Lecture 1

Programming, Debugging, and Compiling

Basics of C, (fortran), C++

Compiling

Debugging

An example with trigonometric functions

programming basics: Fortran

- Syntax can be found on the web
  - e.g http://h18009.www1.hp.com/fortran/docs/rm/dflrm.htm
- Nowadays free style format available (lines of 132 characters).
- Used to be 72 characters (punch cards)
  - 1st column : to indicate comment line
  - 2nd-5th: labels
  - 6th: continuation of previous line
  - 7-72: statements
  - 73-80: comment

```
c example
      program p()
      real*8 a
      integer i
      do 10, i=1,100,3
      a=exp(i*.1d-01)
      write(6,99) i,a
   10 continue
    99 format(i4,E12.2)
      end
```

Fortran

- Fortran passes variables by reference:
  - subroutines change the value of passed parameters inside the main program (or the subroutine that called it)
- Special care needed with common blocks
  - it is advisable to specify the common block in an include file.
  - e.g. routine 1
    - implicit integer*4 [a-h]
    - real*8 a(100)
    - common/h/a,b,c,d
  - routine 2
    - float f
    - dimension f(2,10,10)
    - /common/h/f
  - with such constructs one may overwrite ALL kinds of variables. If f(1,1,3) is overwritten, some a will have an arbitrary value. Note, that not even the dimension of the objects in memory are the same. Indeed, I know of programs where someone put the string ‘STOP’ in a real*4 of an array of another routine.

C

- The C programming language by Kernigham and Ritchie, ed. Prentice Hall, New Jersey.
- (ANSI C)
- common compilers: visual studio on windows, gcc/g++ on unix.
- C passes all variables **by value**.
  - This means that a copy of the variable is given to the function or method that you call with it.
  - If you update a parameter within a routine, it is NOT updated in the caller routine
- C is more complex than fortran, using pointers and variables.
  Frequently-made syntax mistakes:
  - if (a=1)
    - always true, a is set to 1.
  - float *b, myvar; b=&myvar; free(b);
    - Unallocates memory at float myvar as well. Memory may be overwritten.
    - especially when variables themselves are pointers, things may go awry.
  - **int j=6; double a = j/17; // a = 0, not 6.0/17. write a=j/17.0 or a =1.0/17*j or{a=j; a=a/17;} to get this correct.**

C

- **Frequently-made syntax mistakes, continued**
  - double a[100]; a[100]=1e7;
    - array a runs from 0 to 99, a[100] is not defined
  - syntax, e.g. a*b, a**ptrb, a/*ptrb
    - last statement starts comment
  - nested comments
    - e.g. when you comment out an action that already has comments in it
    - Comment runs from first opened /* to first */
  - Order in which statements are executed
    - Be careful with statements like

```
if ( a++ <1 && a/=2) { ...
```

C++

- Bjarne Stroustrup, the C++ programming language
- On unix : g++ compiler
- On windows: visual studio
- C++ contains C
  - C++ is written in C. C syntax is supported.
  - pointers, references and objects
- Object-oriented
  - method overloading: by use of the keyword const a simple function working on variables (pointers/references/objects) may be defined in 18 different ways.
  - for most classes (certainly containing pointers) one should specify the assignment constructor, copy constructor and destructor as well.
  - In Numerical Recipes: include “nr3.h” – structs containing the methods. Like root, it is not very nicely written C++, it is rather an extension of C. Examples will be discussed, I'll comment on the include files where appropriate.
    - Most importantly: numerical recipes does not use classes and the include files are not enclosed with #ifndef….#endif

- Nowadays, often-used functions like cos, sin, atan, exp, etc are very fast and robust. You can test that by
  - creating a program which implements different methods for calculating these cosines and sines
  - checking the precision by applying a formula, e.g.

