR) GOTO 999
   ...
   999 STOP ’INPUT ERROR’
A comment line is indicated by a comment indicator (!,C,*) in the first column, since Fortran 90, also via a preceding exclamation mark “!”.
Also, everything after column 72 is ignored (comment).

0000000001 ... 77777777
1234567890 ... 01234567
C a comment
    ! comment
    EXP(-PI) comment
A simple program:

```
PROGRAM myprogram
WRITE (6,'(A4)') "O.K."
STOP
END
```
Program - basic instructions

- **PROGRAM** program name
  → optional, recommended

- **STOP**
  → optional, recommended

- **END [PROGRAM program name]**
  → obligatory

**Hints:**

- Fortran doesn’t distinguish lowercase and uppercase characters for instructions and variable names.
- Fortran can be written without blanks, e.g., **PROGRAM myprogram**
- default file extensions for Fortran source files:
  .f .for .f90 (and some more)
  → depends on compiler
the universal output instruction is

\texttt{WRITE}

\textsuperscript{4} E.g.:

\texttt{WRITE (*,*) "Hello world!" , "Hello!"

Meaning of the *:
\texttt{WRITE (UNIT,FORMAT)}
The instructions PRINT and WRITE(*,*) write to stdout (terminal, shell). Furthermore, WRITE can write to a “unit” (device):

   WRITE(J,*)

where J is an integer number with $0 \leq J \leq 2147483640$ (ifort)

<table>
<thead>
<tr>
<th>unit</th>
<th>uninitialized meaning</th>
<th>channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*Always: screen / keyboard</td>
<td>STDOUT / STDIN</td>
</tr>
<tr>
<td>0</td>
<td>screen (terminal)</td>
<td>STDERR</td>
</tr>
<tr>
<td>5</td>
<td>keyboard (terminal)</td>
<td>STDIN</td>
</tr>
<tr>
<td>6</td>
<td>screen (terminal)</td>
<td>STDOUT</td>
</tr>
</tbody>
</table>
The import of data is done by

READ

- syntax similar to WRITE, e.g.
  
  \texttt{READ (*,*) RADIUS}
  
  i.e. \texttt{READ (UNIT,FORMAT)}

- problem: wrong type at input (e.g., string instead of float) causes program crash, therefore always:

- catch wrong input, e.g.:
  
  100 \texttt{READ(*,*,ERR=101) RADIUS}
  
  ... 
  
  \texttt{STOP}
  
  101 \texttt{PRINT * , 'Wrong input, again please!' ; GOTO 100 END}
Format statements I

- instead of * in READ / WRITE better use format string
- e.g., WRITE (*,'(A,F10.2)') 'Radius is ', RADIUS
- enclose string with ’ ’ and put them in parentheses ()

Example:

- A : text, A26 → 26 characters
- Fn.d : fixed point, F4.1 → one decimal place, two digits left before decimal point
- En.dEz : floating-point, E10.1E2 → 10 digits in total, 1 digit after decimal point, 2 digits for exponent

Hint

Reading any format of numbers (integer, fixed, float) via fixed point format with number of positions after decimal point = zero into a REAL variable, e.g.,

F20.0
Format statements II

Format instruction

- the format can also be set via an instruction, e.g.,

```
READ (5,200) KARTE
200 FORMAT (A132)
```

or use `READ(5,FMT=200)` for better readability

- advantage of the FORMAT instruction:
  - re-usability (e.g., multiple READ / WRITE statements with same format)
  - clarity: collecting FORMAT statements between STOP and END PROGRAM

- format strings can be constructed during runtime! (flexibility)
Fortran allows the use of variables without explicit declaration. → data type (INTEGER, REAL) is determined by initial letter of the variable name:

- initial letter I, J, K, L, M, N for integer numbers, e.g.,
  DO I = 1, K → counting loop
  Nparticipants = 8
- all others: floating points (REAL), e.g.,
  PI = 3.141
  XMASS = 2.E33

more on that later ...
FORTRAN 77 – fixed form:

```fortran
77777
1234567890 34567
WRITE (*,*) ... comment
```

Fortran 90 – free form:

```fortran
WRITE (*,*) ... ! comment
```

- default: fixed form
- free form via: file extension `.f90`
  or compiler option:

```bash
  gfortran:  -ffree-form  
  ifort:    -free
```
Programming in Fortran - Part 2

Data types, input, output, files, execution control
We already know:

- integer number variables (INTEGER) start with: I, J, K, L, M, N
- floating point number variables (REAL): any other letter

with help of the IMPLICIT instruction it is possible to assign other letters to the variables, e.g.,

- IMPLICIT COMPLEX (c, z)
- IMPLICIT LOGICAL (b)
Implicit declaration II

Advantage of implicit declarations
compact declaration block (only needed for arrays and other data types)

Big disadvantage:
Typos of variable names are not detected at compile time. E.g.,

\[
\begin{align*}
\text{RSUN} &= 69.57\text{E9} \\
\text{RADIUS} &= \text{RADIUS} / \text{RSUNN}
\end{align*}
\]

often leads to errors which are difficult to reproduce
(here: division by 0)
Therefore better and always recommended:

```
IMPLICIT NONE
```

as the first statement in each program (or function/subroutine). This statement switches off implicit declaration and requires the explicit declaration of a variable before use (like, e.g., in C/C++)

**Example**

```
PROGRAM SIMPLE
IMPLICIT NONE
REAL RSUN, RADIUS
```
The declaration block

- Explicit declaration of variables is exclusively done in the “declaration block”.
- This block must be at the beginning of the program.
- The initialization of variables via “=” can only be done after this block.

Exceptions:

- Constants (PARAMETER) can/should be initialized in the declaration block.
- Via the DATA instruction variables can be initialized in the declaration block. Note that this is done at compile time!
Implicit declaration V

Example - declaration

Correct:

```plaintext
REAL R, S, PI, E
DATA PI, E /3.141, 2.718/
INTEGER I, NMAX
PARAMETER (NMAX=100)
CHARACTER LINE*132
```

Incorrect:

```plaintext
REAL X
X = 3.0 ! not permitted:
    ! initialization via =
    ! in the declaration block
INTEGER Y
```
- completely analogous to \texttt{int} in \texttt{C}, e.g.,

\begin{verbatim}
INTEGER I, M
I = 0
M = 1
\end{verbatim}

- integers have a sign \textit{(signed)}:
  \begin{itemize}
  \item J = -1000
  \item K = +200
  \end{itemize}
  \textit{the + is optional}

- if not declared otherwise (see below) each \texttt{INTEGER} occupies 32 Bit
  (=4 Byte) in the memory

- therefore: largest integer (32 Bit, two’s complement):
  \[2^{31} - 1 = 2147483647\]
  \textit{(smallest: \(-(2^{31})\) )}
INTEGER of different size than 32 Bit with help of the KIND parameter, e.g.,

```
INTEGER(KIND=8) trillion
INTEGER*1 c128
```

- KIND means usually the size in Byte (\(= 8\) Bit), so KIND=8 corresponds to 64 Bit
- the default value (if KIND is not used) is KIND=4 (32 Bit)
- all INTEGER of a source code can be set automatically to 64 Bit with the compiler option:

```
gfortran:  -fdefault-integer-8
ifort:  -i8
```
Use of integers:

- for loops: \( \text{DO } I = 1, 10 \)

- indices (subscripts) of arrays: \( A(I) \)

On the importance of integer arithmetic:

- Usually, integers can be faster processed than floats, as there is, e.g., no normalization necessary. Also, operations with integers often require a smaller number of bits.

- Bit patterns of integers are stable. Integer can be represented exactly and can be compared unambiguously.

- With help of integers it is possible to implement a fixed point number arithmetic without round-off errors (e.g., GnuCash).
Floating point numbers

- Floating point numbers are an *approximative* representation of real numbers.
- Floating point numbers can be declared explicit via

```
REAL radius, pi, euler, x, y
DOUBLE PRECISION rbb, z
```

(analogously to `float` or `double` in C).
- valid assignments are

```
x = 3.0
y = 1.1E-3
z = 4.0D-266
rbb = 2.06798E-300_8
```

The last two assignments refer to `DOUBLE PRECISION`: D instead of E or subsequent _8
Floating point numbers II

- Representation and usage of floating point numbers is, e.g., regulated by the standard IEEE 754.

\[ x = s \cdot m \cdot b^e \]  

(562)

where \( b = 2 \) is the basis (IBM Power6: also \( b = 10 \))

For 32 Bit (single precision):

bits

\[
\begin{array}{cccccccccc}
\end{array}
\]

exponent | mantissa

therefore for single precision (32 Bit):

\[-126 \leq e \leq 127 \rightarrow \text{max. } \approx 10^{38} (= 10^{127 \times \log 2}) \]

decimals: 7-8 (\(= \log 2^{23} \))
Floating point numbers III

- analogously for 64 Bit – DOUBLE PRECISION or KIND=8:
  
  exponent: 11 Bit \((r)\), mantissa: 52 Bit
  
  \(-1022 \leq e \leq 1023 \rightarrow \text{max.} \approx 10^{308} (= 10^{1023*\log 2})
  
  decimals: 15-16 (= \log 2^{52})

For the representation:

- sign bit: positive = 0, negative = 1

- exponent \(e\) results from non negative number \(E\) via \(e = E - B\) where
  
  \(B = 2^{r-1} - 1\), e.g., \(B = 2^{11-1} - 1 = 1023\)

- mantissa is set by *normalization* to the format (example)
  
  \(1.0100100 \times 2^4\)

  i.e. with 1 before the point. This 1 is not saved.
Floating point numbers IV

Notes about floats (IEEE 754):

- precise saving of integers with 6 up to 9 digits for 32 Bit floats, etc.
- there are positive and negative infinities, e.g., $-1./0. = -\text{Infinity}$
- there exist two zeros: $-0$ and $+0$, equal for comparison, but, e.g., $1/\pm 0 = \pm \infty$ (different for comparison)
- subnormal (denormal) numbers: fill underflow gap around zero by allowing for leading zeros in mantissa (hence larger exponent)
- NaN: not a number, result of, e.g., $\infty \times 0$, $0/0$, $\sqrt{-1}$, indicated by all exponent bits set to 1 and some non-zero number in mantissa

**Important:** NaNs can propagate through calculations (NaN +1 = NaN, but NaN$^0 = 1$)

<table>
<thead>
<tr>
<th>value</th>
<th>32 bit pattern (sign, exponent, mantissa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>0 000000000 0000000000000000000000000000000000</td>
</tr>
<tr>
<td>-0.</td>
<td>1 000000000 0000000000000000000000000000000000</td>
</tr>
<tr>
<td>inf</td>
<td>0 11111111 0000000000000000000000000000000000</td>
</tr>
<tr>
<td>-inf</td>
<td>1 11111111 0000000000000000000000000000000000</td>
</tr>
<tr>
<td>NaN</td>
<td>1 11111111 1000000000000000000000000000000000</td>
</tr>
</tbody>
</table>
On the KINDs of floats:

- default for REAL: KIND=4 corresponds to 32 Bit
- default for DOUBLE PRECISION: KIND=8 corresponds to 64 Bit
- all REAL of a source code can be automatically set to 64 Bit by the compiler option:

  gfortran:  -fdefault-real-8  
  ifort:    -r8

this option is usually used together with the analogous integer option (-i8).

