A practical guide to computer simulation II

Alexander K. Hartmann, University of Göttingen

May 9, 2003

2.11 Command-line arguments

Make program flexible: pass arguments

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

`argc`= number of arguments, arguments are stored in `argv`, the string `argv[0]` contains the program name, the following strings the arguments.
Example: when calling `simul -start 1000 50`, then

```
argv[0]="simul"  i.e. argv[0][0]='s', argv[0][1]='i',...
argv[1]="-start"
argv[2]="1000"   i.e. argv[2][0]='1', argv[2][1]='0', ...
argv[3]="50"
```

One can use the function `atoi(`*string*`)` to convert a string which represents a number (like `argv[2]`) into the actual number.
Treating the arguments and options, example:

```
int argz, start, end, size;

start = 0; end = 10000;                         /* default values */
while((argz<argc)&&(argv[argz][0] == '-'))   /* process options */
{
  if(strcmp(argv[argz], "-start") == 0)
    start = atoi(argv[++argz]);
  else if(strcmp(argv[argz], "-end") == 0)
    end = atoi(argv[++argz]);
  else                                        /* option not found */
  {
    fprintf(stderr, "unkown option: %s\n", argv[argz]);
    exit(1);
  }
  argz++;
}

if( (argc-argz) < 1)            /* enough arguments remaining ? */
{
```

```
    fprintf(stderr, "USAGE: %s {<options>} <size>\n", argv[0]);
    exit(1);
  }
  size = atoi(argv[argz++]);                    /* read parameter */
```

2.12 Macros

Macros = shortcuts for code, allows higher flexibility
Are processed *before* main compiling (*preprocessing*).
Form: (in ONE line, or use \; beginning if *first* column)

`#define` *name* *definition*

Example constant:

```
#define  PI  3.1415926536
```

Macros are just textually replaced by preprocessor, but not in strings.
Use macros for flexible compiling using `#ifdef` and `#ifndef`, e.g.:

```
#ifdef UNIX
  ...
#endif
#ifdef MSDOS
  ...
#endif
```

Instead defining constants inside the code, one can define them within the compiler call (for higher flexibility), using the option `-D`, e.g. for just defining (without value)

```
cc -DUNIX -o simul simul.c -Wall -lm
```

or (with value)

```
cc -DPI=3.1456 -o simul simul.c -Wall -lm
```

Macros with arguments

```
#define  MIN(x,y)  ( (x)<(y) ? (x):(y) )
```

Beware of side effects: `MIN(a++,b++)` `a` or `b` may be increased twice (does not happen in functions)

2.13 Make files

Make files = used to control compiling of many-parts projects. Basic idea: describe dependencies between parts and rules to generate *targets* in the `Makefile`.

Form of rule:

target : *sources*

<tab> *command(s)*

Here: application to compile programs. But possible to apply in any part of automatic source → target file generation.
Example:

```
simulation.o: simulation.c simulation.h
<tab>   cc -c simulation.c
```

Call compiling. type `make`.
Make builds always first target, e.g. for three targets:

```
all: object1 object2 object3

object1:  <sources of object1>
<tab>    <command to generate object1>

object2: ...
<tab>    <command to generate object2>

object3: ...
<tab>    <command to generate object3>
```

One can call also just `make object3`.
Also nested dependencies are allowed (but not circular).
Define variables (macros):

variable=definition

E.g.

```
OBJECTS=module1.o additions.o
```

Access

```
$(OBJECTS)
```

Environment variables like `$HOME` are accessible inside the `Makefile`.

Special predefined variables (can be changed)

- `$(CC)` = compiler name
- `$@` = name of current target
- `$(CFLAGS)` = optione passed to C compiler
- `$(LIBS)` = options passed to linker

Also standard rules (e.g. for `.o` files) are predefined.

lines beginning with "#" are comments.

Final example

```
#
# sample make file
#
OBJECTS=simulation.o init.o run.o
OBJECTSEXT=$(HOME)/lib/analysis.o
CC=gcc
CFLAGS=-g -Wall -I$(HOME)/include
LIBS=-lm

simulation: $(OBJECTS) $(OBJECTSEXT)
<tab>   $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(OBJECTSEXT) $(LIBS)

$(OBJECTS): datatypes.h

clean:
<tab>   rm -f *.o
```

2.14 Excercise II

Extend the program for the sputter deeposition in two ways:

- Make the system size L a program parameter. Hence you don't have to recompile in case you want to run different system sizes.

  For this purpose you have to allocate the system dynamically by malloc.

- Change the program such that it performs k (another parameter) runs and prints the *average* $\overline{W(t)}$ roughness over all runs as a function of the simulation time t (instead of the roughness for each run). Here $\overline{\cdots}$ denotes the average over different runs, i.e. $\overline{W}(t) = \frac{1}{k}\sum_{i=1}^{k} W_i(t)$. Hint: Use an array (over different times t) to store the sum $\sum_i W_i(t)$ during the simulation.

  Extension: print along with the average also the error bar, i.e. the quantity $\sigma(t) = \sqrt{\left(\overline{W^2(t)} - \overline{W(t)}^2\right)/k}$.

3 Tools for Testing

3.1 gdb

gdb (gnu debugger) = source code debugger
allows for stepwise executing, investgating/changing data.

Example program `gdbtest.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int t, *array, sum = 0;

    array = (int *) malloc (100*sizeof(int));
```

```
    for(t=0; t<100; t++)
        array[t] = t;
    for(t=0; t<100; t++)
        sum += array[t];
    printf("sum= %d\n", sum);
    free(array);
    return(0);
}
```

Compiling: Use option `-g`

`cc -o gdbtest -g gdbtest.c`

invoke:

`gdb gdbtest`

Most important commands of debugger (online help: `help`)

1. List source code:
   `list` or just `l`: List next/current ten lines
   `list` <*from*>, <*to*>: list range of lines

2. Set breakpoints (there execution will stop):
   `break` <*line number*> or **b** <*line number*>

   ```
   (gdb) b 11
   Breakpoint 1 at 0x80484b0: file gdbtest.c, line 11.
   ```

3. Show breakpoints:
   `info break`

4. Remove breakpoints: `clear` <*line number*>
   `delete` <*number of breakpoint*>

5. start running program:
   `run` <*arguments*> (abrev.: `r`)
   Runs until breakpoint, exception or program end

   ```
   (gdb) r
   Starting program: gdbtest

   Breakpoint 1, main (argc=1, argv=0xbffff384) at gdbtest.c:11
   11          for(t=0; t<100; t++)
   ```

6. inspec data:
   `print` <*expression*>

   ```
   (gdb) p array
   $1 = (int *) 0x8049680
   (gdb) p array[99]
   $2 = 99
   (gdb) print *(array+5)
   $3 = 5
   ```

7. Automatic printing after each stop:
   `display`
8. Modify variables:
   `set` *<variable>*=*<expression>*

   ```
   (gdb) set array[99]=98
   ```
9. Continue running:
   - one source code line:
     `next` (`n`)
   - with stepping into subroutines:
     `step` (`s`)
   - until breakpoint, exception, program end:
     `continue` (`c`)
10. show calling sequence of subroutines (usefull when exception occurs):
    `where`.
    One can move within the calling sequence with `up` and `down`
11. exit program:
    `quit`.

3.2 ddd

ddd = data display debugger.
Graphical interface to *gdb*.