$$1 - \sin^2 x - \cos^2 x = 0$$

  - checking the time of execution with the time and gprof commands

methods for the cosine and sine

1) cos and sin functions, mathematical library
2) mycos1, mysin1:
   - 8-byte doubles. factor of power series is remembered and updated. Summation ends when last term is smaller than 10^-18 times the acquired total sum.
   - mycos1 is calculated between PI and –PI
   - mysin1 is calculated between PI/2 and –PI/2
3) mycos2,mysin2:
   - same as mycos1,mysin1 but for 4-byte floats
4) mycos3, mysin3
   - power series: each term calculated separately with fac(i) and pow(x,i).
   - how much slower?
   - Numrec.html

```
#ifndef myfunc_h
#define myfunc_h
/*
   mycos1 and mysin1: ordinary taylor expansions of sine and cosine.
   End the expansion when a term is smaller than 10e-18 of the
   accumulated series  Note: pi is given in 21 decimals here
*/

#define PI 3.14159265358979323846

double mycos1(double x);
double mysin1(double x);
float mycos2(float x);
float mysin2(float x);
double fac(int i);
double mycos3(double x);
double mysin3(double x);
void powercount();
void multiply();
#endif
```

Standard compiler directives.

```
/* mycos1 and mysin1: ordinary taylor expansions of sine and cosine.
  End the expansion when a term is smaller than 10e-18 of the accumulated series
  Note: pi is given in 21 decimals in myfunc.h*/

#include "myfunc.h“
#include <stdio.h>
#include <math.h>
double mycos1(double x) {
        int i = x/PI/2;
        double sum=0;
        double term=1;
         int fac=0;
        x= x -2*PI*i;
        if (x>PI) x -=2*PI;
        if (x<-PI) x+=2*PI;
/* Now, x is between plus and minus PI. Obviously, one could make x smaller
than PI/4 by using cosine and sine identities */
          while (fabs(term) > fabs(sum)*1e-18) {
              sum+= term;
              term*= x/++fac;
              term*= -x/++fac;
        }
        return sum+term;
}
```

```
/* mycos2 and mysin2 are duplicates from mycos1 and mysin1.
 * the difference is that they operate on floats instead of doubles.
 * in principle, one could use overloaded functions in C++ or use template arguments.
 * however, in that case a call like mycos(1.3) is ill-defined: depending on the
 * compiler it would resort into a 4-byte float or an 8-byte double */

float mycos2(float x) {
        int i = x/2/PI;
        float sum=0;
        float term=1;
         int fac=0;
        x= x -2*PI*i;
        if (x>PI) x -=2*PI;
        if (x<-PI) x+=2*PI;
        while (fabs(term) > fabs(sum)*1e-18) {
                sum+= term;
                term*=x/++fac;
                term*=-x/++fac;
        }
        return sum+term;
}
```

```
/*  mycos3, mysin3 : now each term is calculated with pow(x,y). and a call to fac()*/
double fac(int i) {
        double faculteit=1.0;
        while (i>1) faculteit*=i--;
        return faculteit;
}

double mycos3(double x) {
        int i = (int) x/PI/2;
        double sum=0;
        double term=1;
        x= x -2*PI*i;
        if (x>PI) x -=2*PI;
        if (x<-PI) x+=2*PI;
/* Now, x is between plus and minus PI. Obviously, one could gain even more
   by using cosine and sine identities. */
        i=0;
        while (fabs(term) > fabs(sum)*1e-18) {
                i++;
                sum+= term;
                term=pow(x,2.0*i)/fac(2*i);
                if (i%2) term=-term;
        }
        return sum+term;
}
```

```
void powercount() {
/* used to see the time needed by a call to pow() */
        int i;
        double x=0;
        for (i=0; i<1000000; i++) {
                x+=pow(2.3, i/32000.0);
        }
        fprintf(stdout,"powercount : %e\n",x);
}

void multiply() {
/* used to see the time needed by a call to multiplication, in comparison to pow() */
        int i;
        double x=1.0;
        for (i=0; i<1000000; i++) {
                x*=i/1000.0;
        }
        fprintf(stdout,"multip : %e\n",x);
}
```

