Numerical precision

Machine precision I

- as seen for `float 7 + 1.E-7`: because of only 23 bit for mantissa result is 7
- therefore: machine precision ϵ_m is maximum possible number for which $1_\mathrm{c} + \epsilon_\mathrm{m} = 1_\mathrm{c}$, where c means computer representation
- hence: for any number x_c
  $x_\mathrm{c} = x(1 \pm \epsilon), \quad |\epsilon| \leq \epsilon_\mathrm{m}$
- remember: for all 32 bit floats → error in 6th decimal place,
  for 64 bit doubles → error in 15th place

Machine precision II

Determining machine precision

```
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
```

Types of errors I

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, ...

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \approx \sum_{n=0}^{N} \frac{(-x)^n}{n!} = e^{-x} + \mathcal{E}(x, N) \tag{1}$$

where $\mathcal{E}$ vanishes for $N \to \infty$, hence we require $N \gg x$, expecting large $\mathcal{E}$ for $x \approx N$

Types of errors II

- *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

$$5\left(\frac{1}{9}\right) - \frac{5}{9} = 0.555 - 0.556 = -0.001 \neq 0 \tag{2}$$

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

again: for a *float* number like

$$x = 11223344556677889900. = 1.1223344556677889900 \times 10^{19} \tag{3}$$

only the first part (32 bit: 1.12233) is stored, while exponent is stored exactly

Subtractive cancellation I

- consider computer representation x_c of an exact number x:

$$x_\mathrm{c} \simeq x(1+\epsilon_x) \tag{4}$$

with relative error ϵ_x in x_c (similar to machine precision)

- so for subtraction

$$a = b - c \rightarrow \; a_\mathrm{c} \simeq b_\mathrm{c} - c_\mathrm{c} \simeq b(1+\epsilon_b) - c(1+\epsilon_c) \tag{5}$$

$$\rightarrow \; \frac{a_\mathrm{c}}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a} \tag{6}$$

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

$$\frac{a_\mathrm{c}}{a} = 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a}\max(|\epsilon_b|, |\epsilon_c|) \tag{7}$$

as $b \simeq c \rightarrow b/a \gg 1 \rightarrow$ relative error in a blown up

Subtractive cancellation II

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: $v_n = \dfrac{x_{n+1} - x_{n-1}}{2\Delta t}$
- solution of quadratic equation for $b \gg 4ac$:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{or} \quad x_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \tag{8}$$

- in e^{-x} for large x: the first terms $(1 - x + x^2/2 - \ldots)$ are large → as result is small → subtraction by other large terms → improve algorithm by calculating $1/e^x$

Errors in algorithms I

Roundoff error accumulation:

$$a = b * c \rightarrow \; a_\mathrm{c} = b_\mathrm{c} * c_\mathrm{c} = b(1+\epsilon_b) * c(1+\epsilon_c) \tag{9}$$

$$\rightarrow \; \frac{a_\mathrm{c}}{a} = (1+\epsilon_b)(1+\epsilon_c) \simeq 1 + \varepsilon_\mathrm{b} + \varepsilon_\mathrm{c} \tag{10}$$

(neglecting very small ϵ^2 terms) → as for physical error-propagation: adding up relative errors

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

$$\epsilon_\mathrm{roundoff} \approx \sqrt{N}\,\epsilon_\mathrm{m} \tag{11}$$

→ if no detailed error analysis available;
otherwise, if not random: $\epsilon_\mathrm{roundoff} \approx N\epsilon$

Errors in algorithms II

Usually: if A is correct result and numerical approximation is $A(N)$, accuracy of $A(N)$ improves by adding more terms, i.e.

$$\epsilon_{\mathrm{appr}} \simeq \frac{\alpha}{N^{\beta}} \tag{12}$$

with some constants α, β depending on algorithm

However, each calculation step might increase roundoff error, so

$$\epsilon_{\mathrm{tot}} = \epsilon_{\mathrm{appr}} + \epsilon_{\mathrm{roundoff}} \simeq \frac{\alpha}{N^{\beta}} + \sqrt{N}\epsilon_{\mathrm{m}} \tag{13}$$

