Outline

Aims with computer lab "Stochastic simulations"

Main aim:

- To get a better understanding of the different simulation methods.

(For the implementation one needs to think things through in detail.)

A number of additional aims:

- To get experience with using Linux computers—the kinds of systems used at the supercomputer centers.
- To get more experience with programming in C.
- To see how to use a single source code for several somewhat different programs.

How to run the program

```
$ ./sim N=64 rho=0.5 T=1.0 read=0064_r0.500_T1.000_start deltat=0.011 nblock=10 run

Read from conf/0064_r0.500_T1.000_start with data for 64 particles

--- Molecular dynamics of a Lennard-Jones gas ---

Gas with 64 particles at rho = 0.5, T = 1, alpha = 0, deltat = 0.011, L = 11.314
Potential energy = -1.33933
Kinetic energy   = 1.04104
Equilibrate: 1000...done

Simulate 10 blocks x 1000 samples each: 1 2 3 4 5 6 7 8 9 10
Configuration with 64 particles written to conf/0064_r0.500_T1.000_alpha0.00_dt011

Potential E: -1.29895 +/- 0.000600459
Kinetic  E : 1.00165 +/- 0.000617618
Total  E   : -0.297295 +/- 0.000243392

Pressure   : 0.58852 +/- 0.00198527
$
```

Program structure

```
void run_simulation(Par *par, double *atoms) {
  // 1 Initialization... 2 Equilibration... 3 Production run:
  for (iblock = 0; iblock < par->nblock; iblock++) {
    ...                 // step forward in time
    ...                 // measure!
  }
  // 4 Print out results
}

int read_args(Par *par, char *arg) {
  if (!strcmp(arg, "N")) {
    par->n = strtol(s, NULL, 10);
    return 1;
  }
  if (!strcmp(arg, "run")) {
    run_simulation(par, atoms);
    return 1;
  }
  return 0;
}

int main(int argc, char *argv[])
{
  Par par;                  // Initialize parameter struct
  par.rho = 0.6;
  for (iarg = 1; iarg < argc; iarg++)
    if (!read_args(&par, argv[iarg]))
      exit(EXIT_FAILURE);
  exit(EXIT_SUCCESS);
}
```

Simple "atomic" variables. . . reading input

- Define variables before using them:

```
int x = 4; // Four bytes
char letter; // One byte
double val; // Eight bytes

letter = 'a';
val = 3.14 / x;
```

- Input using the `scanf()` function. To print out use `printf()`:

```
int main()
{
    int this_is_a_number;
    printf( "Please enter a number: " );
    scanf( "%d", &this_is_a_number );
    return 0;
}
```

- Input from the command line: With "./prog 3.5":

```
int main(int argc, char *argv[])
{
  double rho;
  rho = strtod(argv[1], NULL); // strtod = string to double
```

Functions in C

- Simple example program:

```
#include <stdio.h>

int mult(int x, int y); // Declaration of mult (prototype)

int main()
{
  int x = 12;
  int y = 19;
  int result;

  result = mult(x, y);
  printf("The product of your two numbers is %d\n", result);
}

int mult(int x, int y) // Definition of mult
{
  return x * y;
}
```

The compiler needs information about the functions:

- In `/usr/include/stdio.h` there is a prototype for `printf`.
- The function `mult` is defined at the top of the example program.

Error message when the prototype declaration is missing

- Consider a file where `#include <stdio.h>` is missing.

```
int main() {
  printf("Just a simple text.\n");
}
```

- A compilation will give an error message:

```
sarek:$ make test

cc     test.c   -o test
test.c: In function ‘main’:
test.c:2:3: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
   printf("Just a simple text.\n");
   ^~~~~~
test.c:2:3: warning: incompatible implicit declaration of built-in function ‘printf’
test.c:2:3: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
```

To fix that:

```
#include <stdio.h>

int main() {
  printf("Just a simple text.\n");
}
```

and the compilation works OK:

```
sarek:$ make test

cc     test.c   -o test
```

Pointers

A pointer is an address where things can be stored.
Compare with a number of drawers. We can put the shirt in the third drawer.