Note: divide i by a floating-point number. i/32000 = 0 for i<32000

```
#include <math.h>
#include <stdio.h>
#include "myfunc.h“
int main() {
// sample program to verify the speed and precision of Taylor-series expansion of sine and cosine
// first, as a baseline, calculate the speed with the optimized standard library functions
        int i,j;
        double a,b;
        double av[4] = {0,0,0,0}; // 4 times the average of sine**2 + cosine**2 -1;
        double rms[4] = {0,0,0,0}; // and the RMS of them
        int tot=0;
        for (j=0; j<10; j++) for (i=0; i<100000; i++) {
                tot++;
                a = cos(i/5000.0);
                b = sin(i/5000.0);
                a = 1 - a*a -b*b;
                av[0] +=a;
                rms[0] += a*a;
                a = mycos1(i/5000.0);
                b  = mysin1(i/5000.0);
                ..........................................
```

```
        av[1] +=a;
        rms[1] += a*a;
        a = mycos2((float) i/5000.0);
        b = mysin2((float) i/5000.0);
        a = 1 - a*a -b*b;
        av[2] +=a;
        rms[2] += a*a;
        a = mycos3((double) i/5000.0);
        b = mysin3((double) i/5000.0);
        a = 1 - a*a -b*b;
        av[3] +=a;
        rms[3] += a*a;
}
```

.....

```
        for (i=0; i<4; i++) {
                if (tot>0) {
                        av[i]/=tot;
                        rms[i]/=tot;
                }
                rms[i] -= av[i]*av[i];
                if (rms[i]<0) fprintf(stderr,
"Due to precision, negative RMS in routine %d\n",i);
                else rms[i] = sqrt(rms[i]);
                fprintf(stdout,"routine %d, average %e, rms %e\n",i,av[i],rms[i]);
        }
        powercount();
        multiply();
}
```

Compiling

- when you have source files, you can make a binary executable or a (shared) library by compiling.
- small programs: compile inline
- larger programs: use make files (man gmake),
  eclipse, visual studio projects
- packages, version control: use e.g. cvs, git
- (LHCb: thousands of classes, tens of packages, hundreds of versions, Gb codes: cmt management)

compiling under unix

the above programs can be obtained from my website. You can compile them with e.g.

- `gcc -pg -g -Iinclude -L/usr/local/lib -lm speed.c myfunc.c`

-pg : get ready for profiling

-g : produce debug information in the executable

-Iinclude: look for include files (.h) in the directory include

-L/usr/local/lib: look for libraries in the directory /usr/local/lib

-lm : link with the library libm.a or libm.so

running, timing

- in the shell, type
  - time ./a.out
- this results in:

```
routine 0, average -2.501814e-21, rms 6.261338e-17
routine 1, average 4.098504e-17, rms 2.703191e-16
routine 2, average 8.092013e-10, rms 1.283643e-07
routine 3, average -8.139485e-19, rms 3.609235e-16
powercount : 5.304614e+13
real 0m40.151s
user0m40.126s
sys  0m0.005s
```

the program took 40 seconds

the amount of time, spent in each subroutine, can be monitored by gprof.

Profiling: gprof output

Profiling can be done when you compile the program with the -pg flag. The executable will run somewhat slower, since many system calls are done to determine the time that each step took.

results: (obtained as output from shell command gprof a.out gmon.out)