Caution: By this all DOUBLE PRECISIONs are set to 16 Byte (=128 Bit)!
Trade-off between speed and accuracy

Often, accuracy and speed are conflictive goals. E.g., the use of 128 Bit floats slows down the code significantly, while switching back from 64 Bit to 32 Bit especially for vector operations (SIMD\(^a\), SSE\(^b\), AVX\(^c\)) can increase the computation speed.

\(^a\)Single Instruction Multiple Data
\(^b\)SIMD Streaming Extension, 128 Bit
\(^c\)Advanced Vector Extensions, 256 Bit
Importance of floating point numbers:
- essential for most scientific calculations, often 64 Bit precision is required

Problems:
- not all rational numbers can be represented exactly (as for, e.g., 1/3 in decimal system)
- bit patterns not “stable”, e.g. because of denormalization for small numbers
- accuracy of representation depends on the KIND, e.g. KIND=10 for FPU (floating point unit), but only KIND=8 when saved and for SSE instructions

Warning
Under no circumstances, it is OK to test blindly two floats on identity, instead one should always use a range of accuracy:
ABS( X - Y ) .LE. EPS instead of X .EQ. Y
COMPLEX Z, C

- intrinsic data type(!), always occupies two REAL memory cells
- access and assignment of complex variables:

```plaintext
REAL REALPART, IMAGINARYPART
COMPLEX Z
Z = ( 2.0 , 1.0 )
Z = CMPLX( 2.0 , 1.0 )
REALPART = REAL( Z )
IMAGINARYPART = AIMAG( Z )
```
Trick: The access on the components of complex variables is also possible with help of EQUIVALENCE:

```fortran
COMPLEX Z
REAL ZR(2), REALPART, IMAGINARYPART
EQUIVALENCE (ZR,Z)
...
REALPART = ZR(1)
IMAGINARYPART = ZR(2)
```
Logical - Boolean variable I

LOGICAL BTEST

- possible values: .TRUE. or .FALSE.
- example: BTEST = .FALSE.
- although only 1 Bit required, 32 Bit are used by default for storage
- default value of most compilers if not initialized: .FALSE.
- recommendation: LOGICAL variable names should start with a B (for boolean)
Computing with LOGICAL

- Variables of type LOGICAL can be combined such that the result is again of type LOGICAL (take care of the ordering, if necessary use parentheses):

\[
\begin{align*}
BBOTH &= \text{BONE .AND. BTWO} \\
BALSO &= ( \text{.NOT. BONE } ) \text{ .OR. BTWO}
\end{align*}
\]

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>logical AND ( (\wedge) )</td>
</tr>
<tr>
<td>.OR.</td>
<td>logical OR ( (\vee) )</td>
</tr>
<tr>
<td>.NOT.</td>
<td>logical NOT ( (\bar{B}) )</td>
</tr>
<tr>
<td>.XOR., .NEQV.</td>
<td>exclusive OR ( (\dot{\vee}) )</td>
</tr>
<tr>
<td>.EQV.</td>
<td>logical EQUIVALENCE (true if both true or both false)</td>
</tr>
</tbody>
</table>
Logical - Boolean variable III

- LOGICALS are also the result of comparisons:
  
  \[
  \text{BTEST} = \text{X} \cdot \text{GT. Y} \\
  \text{BCHECK} = \text{.NOT. ( I .EQ. J )}
  \]

<table>
<thead>
<tr>
<th>Fortran</th>
<th>meaning</th>
<th>C/C++/Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT.</td>
<td>strictly lesser than</td>
<td>&lt;</td>
</tr>
<tr>
<td>.LE.</td>
<td>lesser/equal</td>
<td>&lt;=</td>
</tr>
<tr>
<td>.EQ.</td>
<td>equal</td>
<td>==</td>
</tr>
<tr>
<td>.NE.</td>
<td>not equal</td>
<td>/= (in C: !=)</td>
</tr>
<tr>
<td>.GE.</td>
<td>greater/equal</td>
<td>&gt;=</td>
</tr>
<tr>
<td>.GT.</td>
<td>strictly greater than</td>
<td>&gt;</td>
</tr>
</tbody>
</table>
CHARACTER A, LINE*80

- Variables of type CHARACTER contain a single ASCII character (default), therefore KIND=1 (8 Bit)
- With the following modifier it is possible to create strings, i.e. “character arrays” (here: 80 characters long):

  ```
  CHARACTER LINE*80
  CHARACTER*80 LINE2
  CHARACTER (LEN=80) CARD
  CHARACTER(80) TEXT
  ```

- assignment to CHARACTER constants via quotation marks:

  ```
  LINE="Hello world!"
  TEXT=’Good bye!’
  ```
The access on substrings is possible with help of round parentheses ():

```
TEXT = 'Good bye!'
WRITE (*,*) TEXT(:2), TEXT(3:6), TEXT(7:)
```

 importante for the comparison of test strings:
upper and lower case are distinguished,
trailing blanks are ignored

```
BTEXT = TEXT .EQ. 'Good bye!'
```
## Comparison: Intrinsic data types in C/C++ and Fortran

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>int</td>
<td>32 bit</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
<td>32 bit</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>double</td>
<td>64 bit</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>—</td>
<td>32 bit</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>CHARACTER*80</td>
<td>char[81]</td>
<td>Fortran string not null terminated</td>
</tr>
</tbody>
</table>
We already know:

- **automatic formatting:**
  
  
  \((*,*)\)

- **text formatting:**
  
  \((*,’(A)’), (*,’(A80)’)\)

- **fixed point formatting:**
  
  \((*,’(F20.0, F8.2)’)\)

- **exponential formatting:**
  
  \((*,’(E5.2, E10.1E2)’)\)
FORMAT statement: Integer, Logical, and Complex

INTEGER
- formatting via ‘(In)’, \( n = \) number of digits, e.g., I3

LOGICAL
- format indicator: L or Lw
  e.g., WRITE (*,’(L)’) BTEST
  or WRITE (*,’(L3)’) BCHECK
- output: T or F (depends on value), or \( \Delta \Delta T \) (if ‘(L3)’)
- for input: only the first (non-blank) character is required: .T, T, .t, t
  or .F, F, .f, f
  (the format with the dot should be avoided in the shell)

COMPLEX
- output as for REAL, components separated by line break, e.g., for
  \( z = 1 + 2i \):

  PRINT *, Z (1.000000,2.000000)
  PRINT ’(F4.1)’, Z 1.0
  2.0
More output formatting:
- formatting of arbitrary number data types via G, analogously to E: ’(G10.2E3)’
- output of bit patterns: e.g., ’(B64)’ for 64 Bit

Format modifiers:
- repetition by preceding number of repetitions, hence: ’(4F5.2)’
- blanks via X, e.g.: ’(5X,A,X)’
- scaling – moving the decimal point per P: ’(2PE8.2E1)’ moves the decimal point by two digits to the right
- no line break via $: ’(A,$)’
There is i.e. no type checking (at compile time) in Fortran and data types are often not automatically converted. Therefore, this must be done by the developer, e.g., the conversion from INTEGER to REAL.

```fortran
X = REAL(N) ! argument can be INTEGER, REAL, COMPLEX
Y = FLOAT(M) ! argument is INTEGER
```

Moreover, we have already learnt a method to convert even CHARACTER (strings) to INTEGER or REAL:

```fortran
READ (*,'(F20.0)') X
```

reads from STDIN ("string") and converts to REAL

Hint: the asterisk * can be replaced by a variable name:

```fortran
READ (LINE,'(I10)') M
```
The other way around (INTEGER, REAL $\rightarrow$ CHARACTER) is done by WRITE.

```fortran
WRITE (LINE,'(I5)') NUMBER
```

Already known.
The one-liner:

**IF (EXPRESSION) instruction**

- **EXPRESSION** is a logical expression (LOGICAL) with the values `.TRUE.` or `.FALSE.`.
- Often: comparison (operators) and combinations (sentential connectives), e.g.,

  ```
  IF ( XL .GT. 1. .AND. XL .LT. 200. ) GOTO 100
  ```
The extensive standard form:

```plaintext
IF (EXPR) THEN
  ... instruction ...
ELSE IF (EXPR2) THEN
  ... other instruction ...
ELSE
  ... alternative ...
ENDIF
```
Instead of lengthy IF-THEN-ELSEIF blocks it may be more convenient to use:

```
[name:] SELECT CASE (variable)
    CASE (value1)
        ...
    CASE (value2)
        ...
    CASE DEFAULT
        ...
END SELECT name
```

where variable must be of type
- INTEGER
- LOGICAL
- CHARACTER

and value can also be a range:
- min:max
- min:
- :max
Loops are among the most powerful and useful structures in programs. In Fortran loops begin with the statement:

```
DO
```

In principle there are the following types of loops:

- "count" loops
- WHILE loops
- "infinite" loops
infinite loops have an empty loop header (no break condition):

```
DO
...
instruction block ...
END DO
```

Leaving the loop is therefore only possible by an instruction like
IF (BEND) EXIT

EXIT

immediately terminates the current iteration and exits the current(!) loop
Do forever II

Example

```
DO
    K = K + 1
    IF ( K .EQ. 10 ) EXIT
ENDDO
```

We remember:

Blanks are optional in Fortran. So they are also optional in END DO, DOUBLE PRECISION, ELSE IF, GO TO, …
When entering the count loop the loop header is evaluated:

```
DO K = 1 , NMAX , 1
   ...
ENDDO
```

- the first argument initializes the loop counter (always!)
- the second argument defines the maximum value of the loop variable

Note: If the loop is exited normally, the loop variable \( K \) has the value \( NMAX + 1 \).

