Computer simulations in statistical physics
Homework 5: Molecular dynamics simulations

Usage and functionality of program MD_LJ

First analysis of program output (default parameters)

Code of program MD_LJ

Usage and functionality of program MD_LJ

What can the program do, what are the parameters?

```
WS2006_Simulation/HW5_MD> ./MD_LJ -h
#*********************************************************#
# MD_LJ: molecular dynamics program for Lennard-Jones atoms  #
# Version: 0.6, 22.12.2006 by Nils Bluemer, compiled for D=3 #
# For help on usage and available options: run MD_LJ -h      #
#*********************************************************#
# options: -h  this help                                     #
#          -v   increase verbosity                           #
#         (-n#  number of particles - set automatically)     #
#          -l*  lattice type for init: (h)ypercubic or (f)cc #
#          -s#  system size (# particles/row) | default: 8   #
#          -r$  density                       | default: 0.600 #
#          -e$  energy per particle           | default: 0.500 #
#          -w$  warmup time                   | default: 0.200 #
#          -t$  simulation time               | default: 2.000 #
#          -d$  timestep                      | default: 0.002 #
#          -c$  cutoff (of potential/force)   | default: 2.500 #
#          -b#  number of bins in paircorr    | default: 200   #
# symbol key for parameters: # integer, $ real, * character  #
#*********************************************************#
```

Explanation of input parameters

-v increase verbosity: print state vectors after initialization and at end of run

-l lattice type for initialization: hypercubic or (hyper)fcc (default: fcc)
hypercubic: atoms on cartesic lattice in D dimensions (linear, square, simple cubic, . . .)
(hyper)fcc: atoms on every second site of hypercubic lattice (even taxi-cab distance from origin)

-s system size: integer extent of base lattice in each direction
$\rightsquigarrow$ number of atoms is s^D for hypercubic, $s^D/2$ for fcc
determines finite-size effects

-r density (in natural units)
1st physical parameter; density and number of atoms $\rightsquigarrow$ unit distance

-e energy per particle
2nd physical parameter; enforced at initialization, but nearly constant during simulation

-c cutoff of potential and forces (in natural units)
determines accuracy of Hamiltonian; must be smaller than half linear box size

-d time step (in natural time units)
determines accuracy of MD integration

-t total simulation time (in natural time units, not MD steps!)
determines "statistical" error of averages; must be much larger than autocorrelation times

-w warm-up time (in natural time units, not MD steps!)
determines convergency error; must be larger than "melting" or reordering times

-b number of bins in pair correlation function

Explanation of output (default parameters)

When started, the program first prints a header identifying itself (version, date, author, and dimension from compiler directives):

```
WS2006_Simulation/HW5_MD> ./MD_LJ | more
#*********************************************************#
# MD_LJ: molecular dynamics program for Lennard-Jones atoms  #
# Version: 0.6, 22.12.2006 by Nils Bluemer, compiled for D=3 #
#                                                            #
# For help on usage and available options: run MD_LJ -h      #
#*********************************************************#
```

It then states all parameters used in the simulation:

```
# numparts = 256, densisty = 0.600000, length = 7.528288
# twarmup = 0.200000, simtime = 2.000000, timestep = 0.002000
# E_N = 0.500000, cutoff = 2.50, corrbins = 200, lattice = f
```

Then follows (for usual run, i.e. without the option -v) the primary output: all observables are printed at each time step in the simulation (negative times indicate warmup)

```
# time        Epot/N     Ekin/N     Etot/N    temperature   pressure
 -0.2000    -4.404346   4.904346   0.500000     3.269564    -2.266062
 -0.1980    -4.404603   4.904603   0.500000     3.269735    -2.265327
 -0.1960    -4.405370   4.905369   0.499999     3.270246    -2.263113
...
```

Finally, the pair correlation function is printed (based on all positions observed after warm-up in the simulation):

```
## # distance  paircorr(distance)
## 0.009410  0.000000
## 0.028231  0.000000
...
## 0.818701  0.000000
## 0.837522  0.000862
## 0.856343  0.006825
## 0.875164  0.031667
## 0.893984  0.126551
...
```