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 43.76      9.74     9.74 234074860     0.00     0.00  fac(int)
 10.76     12.13     2.39 10000000     0.00     0.00  mycos1(double)
  8.69     14.06     1.93 10000000     0.00     0.00  mysin1(double)
  8.44     15.94     1.88 10000000     0.00     0.00  mycos3(double)
  8.37     17.81     1.86 10000000     0.00     0.00  mycos2(float)
  8.33     19.66     1.85 10000000     0.00     0.00  mysin3(double)
  6.89     21.19     1.53 10000000     0.00     0.00  mysin2(float)
  4.64     22.22     1.03                             main
  0.23     22.27     0.05        1    50.09    50.09  powercount()
```

And 9 more pages...

compiling, error messages

- In myfunc.c, I omitted one */ behind identities in the function mycos1. (nested comments!) Error message is:
- ..../NumRec/speed.c:21: undefined reference to ‘mysin1’
- in myfunc.h, I omitted
- double mycos1(double x);
  - **error message: none**
    - When compiling in 2 steps, the source files into object files and the object files into executable. g++ catches the error, but gcc does not
  - **The code is known but not linked correctly!**
    - When entering the debugger you see that the code is executed, only the double that is returned is interpreted as a 4-byte integer in the main program
  - Depends on version of compiler/operating system
- results: non-sensible:

```
routine 0, average 1.399112e-19, rms 6.280965e-17

routine 1, average -1.534856e+18, rms 1.376143e+18

routine 2, average 2.312455e-10, rms 1.181418e-07

routine 3, average -1.907645e-18, rms 3.332287e-16
```

Compiling errors

- Compiling errors can look quite overwhelming. If you cannot figure out what is wrong, ask for help.
- Here is the error when one omits 2 brackets in a standard container class for the iterator value returned by end() :
- /home/henkjan/git/tremornet.pilots/LoRaSimulations/MC/error.txt

debugging

- under unix, one can invoke gdb by typing gdb a.out
- the debug session, used to find this previous problem, is given in screen shots on the next slides.
  - man gdb for manual
  - gdb comes without warranty; sometimes it has difficulties printing out (complicated) objects and sometimes the objects are not available on the stack.
- most important commands:
  - break (sets a breakpoint. The program is executed until here
  - run (start running)
  - list, list 40-70 (list a piece of the source code)
  - print (print sin(x)+3, print *a,a,&a, print a[100])
  - cont – run until next breakpoint
  - clear – remove current breakpoint
  - step, next : execute the next command. step steps into a function, next steps over it.

Debug session

```
-bash-4.1$ gdb a.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-90.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /user/henkjan/public_html/NUMREC/PROGRAMMING/a.out...done.
(gdb) break main
Breakpoint 1 at 0x401189: file speed.c, line 13.
(gdb) run
Starting program: /user/henkjan/public_html/NUMREC/PROGRAMMING/a.out

Breakpoint 1, main () at speed.c:13
13              double av[4] = {0,0,0,0}; // 4 times the average of sine**2 + cosine**2 -1;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.192.el6.x86_64
(gdb) list
8       int main() {
9       // sample program to verify the speed and precision of taylor series expansion of sine and
cosine
10      // first, as a baseline, calculate the speed with the optimized standard library functions
11              int i,j;
12              double a,b;
13              double av[4] = {0,0,0,0}; // 4 times the average of sine**2 + cosine**2 -1;
14              double rms[4] = {0,0,0,0}; // and the RMS of them
15              int tot=0;
16              for (j=0; j<10; j++) for (i=0; i<STEPS; i++) {
17                      tot++;
(gdb) list 16,36
16              for (j=0; j<10; j++) for (i=0; i<STEPS; i++) {
17                      tot++;
18                      a = cos(i*10.0/STEPS);
19                      b = sin(i*10.0/STEPS);
20                      a = 1 - a*a -b*b;
21                      av[0] +=a;
22                      rms[0] += a*a;
23                      a = mycos1((double) i*10.0/STEPS);
24                      b = mysin1((double) i*10.0/STEPS);
25                      a = 1 - a*a -b*b;
26                      av[1] +=a;
27                      rms[1] += a*a;
28                      a = mycos2((float) i*10.0/STEPS);
29                      b = mysin2((float) i*10.0/STEPS);
30                      a = 1 - a*a -b*b;
31                      av[2] +=a;
32                      rms[2] += a*a;
33                      a = mycos3((double) i*10.0/STEPS);
34                      b = mysin3((double) i*10.0/STEPS);
35                      a = 1 - a*a -b*b;
36                      av[3] +=a;
(gdb) break 23
```

Debug session