- the third (optional) argument is the increment (or decrement)
Count loops II

Fortran 90:
- loop variables can be also of type REAL.

Fortran 95:
- loop variables can only be of type INTEGER.

General loop control
- EXIT terminates the current(!) loop immediately
- CYCLE skips the rest of the current iteration(!) and jumps to the beginning of the next iteration
- CONTINUE does nothing, useful for loops with labels

Loop with label (there was no ENDDO in FORTRAN77):

\[
\text{DO 101 , K = 3 , 10} \\
\text{...} \\
\text{101 CONTINUE}
\]
The **WHILE** loop checks *before* each iteration for a logical expression:

```
WHILE
    DO WHILE (condition)
        ...
    END DO
```

- **bad**: the condition is changed *in* the loop body, but the condition is checked not before the next iteration
- because of performance issues it is recommended to avoid (extensive) **WHILE** loops
### Comparison: Execution in C/C++ and Fortran

#### Branching

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (X .GT. Y) THEN</td>
<td>if (x &gt; y) {</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>END IF</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

#### For loop

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO I = 1, 10</td>
<td>for (int i = 0; i &lt; 10; ++i)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>END DO</td>
<td>}</td>
</tr>
</tbody>
</table>
Open a file

OPEN (UNIT, FILE="name", ACTION="access", ERR=label)

- opens a file with name “name” (similar syntax as in IDL),
- assigns a device number UNIT of type INTEGER to the file; the only obligatory argument,
- defines the way of access: READ, WRITE, or READWRITE (default),
- and jumps in the case of an error during opening to the label label (e.g., file not found).

Example

OPEN (22, FILE='output.dat', ERR=999)

→ opens the file output.dat, jumps to label 999 in the case of an error
The counterpart:

**Close a file**

\[ \text{CLOSE (UNIT,ERR=label)} \]

where \( \text{ERR=label} \) is optional

**Example**

\[ \text{CLOSE (22)} \]

Note: For performance issues it may happen that write actions on the file will be postponed until the \text{CLOSE} statement (buffering).
Read from an opened file per line

```fortran
CHARACTER line*132
READ (22,'(A132)') line
```

→ read in a text string

or formatted reading into corresponding variables:

```fortran
READ (22,'(I10,1X,F10.2)', END=21) n , X
```

→ if the last line of the file is reached: jump to label 21

Or unformatted reading:

```fortran
READ (22,*,END=11) n , X
```
Write to an opened file, e.g.,

\[
\text{WRITE (42,*) "file header:"}
\]

Or formatted:

\[
\text{WRITE (42,'(A,G12.3)') " x = " , x}
\]
Programming in Fortran - Part 3

Arrays
There are many ways to allocate arrays, i.e., to reserve memory. Most efficient and stable, the *static* allocation (on stack):

- Size and shape of the array must be determined at/before compile time in the program (but shape can be changed in procedures)
- The process of allocation is done when the program is loaded.
- Allocated memory is only freed at the end of the program.

Recommended and unambiguous is the use of

\[ \text{DIMENSION} \text{ array (array specification)} \]

This instruction can stand alone or as a so-called attribute.
Arrays II

Example: allocation of a “vector”

```plaintext
DIMENSION POSITION(3)
```

- The name of the array, POSITION, implies the data type REAL.
- The array consists of 3 elements and has the dimension (rank) 1.

Example: allocation of a “matrix”

```plaintext
DIMENSION ROTATIONALMATRIX(3,3)
```

- The array has the dimension (rank) 2, the shape (3,3) and the size $3 \times 3 = 9$. 
DIMENSION can also be used as an attribute:

```
REAL(8), DIMENSION(3) :: x, v
```

Data type and attribute(s) are separated from the list of variables by double colons.

Moreover:

- The argument of DIMENSION must be a (integer) constant, e.g.,

```
INTEGER, PARAMETER :: NMAX=100
REAL, DIMENSION(NMAX,NMAX) :: NLINE
```

- The maximum dimension is 7. E.g.,

```
INTEGER, DIMENSION (1,2,3,4,5,6,7) :: rank7
```
Arrays IV

DIMENSION M(100,100)

- arrays can be accessed as whole or per element:

  M = 0
  M(1,1) = 1

- arrays can also be accessed with help of ranges:

  M(2:100,100) = 1

- the use of ranges (bounds) offers the interesting possibility to set negative indices:

  DIMENSION M(-50:49,100) = 1
Since Fortran 90 exists the possibility to set the size (but not the rank) of an array at runtime. Such arrays must be marked as allocatable:

**Dynamic Allocation**

```
INTEGER, ALLOCATABLE, DIMENSION(:, :) :: M
...
READ (*,'(I1000)') I
ALLOCATE (M(I,I),STAT=ERR)
```

The `ALLOCATE` statement reserves the corresponding amount of memory (dynamic memory, heap), but does not initialize it. If a STAT variable is given, this variable is set to 0 in case of successful allocation. If no STAT variable is given, the program stops in case of error.

**Deallocation – Freeing dynamically allocated arrays**

```
DEALLOCATE (M,STAT=ERR)
```
Functions are a type of *procedures* in Fortran. Functions always return a result of a distinct data type:

**FUNCTION call**

```
Result = NAME (arg1, ... , argN)
```

The definition of a the function is either after the `END` statement of the `PROGRAM` or in a separate source file:

**FUNCTION definition**

```
FUNCTION NAME (arg1, ... , argN)

...  

NAME = ...

END
```

Usually the result (returned value) is assigned to the variable of the same name as the function within the function body.
A simple example:

```fortran
PROGRAM cylinder_volume
IMPLICIT NONE
REAL r, h, volume
READ (*,*) r, h
WRITE (*,*) volume(r,h)
STOP
END

FUNCTION volume(radius,height)
IMPLICIT NONE
REAL radius, height, volume
volume = 3.141 * radius**2 * height
END
```

The type of the result must be declared if IMPLICIT NONE is used. Arguments are identified via their order in the list of arguments.
Attributes for the declaration of functions:

- Fortran compilers know different builtin (*intrinsic*) functions (e.g. \( \text{SIN}(X) \)). If these intrinsics shall be overwritten (defining functions of the same name), the function must be declared in the calling procedure as `EXTERNAL`:

\[
\text{REAL, EXTERNAL :: COS}
\]

- Functions can be *directly* called recursively. In this case a different name for the result must be declared:

\[
\text{RECURSIVE FUNCTION FAC(N) RESULT(L)}
\]

\[
\ldots
\]

\[
L = \text{FAC}(N-3)
\]
The type of the result can also be given in the function declaration instead of the declaration block:

```fortran
REAL FUNCTION volume(radius, height)
IMPLICIT NONE
REAL radius, height
```

The arguments of functions are passed by address. Hence they can be manipulated within the function!

```fortran
FUNCTION volume(r,h)
  ...
  r = 42.
```

Different from passing arguments by value in C/C++ the altered argument is also changed in the calling procedure.
In Fortran the calling and called procedure work on the same memory area, which is specified in the list of arguments.

To avoid side effects like this, a function can be declared as `PURE` or `ELEMENTAL`, this requires a declaration of arguments to be `INTENT(IN)`:

```
PURE REAL FUNCTION volumen(r,h)
  INTENT(IN) :: r, h
  ...
  r = 42. ! forbidden
```
In addition to the FUNCTION the SUBROUTINE is another important procedure. It is some kind of a “subprogram” and has, different from functions, no result (\(\hat{=}\) void in C/C++).

SUBROUTINE call

\[
\text{CALL rname (arg1, ... , argN)}
\]

Like for the FUNCTION, the SUBROUTINE is defined outside the calling procedure.

SUBROUTINE definition

\[
\text{SUBROUTINE rnam (arg1,...,argN)}
\]
\[
\ldots
\]
\[
\text{END}
\]

The SUBROUTINE and the calling procedure share the same memory area, which is defined by the list of arguments.
Although the SUBROUTINE has no result, it is possible to distinguish between input and output arguments:

```fortran
SUBROUTINE rname (arg1,arg2,arg3,arg4)
  INTEGER, INTENT(IN) :: arg3, arg4
  REAL, INTENT(OUT) :: arg1
  INTEGER, INTENT(INOUT) :: arg2
  ...
END SUBROUTINE
```

Arguments that are declared as INTENT(IN) cannot be changed within the subroutine. INTENT(OUT) arguments shouldn’t be defined when entering the subroutine. These restrictions do not apply to variables that are INTENT(INOUT).
The attribute PURE

```
PURE FUNCTION myfunc(...)  
PURE SUBROUTINE mysubr(...)  
```

allows to use procedures without side effects. Only arguments marked as INTENT(OUT) can be changed; arguments of PURE FUNCTIONs cannot be changed.

Intrinsic functions, e.g., ABS() are always PURE.
Note: While FUNCTIONS are rather used for short procedures, SUBROUTINES often contain complex subprograms.