In verbose mode (option -v), the full states [number of particle, position (D coordinates), velocity (D coordinates), force=acceleration (D coordinates)] are given directly after initialization and at the end of the simulation:

```
# time        Epot/N     Ekin/N     Etot/N    temperature   pressure
 -0.2000    -4.404346   4.904346   0.500000     3.269564    -2.266062

1   0.471   0.471   0.471      -1.094  -2.233   2.277      -0.000  -0.000  -0.000
2   2.353   0.471   0.471       2.312   0.220  -0.951      -0.000  -0.000  -0.000
3   4.235   0.471   0.471      -1.595  -0.542   1.868       0.000  -0.000  -0.000
...
```

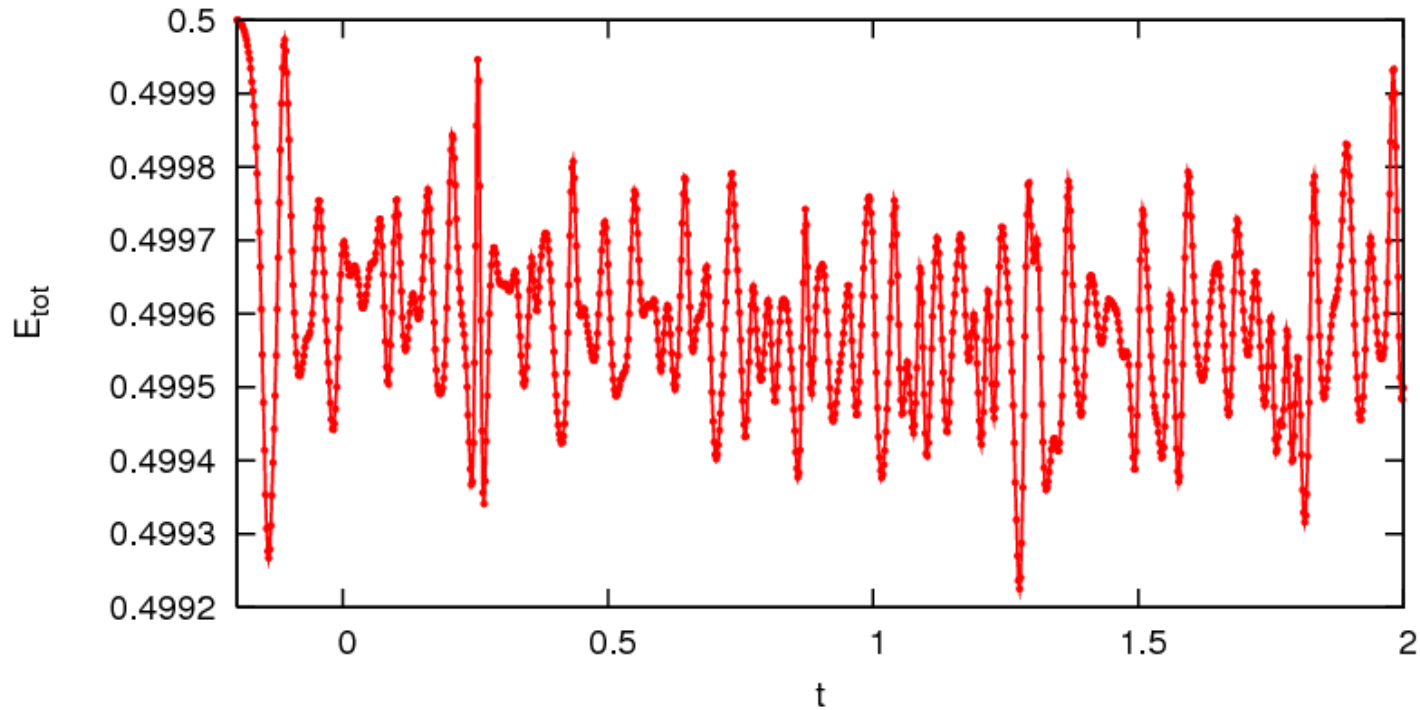
First analysis of program output (default parameters)

Step 1: run simulation and save output

```
WS2006_Simulation/HW5_MD> ./MD_LJ > results_MD_LJ_default.dat
```

Step 2: analyze time-dependent observables

All non-comment lines show observables as a function of time. Let's first check the total energy:



E_{tot} stays constant within about 10^{-3}

$\overline{E_{tot}} = 0.49958 \pm 0.00001$

Significant autocorrelation time (after warm-up): 7 time steps, i.e. $\tau \approx 0.014$ in natural units

How were the plot and the numbers obtained?