```
(gdb) break 23
Breakpoint 2 at 0x4012aa: file speed.c, line 23.
(gdb)
Note: breakpoint 2 also set at pc 0x4012aa.
Breakpoint 3 at 0x4012aa: file speed.c, line 23.
(gdb) cont
Continuing.

Breakpoint 2, main () at speed.c:23
23                      a = mycos1((double) i*10.0/STEPS);
(gdb) print i
$1 = 0
(gdb) print a
$2 = 0
(gdb) step
mycos1 (x=0) at myfunc.c:12
12              int i = x/PI/2;
(gdb) list
7       #include "myfunc.h"
8       #include <stdio.h>
9       #include <math.h>
10
11      double mycos1(double x) {
12              int i = x/PI/2;
13              double sum=0;
14              double term=1;
15              int fac=0;
16              x= x -2*PI*i;
(gdb) list 16,30
16              x= x -2*PI*i;
17              if (x>PI) x -=2*PI;
18              if (x<-PI) x+=2*PI;
19      /* Now, x is between plus and minus PI. Obviously, one could even make x smaller than PI/4
20      by using cosine and sine identities */
21              while (fabs(term) > fabs(sum)*1e-18) {
22                      sum+= term;
23                      term*= x/++fac;
24                      term*= -x/++fac;
25              }
26              return sum+term;
27      }
28
29      double mysin1(double x) {
30              int i = x/PI/2;
(gdb) break 26
Breakpoint 4 at 0x40081b: file myfunc.c, line 26.
```

Debug session

```
Continuing.

Breakpoint 4, mycos1 (x=0) at myfunc.c:26
26              return sum+term;
(gdb) c
Continuing.

Breakpoint 2, main () at speed.c:23
23                      a = mycos1((double) i*10.0/STEPS);
(gdb) print sum+term
No symbol "sum" in current context.
(gdb) c
Continuing.

Breakpoint 4, mycos1 (x=1.0000000000000001e-05) at myfunc.c:26
26              return sum+term;
(gdb) print sum+term
$3 = 0.99999999995
(gdb) next
27      }
(gdb) next
main () at speed.c:24
24                      b = mysin1((double) i*10.0/STEPS);
(gdb) list 23
18                      a = cos(i*10.0/STEPS);
19                      b = sin(i*10.0/STEPS);
20                      a = 1 - a*a -b*b;
21                      av[0] +=a;
22                      rms[0] += a*a;
23                      a = mycos1((double) i*10.0/STEPS);
24                      b = mysin1((double) i*10.0/STEPS);
25                      a = 1 - a*a -b*b;
26                      av[1] +=a;
27                      rms[1] += a*a;
(gdb) print a
$4 = 0
(gdb) n
25                      a = 1 - a*a -b*b;
(gdb) print b
$5 = 9.9999999998333335e-06
(gdb) print &a
$6 = (double *) 0x7fffffffe148
(gdb) print &mycos1
$7 = (double (*)(double)) 0x4006b4 <mycos1>
(gdb) quit
A debugging session is active.
```

Syntax problems, compiling and debugging

- The aim of the course is that
  - you learn to select optimal algorithms
  - you are capable of verifying the correctness of your code;
  - You can estimate the size of the numerical errors introduced in your code
- although it is necessary that you can program and debug, you can ask for assistence when you are stuck. I don't mind helping with debugging your code or helping with syntax issues; in the end the goal is that you can deliver a numerically sound working program. There will be typically 2 or 3 classes before the deadline of an exercise so there is opportunity for this (provided that you start your exercises right after the lecture during the practical part of the class)