- To declare pointers:

```
int *pointer1, *pointer2;
```

- To use pointers:

```
int main()
{
    int x = 51;       // A normal integer
    int *p;           // A pointer to an integer

    p = &x;           // Make p contain the address to x
    printf("Please enter a number: ");
    scanf( "%d", &x );        // Put a value in x (send the address to scanf)
    scanf( "%d", p);          // This is the same
    printf( "%d\n", *p );     // Note the use of "*p" to get the value
```

- To allocate memory for use:

```
int *ptr = malloc( sizeof(int) );
... // use the memory...
free (ptr); // and return it again to the system.
```

Arrays

- We could use arrays "x" and "y" to store positions and velocities of 64 particles

```
double x[64], y[64];
double vx[64], vy[64];
int i;
for (i = 0; i < 64; i++) {
  x[i] = x[i] + delta_t * vx[i]; // Step forward in time
  y[i] = y[i] + delta_t * vy[i];
  vx[i] = ...
```

- For a more flexible solution with "n" particles:

```
double *x, *y;
double *vx, *vy;
x = malloc(n * sizeof(double));
y = malloc(n * sizeof(double));
vx = malloc(n * sizeof(double));
for (i = 0; i < n; i++) {
  x[i] = x[i] + delta_t * vx[i]; // Step forward in time
  y[i] = y[i] + delta_t * vy[i];
  vx[i] = ...
```

- It is sometimes convenient to be able to initialize an array:

```
int fibo[8] = {1, 2, 3, 5, 8, 13, 21, 34};
```

Strings

Arrays of characters—strings—are used a lot.
They contain both the visible character and an end-of-string character, the NULL character.

- There is a special syntax for strings

```
      // Single quotes for characters and double quotes for strings.
char str[15] = {'A', ' ', 's', 'h', 'o', 'r','t', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0'};
char str[15] = "A short string";
      // To print out a string, use %s as a format specifier.
printf("%s", str);
```

- Quite a few functions in the C library work on strings:

```
strcmp(str1, str2); // case sensitive comparison for getting alphabetic order

// To check if arg is equal to "rho" we could do:
int check;
check = strcmp(arg, "rho");
if (check == 0)
  ....

// That if statement can instead be written with "!" which means "not"
if (!check)
  ...

// A different way to do the same thing
if (!strcmp(arg, "rho"))
  ....
```

Structures

It is often convenient to have a single name that refers to a group of a related values. We will use that for the parameters we use in our simulation program. They are put together in the struct data type:

```
typedef struct Par {
  int n;                // number of particles
  double rho;           // density,
  double t;             // temperature,
  double deltat;        // time step
} Par;
```

- When we have a *variable* of type "struct Par" the syntax is "par.n"

```
int main(int argc, char *argv[])
{
  Par par; // Here "par" is a variable of type "struct Par"
  par.n = 64;
  par.rho = 0.6;
  par.deltat = 0.01;

  read_args(&par, arg));
```

- but in most functions "par" is instead a pointer, and we write "par->n"

```
int read_args(Par *par, char *arg)
{
  if (!strcmp(arg, "N")) {
    par->n = strtol(s, NULL, 10); // This means string-to-long
    return 1;
  }

  if (!strcmp(arg, "rho")) {
    par->rho = strtod(s, NULL); // This is string-to-double
```

Program structure

```
void run_simulation(Par *par, double *atoms) {
  // 1 Initialization... 2 Equilibration... 3 Production run:
  for (iblock = 0; iblock < par->nblock; iblock++) {
    ...                 // step forward in time
    ...                 // measure!
  }
  // 4 Print out results
}

int read_args(Par *par, char *arg) {
  if (!strcmp(arg, "N")) {
    par->n = strtol(s, NULL, 10);
    return 1;
  }
  if (!strcmp(arg, "run")) {
    run_simulation(par, atoms);
    return 1;
  }
  return 0;
}

int main(int argc, char *argv[])
{
  Par par;                  // Initialize parameter struct
  par.rho = 0.6;
  for (iarg = 1; iarg < argc; iarg++)
    if (!read_args(&par, argv[iarg]))
      exit(EXIT_FAILURE);
  exit(EXIT_SUCCESS);
}
```

A flexible code

Simplest approach: use arrays x, y, vx, vy, fx, fy for positions and forces. For three dimensions, also include arrays z, vz, and fz.:
To calculate $\{\mathbf{F}\}$ from $\{\mathbf{r}\}$ we would need different functions in 2D and 3D but this is no good idea.