The plot was generated using gnuplot (http://www.gnuplot.info/):

```
WS2006_Simulation/HW5_MD> gnuplot results_MD_LJ_energy.gnu
```

with the input file

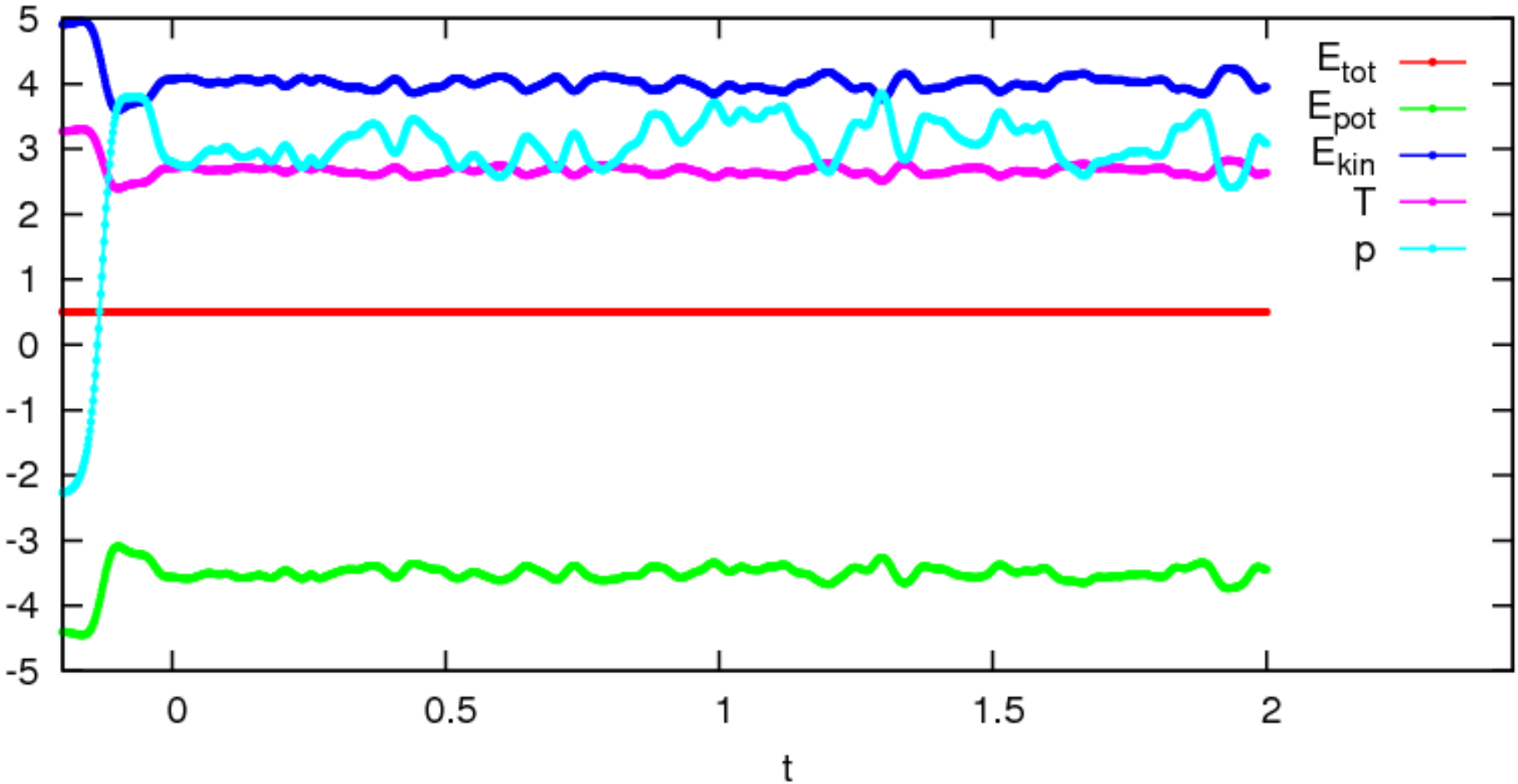
```
set term post eps enh color solid 12
set out "results_MD_LJ_energy.eps"
set size 0.7,0.5

set xlabel "t"
set ylabel "E_{tot}"
set nokey

plot [-0.2:2] "results_MD_LJ_default.dat" u 1:4 w lp lw 2 pt 7 ps 0.4
```