- Both types of procedures can also be called via an ENTRY point:

```fortran
ENTRY
  SUBROUTINE tue123 (arg1, arg2)
  ...
  ENTRY tue23 (arg2)
ENTRY statements cannot stay in IF or DO blocks.
```

Style

Avoid ENTRY points as they reduce the readability of the source code.
Arguments of procedures:

- literal constants:

  \[
  \text{Result} = \text{MYFUNC}(3.0, \ .\text{FALSE}.)
  \]

- scalar variables: \( z = x \times \text{COS}(\alpha) \)

- procedures, especially functions (later more . . .)

- arrays: as a whole, single elements or ranges

  \[
  \text{CALL MYROUTINE ( A, B(1,1), C(1:2) )}
  \]

Arrays must be dimensioned (again) in the called procedure:

\[
\text{DIMENSION} \ A(3,2), \ C(2)
\]
especially for CHARACTER arrays the asterisk is recommended (assumed length):

```
CHARACTER(LEN=*) TEXT
```
The program and other procedures can be assembled in a single source file.

- This is recommended for short programs and functions (e.g., many of our exercises).

For more complex programs:

- one procedure per file, which has the same name as the procedure it contains
- compiling and linking:

```
gfortran -c myfunc.f
```

```
gfortran -o myprogram.exe myprogram.f myfunc.o
```

- option -c generates a (not linked) object file myfunc.o.
- object files can be assembled in libraries:
Recommendations for the handling of procedures II

The archive tools ar & ranlib

```
ar rv libmylibrary.a myfunc.o
ranlib libmylibrary.a
```

- `r` replace or append a file;
  if archive does not exist, it is created
- `v` verbose

```
rsv
```

```
ranlib renew the “table of contents”
```

libraries which have been created in this way can be linked in at compile time:

```
gfortran -L$MY_PATH -lmylibrary myprogram.f
```

- `-L` specifies the path for the library, `-l` the name of the library, which always starts with `lib`. This prefix is omitted for the call.
Names

There are the following restrictions in Fortran for names of variables and procedures:

- the name must start with an ASCII-letter (ifort: also $)
- the name must not be longer than 31 characters (ifort: 63)
- except for letters also allowed are: numbers, underscores _, and the dollar symbol $ (gfortran: -fdollar-ok required)

Supported names

- Radius123
- Begin$_

Invalid names

- 3pel ! does not start with a letter or $
- boundary-condition.2 ! invalid characters - .
Supported characters

Fortran supports the ASCII character set:

- letters (uppercase/lowercase), numbers
- + the usual special characters:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank/TAB</td>
<td>colon</td>
</tr>
<tr>
<td>=</td>
<td>equality sign</td>
</tr>
<tr>
<td>+</td>
<td>exclamation mark</td>
</tr>
<tr>
<td>-</td>
<td>double quotation mark</td>
</tr>
<tr>
<td>*</td>
<td>per cent</td>
</tr>
<tr>
<td>/</td>
<td>ampersand</td>
</tr>
<tr>
<td>( )</td>
<td>semicolon</td>
</tr>
<tr>
<td>.</td>
<td>lesser than</td>
</tr>
<tr>
<td>,</td>
<td>greater than</td>
</tr>
<tr>
<td>'</td>
<td>question mark</td>
</tr>
<tr>
<td>'</td>
<td>Dollar symbol</td>
</tr>
</tbody>
</table>

+ some symbols which are only printable (e.g., curly braces)
Source code formatting

- Fortran 90/95 allows up to 132 characters per line (arbitrary in free form)
- continuation lines: up to 19 (ifort: 511)

<table>
<thead>
<tr>
<th>gfortran</th>
<th>option</th>
<th>usual arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-ffixed-line-length-n</td>
<td>0, 72 (default), 80, 132, none (=0)</td>
</tr>
<tr>
<td></td>
<td>-ffree-line-length-n</td>
<td>0, 132 (default), none (=0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ifort</th>
<th>option</th>
<th>usual arguments</th>
<th>alt. options</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-extend-source [n]</td>
<td>72, 80, 132 (default)</td>
<td>-72, -80, -132</td>
</tr>
<tr>
<td></td>
<td>-free</td>
<td>- (no line limit)</td>
<td></td>
</tr>
</tbody>
</table>
Some intrinsic functions and subroutines

- random numbers

\[
\text{result} = \text{RAND}(I)
\]

returns uniformly distributed random numbers of a sequence \((I = 0)\);
Restart if argument different from 0

- date and time

\[
\begin{align*}
\text{CALL ITIME (ITARRAY)} \\
\text{CALL IDATE (IDARRAY)}
\end{align*}
\]

fills \(\text{ITARRAY}(3)\) with hour, minute, second;
fills \(\text{IDARRAY}(3)\) with day, month, year
representable numbers

Result = HUGE(I) ! also for integers
Result = TINY(X)
Result = EPSILON(X)

returns the largest or smallest representable number, respectively, of the same type as the argument X. EPSILON gives the smallest representable number $E$, such that $1 + E > 1$ (minimal step width for floats).

command line arguments for program call

Result = IARGC()
CALL GETARG(N,TEXT)

returns the number of command line arguments; fills the CHARACTER variable TEXT with the $N$th command line argument
CALL SYSTEM (TEXT): starts a *shell* in which the command(s) stored in TEXT are executed, e.g.,

```
CALL SYSTEM('cp scratch.ps wrplot1.ps')
```

Note that one cannot read from STDOUT with SYSTEM, i.e. CALL SYSTEM (’ls’) is useless

- LEN (TEXT) returns the length of the string TEXT
- TRIM (TEXT) removes trailing blanks from string TEXT
The graphical library PGPlot can be called from C or Fortran programs. This must be declared at link time:

**Linker call**

```
-lX11 -L$PGPLOT_DIR -lpgplot
```

where the environment variable $PGPLOT_DIR must contain the (absolute) path
The following environment variables should therefore be set:

**PGPLOT environment (bash)**

```
export PGPLOT_DIR=${HOME}/PGPLOT/
export LD_LIBRARY_PATH=${PGPLOT_DIR}
export PATH=${PATH}:${PGPLOT_DIR}
export PGPLOT_DEV="/XWIN"
```
The most important procedures of PGPLOT:

- **PGBEG**

  
  \[
  \text{IER} = \text{PGBEG}(0, '/XWIN', 1, 1) \\
  \text{IF (IER .NE. 1) STOP 'PGPLOT failed'}
  \]

  a function(!) that starts PGPLOT (server). The second argument is for the output device; this can be, e.g., '/XWIN', also '?' which implies that the user is asked for a device at run time. In the case of success the function returns '1'.

- **PGENV**

  \[
  \text{CALL PG ENV (XMIN, XMAX, YMIN, YMAX, JUST, AXIS)}
  \]

  defines the drawing area; for JUST = 1 the x- and y-axis are scaled equally; AXIS=1 also plots the coordinate axes and labels
- **PGLAB**

  \[
  \text{CALL PGLAB('x','y','title')}
  \]

  defines the labels of the x- and y-axis and of the plot

- **PGPT**

  \[
  \text{CALL PGPT(1,X,Y,-1)}
  \]

  draws a single point (1st argument) with given X- and Y-coordinate with the size given in pixel (last argument)

- **PGEND**

  \[
  \text{CALL PGEND}
  \]

  Closes the plot correctly.
PGPLOT IV

- **PGSCI**

  \[\text{CALL PGSCI(INDEX)}\]

  Set color index for the following drawing actions.

- **PGSCR**

  \[\text{CALL PGSCR(INDEX,R,G,B)}\]

  Set the color for the given color index:
  INDEX=0 is background, INDEX=1 the foreground (default: white on black). \(R, G, B\) are the color fractions red, green, blue, each in the range 0 to 1.
Programming in Fortran - Part 4

Variables and passing variables
We already know:

- functions and subroutines get a list of arguments (usually names of variables) when called
  
  \[
  \text{CALL MY\_SUB (arg1, arg2)}
  \]

- internally: only start addresses are passed, i.e. correct dimensioning in calling and called procedures necessary:
  
  \[
  \text{SUBROUTINE calcE (velo, pos)}
  \]
  
  REAL :: velo(2), pos(2)
the list of arguments for a declared/defined procedure and for its call must (normally) be matching

    SUBROUTINE my_sub (arg1, arg2, arg3)
    ...
    CALL my_sub (x, y, z)

however, the last argument(s) can be omitted (not checked by compiler):

    SUBROUTINE my_sub (arg1, arg2, arg3)
    ...
    CALL my_sub (x)
→not recommended, bad style!
there is (usually) no type checking (exception: intrinsics, procedures in same source file) and no automatic type conversion, e.g.,

\[ \text{CALL PGPT(1.,0_D,0,1.)} \]

might not give the expected result …

as a preceding declaration of procedures is not necessary (cf. include files in C/C+) the compiler can usually not check the matching of argument lists
Solution:

- To make the argument list verifiable by the compiler an explicit INTERFACE block can be included in the calling procedure.
- This contains exact copies of the declaration block of the called procedure, including the header, declaration of arguments, and the END statement.
INTERFACE

... INTERFACE

SUBROUTINE my_sub(arg1, arg2)
    REAL :: arg1
    INTEGER :: arg2
END SUBROUTINE my_sub

FUNCTION my_func(N)
    INTEGER :: N
    RETURN
END FUNCTION my_func

END INTERFACE

...
Rules for (explicit) INTERFACE blocks:

- the INTERFACE block is part of the declaration section must therefore appear *before* any instruction
- each procedure can only have one interface per calling procedure
- an explicit interface is required, if one or more arguments have the following attributes:
  - ALLOCATABLE
  - OPTIONAL (see below)
  - reference: POINTER
  - object attribute: TARGET, VOLATILE
- an explicit interface is also required for functions those result is an array or a pointer
The OPTIONAL attribute

- arguments which are marked with the OPTIONAL attribute can be omitted for the call of the procedure, if they:
  - are the last ones in the list of arguments
  - or all following arguments of this list are passed by a keyword

```
INTERFACE
  SUBROUTINE calcT (a, x, h)
    OPTIONAL x, h
  END
END INTERFACE

CALL calcT(a, h=r)
```

- the *keyword* must be identical to the name of the dummy argument in the interface
The PRESENT function:

- the presence of *optional* arguments while passing them can be determined with help of the function

\[
\text{PRESENT}(\text{arg})
\]

E.g.,

```fortran
SUBROUTINE calcE (velo, pos)
    REAL, OPTIONAL, DIMENSION(2) :: pos
    ...
    IF ( PRESENT(pos) ) THEN
        r = SQRT( pos(1) * pos(1) + pos(2) * pos(2) )
        Eg = - GM / r
    ENDIF
ENDSUBROUTINE
```

obvious: PRESENT has result of type LOGICAL
Volatility of variables:

```fortran
SUBROUTINE output(pos,t,fpos)
  REAL :: pos(2), fpos(2), t
  LOGICAL bset
  ...
  bset = .TRUE.
```

the used LOGICAL is a *local* variable, which is by default *automatic*, i.e. it is removed from memory when leaving the procedure. → when re-entering the procedure output the value of bset is (usually) deleted
Solution:

Variables can be made non-automatic by the SAVE attribute or by the SAVE declaration statement (corresponds to static in C/C++). Then, the content of the variable is saved for the re-entry of the procedure.