- 2D: `forces_from_pos(par, x, y, fx, fy)`,
- 3D: `forces_from_pos(par, x, y, z, fx, fy, fz)`,

The code uses a more flexible solution:

- The same code should work for both two and three (and higher) dimensions.
- Some programs use both **r** and **v**, other only use **r**.
- Introduce `df` $= d_f =$ the number of degrees of freedom per particle $= d$ or $2d$.

Store everything in array `atoms` that contains $N \times d_f$ values:

```
double *atoms;
atoms = malloc(par->n * par->df * sizeof(double));
```

Variables for the preprocessor:

- D = dimensionality, integer > 0,
- VEL — should be defined if the velocity variables are used.

Define `pos` and perhaps also `vel`:

```
  double *pos = atoms;
#ifdef VEL
  double *vel = atoms + par->n * D;          // The second half of the array
#endif
```

In 2D: `(pos[0], pos[1], pos[2], pos[3]...)` $= x_0, y_0, x_1, y_1, \ldots$
In 3D: `(pos[0], pos[1], pos[2], pos[3], pos[4]...)` $= x_0, y_0, z_0, x_1, y_1, \ldots$
Generally speaking, $x_i =$ `pos[D * i]` and $y_i =$ `pos[D * i + 1]`.
...and in three dimensions: $z_i =$`pos[D * i + 2]`

Arrays and pointers

- Consider a function with two arguments: number of particles and an array with positions:

```
void do_nothing(int n, double *pos) {
  for (i = 0; i < n; i++) {
    double *ipos;
    ipos = pos + D * i;
    ...
  }
}
```

There are then several ways to access the x coordinate of particle i:

- `pos[D * i]`
- `ipos[0]` — use the pointer `ipos` which points to the memory where the coordinates of particle i are stored
- `*(pos + D * i)` — with "pointer arithmetics"

- A confusing detail:

```
// Short form of writing:
double *ipos = pos + D * i;
// Here *ipos is not dereferencing ipos,
// instead consider '*' to be a part of the type declaration
// ipos is of type "double *"
```

Dynamics

The Langevin dynamics,

$$\dot{\mathbf{v}}_i = \mathbf{F}_i - \alpha \mathbf{v}_i + \boldsymbol{\zeta}_i,$$

is implemented by adding to the existing velocity to get the new velocity,

$$\mathbf{v}_i + [\mathbf{F}_i - \alpha \mathbf{v}_i + \boldsymbol{\zeta}_i]\Delta_t \to \mathbf{v}_i.$$

The function `step(par, atoms, force)` in `common.c` calls functions for the dynamics:

Langevin dynamics:

1. `forces_from_pos(par, pos, force)` — calculate $\{\mathbf{F}\}$ from $\{\mathbf{r}\}$,
2. `langevin_forces(par, vel, force)` — add the Langevin terms to $\{\mathbf{F}\}$,
3. `vel_from_force(par, vel, force)` — step forward: $\mathbf{v}_i + \mathbf{F}_i\Delta_t \to \mathbf{v}_i$,
4. `pos_from_vel(par, pos, vel)` — new position: $\mathbf{r}_i + \mathbf{v}_i\Delta_t \to \mathbf{r}_i$.

Brownian dynamics— $\mathbf{r}_i + [\mathbf{F}_i/\alpha + \boldsymbol{\eta}_i]\Delta_t \to \mathbf{r}_i$,

1. `forces_from_pos(par, pos, force)` — calculate $\{\mathbf{F}\}$ from $\{\mathbf{r}\}$,
2. `pos_from_force(par, pos, force)` — new $\{\mathbf{r}\}$ from $\{\mathbf{F}\}$ and random noise.

Force calculations

The functions behind the force calculations are:

- `double distance(double L, double r1, double r2)`
  the one-dimensional distance, using periodic boundary conditions.
  Here `r1` and `r2` are the coordinates of particle 1 and 2, e.g. x1 and x2.
- `double dist2(double *L, double *p1, double *p2, double *dist)`
  returns the distance squared and the vector `dist`, when peridic boundary conditions are considered.
  Here `p1` and `p2` are pointers to the position vectors,
- `force_magnitude(double r2)`
  calculates the magnitude of the force based on the distance squared between two particles using the Lennard-Jones interaction.
- `void one_force(f, r2, dist)`
  calculates the force vector.
- `forces_from_pos`
  calculates $\{\mathbf{F}\}$ with a double loop over i and j

Header files

By itself the C language doesn't contain much and it is therefore necessary to get access to external library functions. For the compiler to know about these functions they need to be declared in some header files and this is done through statements as below: (Files in `/usr/include`.)