Hopefully: ϵ_{appr} dominant, but $\epsilon_{\mathrm{roundoff}}$ grows slowly
→ stop calculation (optimum N) for minimum ϵ_{tot}

Errors in algorithms III

If we knew exact A (then we also wouldn't need to calculate $A(N)$):

$$A(N) \simeq A + \frac{\alpha}{N^\beta} \tag{14}$$

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

$$A(N) - A(2N) \simeq \frac{\alpha}{N^\beta} \simeq \epsilon_\text{appr} \tag{15}$$

Errors in algorithms IV

Minimize the error

Let's assume that some algorithm behaves like

$$\epsilon_{\mathrm{appr}} \simeq \frac{1}{N^2} \rightarrow \epsilon_{\mathrm{tot}} \simeq \frac{1}{N^2} + \sqrt{N}\,\epsilon_{\mathrm{m}} \tag{16}$$

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

$$\frac{d\epsilon_{\mathrm{tot}}}{dN} = 0 \rightarrow N^{\frac{5}{2}} = \frac{4}{\epsilon_{\mathrm{m}}} \tag{17}$$

So, for single precision ($\epsilon_{\mathrm{m}} \simeq 10^{-7}$)

$$N^{\frac{5}{2}} = \frac{4}{10^{-7}} \rightarrow N \simeq 1099 \rightarrow \epsilon_{\mathrm{tot}} = 4 \times 10^{-6} \tag{18}$$

→ total error dominated by ϵ_{m}, typical for single precision

Errors in algorithms V

Minimize the error II

So, if another algorithm

$$\epsilon_{\mathrm{appr}} \simeq \frac{2}{N^4} \rightarrow \epsilon_{\mathrm{tot}} \simeq \frac{2}{N^4} + \sqrt{N}\,\epsilon_{\mathrm{m}} \tag{19}$$

And again minimum error obtained for an N

$$\frac{d\epsilon_{\mathrm{tot}}}{dN} = 0 \rightarrow N^{\frac{9}{2}} = \frac{16}{\epsilon_{\mathrm{m}}} \rightarrow N \simeq 67 \rightarrow \epsilon_{\mathrm{tot}} = 9 \times 10^{-7} \tag{20}$$

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`

Passing arrays to functions in C++ I

Declaration of a 1d-array:

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

Declaration of a function with an array type argument:

```
int sumsort (int m[], int n) ; // n = lenght of m
```

Calling a function with an array type argument:

```
sum = sumsort (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

Passing arrays to functions in C++ II

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

Passing arrays to functions in C++ III

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:

```
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.

Excursus: Libraries III

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

Excursus: Libraries IV

Dynamic libraries

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

```
linux-vdso.so.1 (0x00007fff72bff000) †
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007ff2d9c0b000)
```

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`)

Excursus: Libraries VI

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.

Linking and libraries I

- 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!

Linking and libraries II

- dynamic libraries must be located in a default system path (e.g., `/lib`) or the 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 linking I

- 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`

make I

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

make II

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

make III

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

```
echo 'int main() {}' > myprog.cpp
make myprog
```

executes `g++ -o myprog myprog.cpp`[1]

- `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`

make IV

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)
```

make V

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

make VI

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:

  ```
  CC = gcc
  CFLAGS = -O3
  PROJECT = galaxy
  ```

- variables can, as in the shell, be hold together with help of curly braces `${OBJECTFILES}`, or with help of round parentheses `$(CFLAGS)`

make VII

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`.

make VIII

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)
```

make IX

`Makefile` uses a shell-like syntax:

- comments are started with a #:

  ```
  # a comment
  ```

- one command per line, multiple commands via ; and line continuation via \

  ```
  $FC $?  ; ldconfig
  ```

- every command corresponds to a shell command, and is printed before execution:

  ```
  .c.o :
        echo "Hello ${USER}"
  ```

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

  ```
        @echo "Hi ${DATE}"
  ```

make X

- 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 {}.

make XI

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 `.`

make XII

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.

make XIII

- `.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`.