E.g.

```fortran
SUBROUTINE output(pos,t,fpos)
    REAL :: pos(2), fpos(2), t
    LOGICAL, SAVE :: bset
    SAVE fpos
```
Moreover, the compiler can convert any automatic (local) variable to a static (SAVE) variable (different from C++):

- gfortran: -fno-automatic
- ifort: -save

But, the use of local variables without SAVE attribute for buffering may cause segmentations faults which are hard to track (randomness).
Fortran procedures usually communicate by passing variables as arguments at call.

Alternatively, a common memory area can be created, the so-called COMMON block(s), corresponding to global variables in C/C++:

```
COMMON /velpar/ vfinal, vpar2
COMMON // buffer(1000)
```

where the name of the block (in //) is optional → blank Common block.

All procedures that declare a common block with its variables have shared(!) access to the corresponding memory area.
the partitioning of the shared memory does not need to be the same for all procedures, e.g.,

```
COMMON /coords/ x, y, z, i(10)
```

and an alternative declaration

```
COMMON /coords/ r, p, k(11)
```

where i(1) then is the same as k(2) and so on.

The use of COMMON blocks is generally not recommended.
Programming in Fortran - Part 5

Overloading, Modules
Overloading

- if the interface is given a name, the corresponding procedure can be overloaded, i.e. the compiler chooses the matching procedure by the type of arguments:

```fortran
INTERFACE mysub
   SUBROUTINE mysub1 (n,...)
      INTEGER :: n
   END SUBROUTINE mysub1
   SUBROUTINE mysub2 (x,...)
      REAL :: x
   END SUBROUTINE mysub2
END INTERFACE
```

- and call via

```fortran
CALL mysub(arg,...)
```
Overloading/extending operators

- an interface can also overwrite an operator:

\[
\text{INTERFACE OPERATOR}(\text{op})
\]
\[
\text{FUNCTION myop (arg1, arg2)}
\]
\[
\text{type, INTENT(IN) :: arg1, arg2}
\]
\[
\text{END INTERFACE}
\]

- in this case only functions with one (unary) or two (binary) non optional arguments with INTENT(IN) can appear

- op is, e.g., + or - or .myop.

- if an intrinsic operator (e.g., .LE.) is extended, also the alternative notation (here: <=) is affected and the number of arguments must be the same as for the intrinsic form
Extending assignments:

- there is only one assignment operator in Fortran: =
- this operator can only be extended by a SUBROUTINE with two arguments in the following way:

```fortran
INTERFACE ASSIGNMENT(=)
    SUBROUTINE myassign (arg1,arg2)
        type, INTENT(OUT) :: arg1
        type, INTENT(IN) :: arg2
    END SUBROUTINE myassign
END INTERFACE
```

- e.g., \( x = n \rightarrow \text{type casting} \)
Unambiguousness:

- When overloading it must be clear from the list of arguments (type, number of arguments, name) what to choose:

Wrong:

```fortran
INTERFACE f
  FUNCTION fxi (x,i)
    ...
  FUNCTION fix (i,x)
    ...
END INTERFACE f
```

- In the case of ambiguities while extending intrinsic procedures the non-intrinsic procedure is chosen.
**Header files**

- with help of an explicit INTERFACE the compiler gets informed about the data types of the called procedures in the calling procedure
- more convenient: put INTERFACE in an extra file, e.g., `file.h` and include it via

```
INCLUDE 'file.h'
```

- INCLUDE can appear anywhere in the source code
- INCLUDE requires as argument a text string which contains the name of a Fortran source file, which is included at this very position
- i.e. *header* files can be created similar to *C++*
Include a self-made header file:

Example

calcE.h :

INTERFACE

  SUBROUTINE calcE (vel, pos, E)
    REAL :: vel(2), pos(2), E
    END SUBROUTINE calcE

END INTERFACE

dkepler.f :

  PROGRAM kkepler
    INCLUDE 'calcE.h'
    ...

H. Todt (UP) Computational Astrophysics SoSe 2020 576 / 647
Better than separated files with interfaces and procedures:
→ modules

- contain variables and procedures, which can be used by other procedures
- procedures that shall access a module must import this module through

```
USE modulename
```

Structure of a module
```
MODULE name
  ... declarationes ...
CONTAINS
  ... procedures ...
END MODULE name
```
Example

MODULE energies
  REAL, PARAMETER :: PI = 3.141519
  CONTAINS
    SUBROUTINE calcE (E, vel, pos)
      ...
    END SUBROUTINE calcE
    FUNCTION Egrav (pos)
      ...
    RETURN
  END FUNCTION Egrav
END MODULE energies
CONTAINS

- The CONTAINS statement allows the definition of functions or subroutines within another procedure (e.g., in PROGRAM) and appears at the end of the definition of the procedure before END.

- The procedures that follow a CONTAINS statement are referred to as *internal subprograms* and must *not* contain another CONTAINS (no nesting).

- ENTRY points must appear before CONTAINS, internal subprograms must not contain an ENTRY point.

- Internal subprograms have access to all names declared in the *host* procedure (e.g., variables), the internal subprogram has an explicit interface.

- The host procedure can call the internal subprogram, as well as the internal subprogram can call itself.
Compiling modules:

- the module is compiled via
  ```
gfortran -c module.f
  ```
- the result are two(!) files:
  ```
  module.o  module.mod
  ```
- `.o` object file: the usual machine-readable instructions
- `mod` file: ASCII file with interface instructions for the compiler, i.e., the functionality is described that is provide to the calling procedure via
  ```
  USE module
  ```
  (explicit interface, see below)
- correspond to header files in `C++`
compiled modules are in general not compatible for different compilers, but must exist as source code and re-compiled for each compiler

as MODULEs are already a kind of an explicit interface, the procedures in them cannot be overloaded by a named interface (see above) → instead – preferably in the MODULE:

```
INTERFACE mysub
  MODULE PROCEDURE mysub1, mysub2
END INTERFACE mysub
```
Examples:

The Intel Fortran compiler offers a variety of procedures encapsulated in modules:

- **USE IFPORT**: e.g., `len = FULLPATHQQ (file,output)` returns the full path of a file;
  `CALL GETENV(variable,content)` returns the content of an environment variable → intrinsic in gfortran

- **USE IFPOSIX**: e.g., `CALL PXFMKDIR (name, len, mask, result)` makes a directory

- **USE IFCORE**: e.g., `bpressed = PEEKCHARQQ ()` detects, if a key is pressed (without pausing program)

- **USE OMP_LIB**: OpenMP library

- **Windows specific**: e.g., **USE IFLOGM** for dialog boxes, etc.
Defining structure, i.e. a collection of variables:

```
TYPE typename
    ... declaration of components ...
END TYPE typename
```

declares a (derived) data type with its components, e.g.,

```
TYPE star
    REAL :: radius, mass_i, mass_c
    CHARACTER :: spectraltype*2
END TYPE star
```
declaration of structures created by that:

```
TYPE (star) :: WR144
TYPE (star) :: sun = star (1.,1.,1.,’G2’)
```

access to the components possible via the % symbol (cf. dot . in C++):

```
xinitialmass = sun%mass_i
```

for TYPE data types an operation can be defined with help of INTERFACE OPERATOR (op)
e.g., addition component-by-component
Programming in Fortran - Part 6

Name spaces, scopes, pointer, C
We already know:

PROGRAM myexe
    INTEGER :: init, k
    init = 4 ; k = 2
    CALL mysub (k)
    ...
SUBROUTINE mysub (m)
    IMPLICIT NONE
    INTEGER :: m, j
    j = init ! does not work

Value of variable init in subroutine mysub not available, as the variable init neither

- has been passed by argument,
- was made visible globally by a COMMON block,
- nor is automatically globally visible (e.g., CONTAINS, MODULE)
Visibility of names II

In Fortran

- variables, hence “instances” of data types
- program units, like functions, subroutines, modules
- certain structures, e.g., named interfaces

are identified via a name by the compiler/linker

This name is only visible within a scoping unit:

- TYPE definition
- INTERFACE block
- program unit (e.g., SUBROUTINE, FUNCTION)

(except for scoping units that are contained in these units) and therefore needs to be unambiguous only within this unit:
Visibility of names III

Example for scopes

```plaintext
MODULE mod1 ! 1
  INTEGER hello1 ! 1
  CONTAINS ! 1
    SUBROUTINE sub2 ! 2
      TYPE mytype3 ! 3
        REAL :: r,t ! 3
        INTEGER hello1 ! 3 ok, as not in scope1
      END TYPE mytype3 ! 3
      ... ! 2
    END SUBROUTINE sub2 ! 2
  END MODULE mod1 ! 1
```