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

There are also often user-defined header files:

```
#include "define.h"
#include "sim.h"
```

They typically contain statements like

```
extern void one_force(double *f, double r2, double *dist);
```

which are needed by the compiler if the code is split into more than one source file.

Compiling with make

The most common way to compile in a Linux system is by just typing "make" or "make program-name".

The make program uses a file called `Makefile` in order to know what to do. Key statements in the Makefile are dependencies which can look like

```
sim: sim.o ran.o common.o config.o
```

which means that `sim` depends on "sim.o ran.o common.o config.o" and therefore needs to be regenerated (in some way) if any of the ".o"-files has been made more recently.

. . . compiling with make

There are two different ways to use "make":

1. Include the command that should be used to generate `sim` from the ".o"-files in the Makefile. (And also some commands to generate the ".o"-files from the ".c"-files):

```
sim: sim.o ran.o common.o config.o
	gcc sim.o ran.o common.o config.o -lm  -o sim
```

2. To make use of the built-in knowledge of "make". One then just needs to specify some flags and the dependencies.

```
CFLAGS = -g -O3
CPPFLAGS = -I.
LOADLIBES = -lm
sim: sim.o ran.o common.o config.o
```

The meaning of these flags are, shortly:

- `CFLAGS` — Flags for compilation. Here `-g` means to generate information for the debugger, `-O3` for optimisation, level 3.
- `CPPFLAGS`— Preprocessor flags. (The preprocessor handles things like `#include` and `#ifdef`.) Here `-I.` specifies that the preprocessor should look for files at "." which is the present directory.
- `LOADLIBES` — which libraries to load in the linking stage. Here `-lm` means to try to access the math library, in `libm.so`. `-labc` would mean `libabc.so`.

The LabStoch directory tree

This information is peculiar to the computer lab in the ModSim course.

After executing

```
$ mkdir LabStoch
$ cd LabStoch
$ wget www.tp.umu.se/modsim/files/LabStoch.tgz
$ tar xzf LabStoch.tgz
```

you are left with a directory tree with directories

```
src/    lang/    brown/    mc/
```

the `src` directory contains the source:

```
src/sim.h   src/common.c   src/ran.h   src/ran.c   src/config.c   src/sim.c
```

The other directories (`lang`, `brown`, and `mc`) should have their own `define.h` and `Makefile`:

```
lang/efile/   lang/conf/0064_r0.500_T1.000_start   lang/Makefile   lang/define.h
```

Files in directory `conf` store configurations; coordinates for one particle per line.

Makefile and define.h

The idea is to be able to use a single source to get different programs.
For Langevin dynamics—directory `lang`—the file define.h contains

```
#define D 2
#define VEL
#define CUT 3
```

and the Makefile refers to the source directory through `VPATH = ../src`:

```
CFLAGS = -g -O3
CPPFLAGS = -I.
LOADLIBES = -lm
VPATH = ../src

OBJS = sim.o ran.o common.o config.o

sim: ${OBJS}

${OBJS}: Makefile sim.h define.h ran.h
```

Note the variable OBJS which is used to keep track of the object files that will be linked to make up the executable program `sim`.

The *built-in* rule is that `make` runs the compiler to produce e.g. an updated `common.o` if `common.c` has been changed.

The last line in the file tells `make` that `common.o` *also* depends on `Makefile` and the header files and will be recompiled if any of these is more recent than `common.o`.

More on the read_args function

A string in C is an array of characters terminated by a null character.

- `strchr` returns pointer to the desired character or NULL,
- `strcmp` compares the lexical order. Returns 0 if equal.
- Also `strstr`, `strcat`, `strlen`...

```
int read_args(Par *par, char *arg)
{
  static double *atoms = NULL;
  char *s;
                    // strchr may e.g. be called with arg="read=0064_start"
  s = strchr(arg, '=');
  if (s)            // If '=' was found...
    *s++ = '\0';    // put end-of-string and let s point at the char after '='

  if (!strcmp(arg, "read")) {
    atoms = read_conf(par, atoms, s);
```

After the manipulations with the pointer `s` (`*s++ = '\0'`): `arg="read"` `s="0064_start"`