Here, the important part is plot [-0.2:2] "results_MD_LJ_default.dat" u 1:4 (u as abbreviation for "using" $\rightsquigarrow$ 1st and 4th column are plotted)

The data was analyzed using the tool awk (http://www.gnu.org/software/gawk/gawk.html) and "our" usual statistics tool:

```
HW5_MD> grep -v "#" results_MD_LJ_default.dat | awk '{if ($1>=0) print $4}' | stats -a
Average: 0.49959043, variance: 1.4889776e-08, error: 3.679152e-06
Korrelation time:  8.3545387, corrected error: 1.0634302e-05, transient: -8.4977902e-08
```

Now, let's look at all energy observables:



Explain the lead-in behavior, in particular the initially negative pressure!

Is the warm-up time well chosen?

Code of program MD_LJ

Program header

```
#define PROGNAME "MD_LJ"
#define VERSION "0.6"
#define DATE  "22.12.2006"
#define AUTHOR "Nils Bluemer"

/* project: molecular dynamics program for Lennard-Jones system */

/* new: 0.3: cutoff, full interface
        0.4: pair correlation function
        0.5: pressure, warm-up with negative times, tail corrections
             for potential and pressure, code clean-up */
        0.6: fcc lattice (alternative to hypercube), numlin->systemsize

/* todo: velocity distribution,
         binning for pair correlation function (-> error analysis),
   extra files for correlation function, velocity distribution etc.,
   make dimension regular variable, implement D-general volume,
   make computation of Epot more efficient (together with force)
         next major version: neighbor lists */
```

```c
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "pointer_utils.c"      /* vector-definitions */
#define BUFSIZE 100      /* used for reading command line */
#define EPSILON 0.000001 /* used for end time in time loop */

#define DIMENSION 3          /* todo: introduce dimension variable */

#define DEF_verbosity 0
#define DEF_systemsize 8
#define DEF_simtime 2.0
#define DEF_twarmup 0.2
#define DEF_timestep 0.002
#define DEF_density 0.6
#define DEF_E_N 0.5
#define DEF_cutoff 2.5
#define DEF_corrbins 200
#define DEF_lattice 'f'
```