These declared names are only visible within the units where they are declared.
If more than one module is used:

```plaintext
USE std_lib
USE math_lib
```

the problem of *name clashes* can occur, i.e. identical names (variables, procedures) in both modules

Instead of parallel name spaces, as in C++, there are two methods to circumvent such problems:

**Method I:**

```plaintext
USE module, rename-list
```

where `rename-list` has the form:

```plaintext
USE module, name_in_module => aliasName
```

I.e. the name declared in the module is → replaced by another name

**Method II:** the export of names from a module can be restricted

```plaintext
USE std_module, ONLY: name1, name2
```
PRIVATE and PUBLIC
We already know:

```fortran
MODULE graph_ps
  TYPE point
    REAL :: x, y
  END TYPE point
  REAL :: scale = 400.0
  INTEGER :: graphics_unit = 20
END MODULE graph_ps
```

- procedures that use module `graph_ps` have access to `point`, `scale`, `graphics_unit`
- useful for usage of data type `point`
- sometimes it is required that names (e.g., variables like `graphics_unit`) are not visible from outside the module
for the *encapsulation* of data the attribute

```
PRIVATE :: var1, ...
REAL, PRIVATE :: x
```

can be used.
The opposite is

```
PUBLIC :: var1, ...
INTEGER, PUBLIC :: k
```

These attributes can, used as an instruction, also define a default.

```
MODULE mod1
PUBLIC
```
Example 1

```fortran
MODULE graph_ps
PUBLIC

  TYPE point
    PRIVATE
    REAL :: x
    REAL, PUBLIC :: y
  END TYPE

REAL, PRIVATE :: scale
```

- procedures that use this module have access to `point` and `point%y`, but *no access to* `scale` and `point%x`
- members (procedures) of this module have complete access to all variables
Example 2

MODULE example

    PRIVATE only_int, only_real

INTERFACE general
    MODULE PROCEDURE only_int, only_real
END INTERFACE general

CONTAINS
    SUBROUTINE only_int (i)
       ...
    SUBROUTINE only_real (x)
       ...
END MODULE bsp

- procedures that use example can only use the named interface general
• with help of PRIVATE instruction or attribute it is possible to hide *names* in a module from access by external program units

☛ cf. also (same effect)
USE mod1, ONLY name1

• in the same way the components of a TYPE definition can be hidden from access by external units, so that → only procedures of the MODULE have access

The consequent use of PRIVATE helps to ensure data integrity (cf. global vs. local variables).
Attributes

- for the declaration of data types (e.g., REAL) or functions (result) attributes (modifiers) can be given, e.g.:

**Attributes**

ALLOCATABLE, AUTOMATIC, DIMENSION, EXTERNAL, INTENT, OPTIONAL, PARAMETER, PRIVATE, PUBLIC, PURE, SAVE

**Example**

```
INTEGER, SAVE, DIMENSION(2,2) :: sigma
REAL, EXTERNAL :: COS
...
FUNCTION COS (X)
```
most attributes can also be given as an *instruction*, defining some default for the data types and procedures declared in a procedure

---

**Example**

```fortran
SUBROUTINE mysub (x, y, z)
    SAVE
    REAL :: x, y, z, r
    INTEGER :: k
```

→ all local variables are put to the static memory (instead of dynamic memory/heap), cf. compiler option `-fnoautomatic` (gfortran) or `-save` (ifort)
We already know:

Assignment: memory $\leftrightarrow$ variable

principle: compiler allocates (reserves) memory (accessible via memory address) for variables following a certain scheme, e.g., INTEGER occupies 4 byte (32 bit), beginning at the start address variable within the source code accessible via name

Pointer:

- stores addresses of something, e.g., variables, arrays, functions
- in e.g. C pointers are the only realization of a reference (regarding the call behavior)

Pointer in Fortran:

- only restricted pointer methods
- pointer are always associated with an “object”:
• either by *allocation* $\rightarrow$ ALLOCATE
• or with help of an *assignment* $\Rightarrow$ i.e. association with an already existing “object”
• no pointer arithmetics
REAL, POINTER :: p ! declared, but not allocated
! -> not "existing"
ALLOCATE (p) ! p allocated, from now on usable
p = 2.7182 ! access as for normal REAL variable
DEALLOCATE (p) ! free memory

not very useful example, clearer for arrays → only number of dimensions (shape) not size for declaration required:
REAL, DIMENSION (:,:,:), POINTER :: cube

...  
N = 42  
ALLOCATE (cube(N, N, N))

....

ALLOCATE (cube(1,2,3)) ! allocation w/o previous DEALLOCATE  
  ! possible only for POINTER
  ! note: entries are not kept

...  
DEALLOCATE (cube)
REAL, TARGET :: x, y ! attr. TARGET required
REAL, POINTER :: p, q
y = 0.75
p => x ! associates p with x
p = y ! normal assignment (values) of y to p
WRITE (*,*) x ! gives 0.75
q => p ! x, p, q are now the same

Note: For p => x the pointer p is assigned to the target x, while q => p lets the pointer q reference the same object as p

Meaning of the TARGET attribute:

To enable optimization the compiler needs to know which variables are referenced, i.e. whether their is an alternative method of access.
TARGET and the corresponding POINTER must be of the same type (REAL, INTEGER, ...), moreover, in the case of arrays they must agree shape (rank, dimension):

```
REAL, TARGET :: cube (16, 16, 1000)
REAL, POINTER :: image (:,:)
...
image => cube(:,:,42) ! Sub-Array
...
NULLIFY (image)
```

It is not necessary, but safer to disassociate a pointer after usage with help of NULLIFY or
image => NULL ()
Applications of pointers I

Typical application: linked lists

```fortran
TYPE entry
  REAL :: value
  INTEGER :: index
  TYPE (entry), POINTER :: next
END TYPE entry

TYPE (entry), POINTER :: first, current
! with first%index and first%next%index
ALLOCATE (first)
first = entry (1.,10,null())
ALLOCATE (current)
current = entry (2.,20, first)
! = means current%next => first
first => current
! first now points to new entry w/o deleting
! the first entry
```
So, from a list of two elements (each has three components):

\[
\begin{align*}
\text{first} & : (2.0, 20, \text{associated}) \\
\text{first}\%\text{next} & : (1.0, 10, \text{null})
\end{align*}
\]

it is possible to create a list of three with help of \texttt{current}:

\[
\begin{align*}
\text{allocate (current)} \\
\text{current} & = \text{entry (3., 30, first)} \\
\text{first} & \Rightarrow \text{current} \\
\text{print } *, \text{first}\%\text{value} & \text{! gives 3.0}
\end{align*}
\]

\[
\begin{align*}
\text{first} & : \quad (3.0, 30, \text{associated}) \\
\text{first}\%\text{next} & : \quad (2.0, 20, \text{associated}) \\
\text{first}\%\text{next}\%\text{next} & : \quad (1.0, 10, \text{null})
\end{align*}
\]
Apart from pointers there is another way to address the same memory area by a different name:

```
EQUIVALENCE
  COMPLEX :: z, c ! complex has two real entries
  REAL :: zr(2) ! real array size 2
  EQUIVALENCE (zr, z, c) ! all three variables access now
                   ! same memory
```

- this assignment is fixed in the declaration block and cannot be changed any more
Applications of pointers IV

- the access to the same *memory area* does not require that the involved variables have to be of the same type:

```
INTEGER :: i(100)
REAL    :: x(100)
EQUIVALENCE (i, x)
```

→ if memory is short, but very dangerous

- for character variables, the sizes do not need to be the same:

```
CHARACTER :: a*4, b(2)*3
EQUIVALENCE (a, b(1)(3:))
```

character variable “a” corresponds to the last four characters of the array “b”
In Fortran, data are passed by “reference”, i.e. the calling procedure

```fortran
CALL SUBROUTINE calcE (velo, pos, E)
```

- passes only start addresses. This enables data exchange with C functions, if they use pointers as arguments:

```c
int *i){ ... }
```

- data exchange is done by the argument list, i.e. use of subroutine calls and void functions in C:

```fortran
CALL c_func (x,i) in Fortran
void c_func_ (float *x, int *i){ ... }
```
Access to C functions II

Note:

- the Fortran compiler appends an underscore “_” to the name of the subroutine, therefore the name of the called C function must end with an underscore in the C source code
- C always expects null terminated strings, so in Fortran:

```fortran
    text = text // CHAR (0) ! appending null
```

Compiling and linking of the Fortran program:

```bash
    gcc    -c    csource.c
    gfortran -c    fprogram.f
    gfortran -o    fprogram.exe    fprogram.o    csource.o
```
As Fortran uses column-(major)-order arrays (first index runs first), while C/C++ uses row-(major)-order arrays (last index runs first) this must be taken into account for procedure calls. So the correspondence is, e.g.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER m(2,3)</td>
<td>int m[3][2] ;</td>
</tr>
</tbody>
</table>

However, note that C/C++ strings must be terminated by null, i.e. text element has effectively only a length of 5.
From above mentioned differences in argument passing between C and Fortran, access from C to Fortran procedures:

- pointers for all arguments in your C code
- especially, no literal (=constant) arguments, e.g.
  \[ x = \text{fort\_func\_}(1., y); \]
- most probably, appending underscores in the call to the Fortran procedures is required, e.g., Fortran: \texttt{REAL MYFUNC(Z,Y)} → C:
  \[ x = \text{myfunc\_}(a, b); \]
- compilation of C and Fortran procedures separately, linking .o files together
- Fortran expects arrays in row-major order
Call to LINPACK

double a[ndim][ndim], help[ndim*ndim] ;
...
for (int i=0; i<ndim; ++i)
  for (int j=0, j<ndim; ++j) help[j+ndim*i] = a[j][i];
...
dgefa_(help, &ndim, &ndim, &ipvt, &info) ;
Insertion: data analysis
Recursive mean I

arithmetic mean

\[ \langle x \rangle = \frac{1}{n} \sum_{i=1}^{n} x_i \] (563)

Problem: calculation of the mean for measured data

- Eq. (563) must be evaluated again for every new data point
- for \( n \gg 1 \) and at the same time \( x_i \ll 1 \) numerical inaccuracy for strict use of Eq. (563) because of \( \rightarrow \) saturation in \( x_i \)

\( \rightarrow \) hence: definition of the recursive mean:

\[ \langle x \rangle_i = \frac{i - 1}{i} \langle x \rangle_{i-1} + \frac{1}{i} x_i \] (564)
Recursive mean II

proof:

\[ i - 1 : \langle x \rangle_{i-1} = \frac{x_1 + \cdots + x_{i-1}}{i-1} \quad (565) \]

\[ i : \langle x \rangle_i = \frac{x_1 + \cdots + x_{i-1} + x_i}{i} \quad (566) \]

\[ = (i - 1) \frac{x_1 + \cdots + x_{i-1}}{i-1} + \frac{x_i}{i} \quad (567) \]

\[ = \frac{i - 1}{i} \langle x \rangle_{i-1} + \frac{x_i}{i} \quad (568) \]

q.e.d.
analogously: **recursive variance**

\[
\sigma^2_i = \frac{i - 1}{i} \sigma^2_{i-1} + \frac{1}{i - 1} (x_i - \langle x \rangle_i)^2
\]  

(569)

proof similar as for recursive mean

correction of a single value:

\[
\begin{align*}
\langle x \rangle_{\text{new}} &= \langle x \rangle_{\text{old}} + \frac{x_{\text{new}} - x_{\text{old}}}{n} \\
\sigma^2_{\text{new}} &= \sigma^2_{\text{old}} + \frac{x^2_{\text{new}} - x^2_{\text{old}}}{n} - \frac{x_{\text{new}} - x_{\text{old}}}{n} \left( \langle x \rangle_{\text{old}} + \frac{x_{\text{new}} - x_{\text{old}}}{n} \right)
\end{align*}
\]  

(570)
Recursive mean IV

proof for the correction of the mean:

\[
\langle x \rangle_{\text{new}} = \frac{1}{n} \left( \sum_{i=1}^{n} x_i - x_{\text{old}} + x_{\text{new}} \right) = \langle x \rangle_{\text{old}} + \frac{x_{\text{new}} - x_{\text{old}}}{n} \quad \text{q.e.d.}
\]
Linear regression I

We already know

Straight line fit without errors

\[ y = b \cdot x + a \] (572)

with slope

\[ b = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \langle x \rangle)(y_i - \langle y \rangle) = \frac{\sigma(xy)^2}{\sigma(x)} \] (573)

and

\[ a = \langle y \rangle - b \cdot \langle x \rangle \] (574)

quality of fit \( y = a + bx \) measured by \( \chi^2 \):

\[ \chi^2(a, b) = \sum_{i=1}^{n} \left( \frac{y_i - a - bx_i}{\sigma_i} \right)^2 \] (575)

with error \( \sigma_i \) in measuring of \( y_i \) (\( x_i \) exact)
Best fit for $\chi^2$ minimum, hence (see also Numerical Recipes)

\[
0 \overset{!}{=} \frac{\partial \chi^2}{\partial a} = -2 \sum_{i=1}^{n} \frac{y_i - a - bx_i}{\sigma_i^2}
\]  
(576)

\[
0 \overset{!}{=} \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=1}^{n} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2}
\]  
(577)

can be rewritten as system of equations:

\[
a \sum_{i=1}^{n} \frac{1}{\sigma_i^2} + b \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} = \sum_{i=1}^{n} \frac{y_i}{\sigma_i^2}
\]  
(578)

\[
a \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} + b \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} = \sum_{i=1}^{n} \frac{x_i y_i}{\sigma_i^2}
\]  
(579)

\[
a \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} + b \sum_{i=1}^{n} \frac{x_i^3}{\sigma_i^2} = \sum_{i=1}^{n} \frac{x_i^2 y_i}{\sigma_i^2}
\]  
(580)
solution for the system:

\[
a = \frac{\sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} \sum_{i=1}^{n} \frac{y_i}{\sigma_i^2} - \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i y_i}{\sigma_i^2}}{\sum_{i=1}^{n} \frac{1}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left( \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2}
\]  \hspace{1cm} (581)

\[
b = \frac{\sum_{i=1}^{n} \frac{1}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i y_i}{\sigma_i^2} - \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \sum_{i=1}^{n} \frac{y_i}{\sigma_i^2}}{\sum_{i=1}^{n} \frac{1}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left( \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2}
\]  \hspace{1cm} (582)
errors in $a$ and $b$ from error propagation for a quantity $f$:

$$
\sigma_f^2 = \sum_{i=1}^{n} \sigma_i^2 \left( \frac{\partial f}{\partial y_i} \right)^2
$$

(583)

where

$$
\frac{\partial a}{\partial y_i} = \frac{\sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - x_i \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2}}{\sigma_i^2 \left( \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left( \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2 \right)}
$$

(584)

$$
\frac{\partial b}{\partial y_i} = \frac{x_i \sum_{i=1}^{n} \frac{1}{\sigma_i^2} - \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2}}{\sigma_i^2 \left( \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left( \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2 \right)}
$$

(585)
Linear regression V

adding up according to Eq. (583)

\[
\sigma_a^2 = \frac{\sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2}}{\sum_{i=1}^{n} \frac{1}{\sigma_i^2}} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left( \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2
\]  

(586)

\[
\sigma_b^2 = \frac{\sum_{i=1}^{n} \frac{1}{\sigma_i^2}}{\sum_{i=1}^{n} \frac{1}{\sigma_i^2}} \sum_{i=1}^{n} \frac{x_i^2}{\sigma_i^2} - \left( \sum_{i=1}^{n} \frac{x_i}{\sigma_i^2} \right)^2
\]  

(587)

Caution!

This (purely formal) error may drastically underestimate the real error in \(a\), \(b\)!
Example: bad fit but small error

\[ a = -9.34 \pm 0.37 \]
\[ b = 6.91 \pm 0.10 \]

→ small formal error, as error in the measurements is small but:
→ model does not fit to data
Our original case: errors $\sigma_i$ not available.

Then: Set $\sigma_i = 1$ in equations for $a$, $b$ and multiply factor $\sqrt{\frac{\chi^2}{n-2}}$ to the formal errors

$$
\sigma^2_a = \frac{\sum_{i=1}^{n} x_i^2}{n \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2} \sqrt{\frac{\chi^2}{n-2}} \tag{588}
$$

$$
\sigma^2_b = \frac{n}{n \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2} \sqrt{\frac{\chi^2}{n-2}} \tag{589}
$$

where

$$
\chi^2 = \sum_{i=1}^{n} (y_i - a - bx_i)^2 \tag{590}
$$
Estimating the errors in fit variables (e.g., the slope $b$)

Methods:

1. formal error from errors in measuring in $y_i \rightarrow$ without consideration of the fit quality $\chi^2$

2. error from $\chi^2$ without consideration of the errors in measuring $y_i$
   $\rightarrow$ usually results in an underestimation of $\sigma_b$
Example: measuring the magnetic field from polarization

Stokes $I$: intensity
Stokes $V = I_R - I_L$ (so: right-hand circularly polarized – left-hand circularly polarized)
$\rightarrow V/I$: fraction of polarized light
$\rightarrow \frac{\lambda^2}{I} \frac{dl}{d\lambda}$: Zeeman shift

Idea: for broad spectral lines (Balmer lines in WD, WR emission lines)
Zeeman splitting not directly detectable because of Doppler shifts.
Therefore: measure “line displacement” per pixel per line together with $V/I$. 
No B-field $\rightarrow$ no correlation. Otherwise, slope gives longitudinal $\langle B_z \rangle$

$$\frac{V}{l} = -\frac{g_{\text{eff}} e \lambda^2}{4\pi m_e c^2} \frac{1}{l} \frac{dl}{dl \lambda} \langle B_z \rangle$$

(591)

with average effective Landé factor
Bootstrapping IV

Example: measuring the magnetic field from polarization

\[ \frac{4.67 \times 10^{-13} \lambda^2}{d/l/d\lambda [G^{-1}]} \]

→ slope dominated by only few data points?
Problem: the distribution of $b$ is usually not known
Idea: construct a distribution with help of Bootstrapping

Construction of a Bootstrapping distribution

random sample $j$ by random drawing of $n$ data $(x_i, y_i)$ from the complete set of $n$ data with repetition and determination of $b_j$. Repeating $m$-times this procedure, where $m \gg n$.

In each random sample are only $\sim 1/e \approx 37\%$ of the original data because of repetitions.

→ result: sample of $m$ measured quantity $b_j$.

Then, determination of the expectation value, variance, etc. for the obtained bootstrapping sample, e.g.,
Example: magnetic field $B_z$ from polarization

Mean = 199.9 G, sigma = 86.5 G
Remember following measurement of a cross section

\[
\begin{array}{cccccccccc}
E_i \text{ [MeV]} & 0 & 25 & 50 & 75 & 100 & 125 & 150 & 175 & 200 \\
\sigma(E_i) \text{ [Mb]} & 10.6 & 16.0 & 45.0 & 83.5 & 52.8 & 19.9 & 10.8 & 8.25 & 4.7 \\
\end{array}
\]

The cross section can be described by Breit-Wigner formula

\[
f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4}
\]  

(592)
Interpolation problem

We want to determine \( \sigma(E) \) for values of \( E \) which lie between measured values of \( E \)

By
- numerical interpolation (assumption of data representation by polynomial in \( E \)):
  - see previous lectures
  - ignores errors in measurement (noise)
- fitting parameters of an underlying model, e.g., Breit-Wigner with \( f_r, E_r, \Gamma \), (taking errors into account), i.e., minimizing \( \chi^2 \)
- Fourier analysis (next semester lecture)
Already seen for linear regression:
We have $N_D$ data points

$$ (x_i, y_i \pm \sigma_i) \quad i = 1, \ldots, N_D \tag{593} $$

and a function $y = g(x)$ (=model) with parameters $\{a_m\}$; fit function to data, such that $\chi^2 = \min$:

$$ \chi^2 := \sum_{i=1}^{N_D} \left( \frac{y_i - g(x_i; \{a_m\})}{\sigma_i} \right)^2 \tag{594} $$

i.e. for $M_P$ parameters $\{a_m, m = 1 \ldots M_P\}$

$$ \frac{\partial \chi^2}{\partial a_m} = 0 \Rightarrow \sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0 \quad (m = 1, \ldots, M_P) \tag{595} $$

→ solve $M_P$ equations, usually nonlinear in $a_m$
least square fitting II

goodness of fit, assumptions

- deviations to model only due to random errors
- Gaussian distribution of errors

→ then, fit is good when $\chi^2 \approx N_D - M_P$ (degrees of freedom)

- if $\chi^2 \ll N_D - M_P$ → probably too many parameters or errors $\sigma_i$ to large (fitting random scatter)
- if $\chi^2 \gg N_D - M_P$ → model not good or underestimated errors or non-random errors

→ for linear fit see above
Non-linear fit

remember Breit-Wigner resonance formula Eq. (592)

\[ f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4} \]  

\( \rightarrow \) determine \( f_r, E_r, \Gamma \)

\( \rightarrow \) nonlinear equations in the parameters

\( a_1 = f_r, \ a_2 = E_r, \ a_3 = \Gamma^2/4 \)  

\( \Rightarrow g(x) = \frac{a_1}{(x - a_2)^2 + a_3} \)  

\( \frac{\partial g}{\partial a_1} = \frac{1}{(x - a_2)^2 + a_3} \) \quad \frac{\partial g}{\partial a_2} = \frac{-2a_1(x - a_2)}{[(x - a_2)^2 + a_3]^2} \quad \frac{\partial g}{\partial a_3} = \frac{-a_1}{[(x - a_2)^2 + a_3]^2} \)  

(597)  

(598)  

(599)
Insert into Eq. (595):

\[
\sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0 \\
\sum_{i=1}^{9} \frac{[y_i - g(x_i, a)](x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0 \\
\sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0
\]  

(600)

→ three nonlinear equations for unknown \(a_1, a_2, a_3\), i.e. cannot be solved by linear algebra but can be solved with help of Newton-Raphson method, i.e. find the roots for the equations above

\[
f_i(a_1, \ldots, a_M) = 0 \quad i = 1, \ldots, M
\]  

(601)
So

\begin{align*}
  f_1(a_1, a_2, a_3) &= \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0 \quad (602) \\
  f_2(a_1, a_2, a_3) &= \sum_{i=1}^{9} \frac{[y_i - g(x_i, a)](x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0 \quad (603) \\
  f_3(a_1, a_2, a_3) &= \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0 \quad (604)
\end{align*}

with initial guesses for \( a_1, a_2, a_3 \).
Newton-Raphson method for a system of nonlinear equations

Remember for 1dim Newton-Raphson method, correction for $\Delta x$:

\[
\begin{align*}
  f(x_0) + f'(x_0) \cdot \Delta x & = 0 \quad (605) \\
  \Delta x & = -\frac{f(x_0)}{f'(x_0)} \quad (606)
\end{align*}
\]

For our system of equations $f_i(a_1, \ldots, a_M) = 0$, we assume that for our approximation (initial guess) $\{a_i\}$ corrections $\{\Delta x_i\}$ exist so that

\[
\begin{align*}
  f_i(a_1 + \Delta a_1, a_2 + \Delta a_2, a_3 + \Delta a_3) & = 0 \quad i = 1, 2, 3 \quad (607)
\end{align*}
\]

→ linear approximation (two terms of Taylor series):

\[
\begin{align*}
  f_i(a_1 + \Delta a_1, \ldots) & \approx f_i(a_1, a_2, a_3) + \sum_{j=1}^{3} \frac{\partial f_i}{\partial a_j} \Delta a_j = 0 \quad i = 1, 2, 3 \quad (608)
\end{align*}
\]
Least square fitting VII

→ set of 3 linear equations in 3 unknowns as explicit equations:

\[ f_1 + \frac{\partial f_1}{\partial a_1} \Delta a_1 + \frac{\partial f_1}{\partial a_2} \Delta a_2 + \frac{\partial f_1}{\partial a_3} \Delta a_3 = 0 \] (609)

\[ f_2 + \frac{\partial f_2}{\partial a_1} \Delta a_1 + \frac{\partial f_2}{\partial a_2} \Delta a_2 + \frac{\partial f_2}{\partial a_3} \Delta a_3 = 0 \] (610)

\[ f_3 + \frac{\partial f_3}{\partial a_1} \Delta a_1 + \frac{\partial f_3}{\partial a_2} \Delta a_2 + \frac{\partial f_3}{\partial a_3} \Delta a_3 = 0 \] (611)

Or as single matrix equation:

\[
\begin{pmatrix}
  f_1 \\
  f_2 \\
  f_3
\end{pmatrix}
+ 
\begin{pmatrix}
  \frac{\partial f_1}{\partial a_1} & \frac{\partial f_1}{\partial a_2} & \frac{\partial f_1}{\partial a_3} \\
  \frac{\partial f_2}{\partial a_1} & \frac{\partial f_2}{\partial a_2} & \frac{\partial f_2}{\partial a_3} \\
  \frac{\partial f_3}{\partial a_1} & \frac{\partial f_3}{\partial a_2} & \frac{\partial f_3}{\partial a_3}
\end{pmatrix}
\begin{pmatrix}
  \Delta a_1 \\
  \Delta a_2 \\
  \Delta a_3
\end{pmatrix}
= 0
\] (612)

Or in matrix notation

\[ f + F' \Delta a = 0 \Rightarrow F' \Delta a = -f \] (613)
Where we want to solve for $\Delta a$ (the corrections)

Matrix $\mathbf{F}'$ sometimes written as $\mathbf{J}$ is called the *Jacobian* matrix (with entries $f'_{ij} = \partial f_i / \partial a_j$).

Equation $\mathbf{F}' \Delta a = -\mathbf{f}$ corresponds to standard form $\mathbf{A} \mathbf{x} = \mathbf{b}$ for systems of linear equations. Formally solution obtained by multiplying with inverse of $\mathbf{F}'$

$$\Delta a = -\mathbf{F}'^{-1} \mathbf{f}$$

$\rightarrow$ inverse must exist for unique solution
$\rightarrow$ same form as for 1d Newton-Raphson: $\Delta x = -(1/f')f$
$\rightarrow$ iterate as for 1d Newton-Raphson till $\mathbf{f} \approx 0$

compute derivatives for the system numerically

$$f'_{ij} = \frac{\partial f_i}{\partial a_j} \approx \frac{f_i(a_j + \Delta a_j) - f_i(a_j)}{\Delta a_j}$$

(615)

with $\Delta a_j$ sufficiently small, e.g., 1% of $a$
Basic matrix problem: solve system of linear equations for unknown $x$

$$Ax = b$$ \hfill (616)

with $n \times n$ matrix $A$ and $N \times 1$ vectors $x, b$

→ best way to solve: Gaussian elimination or LU decomposition

less robust & slower: calculate $A^{-1}$

$$x = A^{-1}b$$ \hfill (617)

→ can be determined by Gauss-Jordan elimination

**Gaussian LU decomposition**

idea: systems of linear equations with triangular matrices easier to solve, e.g.
Linear algebra II

\[ Rx = c, \quad R = \begin{pmatrix} r_{11} & \cdots & r_{1n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & r_{nn} \end{pmatrix} \]  \hspace{1cm} (618)

has solution for \( x \)

\[ x_n = \frac{c_n}{r_{nn}}, \quad x_i = \frac{c_i - \sum_{j=i+1}^{n} r_{ij}x_j}{r_{ii}} \quad \text{for} \quad i = n - 1, \ldots, 1 \]  \hspace{1cm} (619)
Therefore, decompose matrix $A$ into two triangular matrices

$$A = L U$$

(620)

with lower (left) triangular matrix $L$ and upper (right) triangular matrix $U$ (or $R$), hence

$$Ax = LUx = b$$

(621)

$\rightarrow$ first, solve $Ly = b$

(622)

then $Ux = y$

(623)

usually no need to implement by yourself, instead use libraries, e.g., LINPACK:

- DGEFA performs LU decomposition by Gaussian elimination
- DGESL uses that decomposition to solve the given system of linear equations
- DGEDI uses decomposition to compute inverse of a matrix
Nonlinear fit with Newton-Raphson

In our nonlinear fit problem the Newton step

\[ F' \Delta a = -f \]  \hspace{1cm} (624)

can be solved for \( \Delta a \) with help of DGEFA and DGESL:

CALL DGEFA(FPRIME, NDIM, NDIM, IPVT, INFO)
IF (INFO .NE. 0) STOP 'JACOBIAN MATRIX WITH 0 ON DIAGONAL'
CALL DGESL(FPRIME, NDIM, NDIM, IPVT, F)

Barnes, J. & Hut, P. 1986, Nature, 324, 446


Beeman, D. 1976, Journal of Computational Physics, 20, 130


Cromer, A. 1981, American Journal of Physics, 49, 455

Ebert, R. 1955, ZAp, 37, 217

Emden, R. 1907, Gaskugeln (B. Teubner, Berlin)

Feautrier, P. 1964, SAO Special Report, 167, 80


Landau, R. H., Páez, M. J., & Bordeianu, C. C., eds. 2007, Computational Physics (Wiley-VCH)

Lane, H. J. 1870, American Journal of Science, 50, 57


Pang, T. 1997, Introduction to Computational Physics (USA: Cambridge University Press)


Schofield, P. 1973, Computer Physics Communications, 5, 17