main program

```
int main (int argc, char *argv[]) {
  char c, lattice;
  double **position, **velocity, **force;
  unsigned long *paircorr, ncollect;
  double time,simtime,timestep,twarmup;
  double density, length, length2, cutoff, cutoffsq;
  double Epot, energy, E_N, corrfac, virial;
  unsigned long numparts;
  unsigned int systemsize, corrbins;
  unsigned int verbosity;

  /* default values for parameters */
  verbosity = DEF_verbosity;
  systemsize    = DEF_systemsize;
  simtime   = DEF_simtime;
  twarmup   = DEF_twarmup;
  timestep  = DEF_timestep;
  density   = DEF_density;
  E_N       = DEF_E_N;
  cutoff    = DEF_cutoff;
  corrbins  = DEF_corrbins;
  lattice   = DEF_lattice;
```

```
/* parse command line (read in parameters) */
while (--argc > 0 && (*++argv)[0] == '-')
  while (c= *++argv[0])
    switch (c) {
    case 'h':  printhelp(); exit(0); break;
    case 'n':  sscanf(++argv[0],"%ld\n",&numparts);   error("change of numparts
not yet implemented"); break;
    case 'r':  sscanf(++argv[0],"%lf\n",&density); break;
    case 'e':  sscanf(++argv[0],"%lf\n",&E_N); break;
    case 't':  sscanf(++argv[0],"%lf\n",&simtime); break;
    case 'd':  sscanf(++argv[0],"%lf\n",&timestep); break;
    case 'w':  sscanf(++argv[0],"%lf\n",&twarmup); break;
    case 'c':  sscanf(++argv[0],"%lf\n",&cutoff); break;
    case 's':  sscanf(++argv[0],"%d\n",&systemsize); break;
    case 'l':  sscanf(++argv[0],"%c\n",&lattice); break;
    case 'v':  verbosity++; break;
    }
```

```
/* compute derived parameters */
if (lattice=='h')
  numparts=powD(systemsize);
else { /* lattice == 'f */
  if (systemsize%2==1)
    error ("fcc lattice only for even systemsize");
  numparts = powD(systemsize)/2;
}
length=pow(numparts/density,1.0/DIMENSION);
length2=0.5*length;
energy=E_N*numparts;
cutoffsq=cutoff*cutoff;
if (cutoff>length2)
  error("box too small for cutoff; decrease density or increase particle number");
```

```
/* first output */
print_header ();
printf ("# numparts = %ld, densisty = %f, length = %f\n", numparts, density, length);
printf ("# twarmup = %f, simtime = %f, timestep = %f\n", twarmup, simtime, timestep);
printf ("# E_N = %f, cutoff = %4.2f, corrbins = %d, lattice = %c\n", E_N, cutoff, corrbins,
lattice);
printf ("#\n");
print_obs_step (numparts,velocity,virial,Epot,length,time,cutoff,density,0);

/* set up pointers, init binned observables */
position = dmatrix(1,numparts,1,DIMENSION);
velocity = dmatrix(1,numparts,1,DIMENSION);
force = dmatrix(1,numparts,1,DIMENSION);
paircorr = lvector(0,corrbins); /* component 0 holds number of bins */
init_paircorr (paircorr,corrbins);

/* initialize positions and variables */
init_positions(numparts,position,length,systemsize,lattice);
Epot=Epot_fun(numparts,position,length2,cutoff,density);
if (energy<Epot)
  error("energy too low (for hypercubic setup) -> negative Ekin");
init_velocities(numparts,velocity,energy-Epot);
ncollect=0;
```

```
/* initialize simulation */
time=-twarmup;
compute_forces(numparts,position,force,length2,cutoffsq,&virial,paircorr,time);

/* compute and print observables for initial state */
print_obs_step (numparts,velocity,virial,Epot,length,time,cutoff,density,1);
if (verbosity>0)
  print_pos_vel_force(numparts,position,velocity,force);

/* start of simulation loop */
for (time=time+timestep;time<=simtime+EPSILON;time+=timestep){
  if(time>=0)
    ncollect++; /* number of measurements (e.g. for paircorr) */
  /* advance system by one time step */
  velocity_verlet(numparts,position,velocity,force,timestep,length,cutoffsq,
    &virial,paircorr,time);
  Epot=Epot_fun(numparts,position,length2,cutoff,density);
  print_obs_step (numparts,velocity,virial,Epot,length,time,cutoff,density,1);
} /* end of simulation loop */
```

```
  /* print collected observables (e.g. pair correlation) */
  if (verbosity>0)
    print_pos_vel_force(numparts,position,velocity,force);
  corrfac=2.0/3.0*M_PI*ncollect*numparts*density*powD(length2);
  if (DIMENSION!=3)
    warning("incorrect scale in pair correlation function!");
  print_paircorr(paircorr,length2,corrfac);

  /* finish up */
  freevectors(position,velocity,force,paircorr,numparts);
  return(0);
}
```

```
double powD(double x) /* computes x^DIMENSION */
{
  unsigned int d;
  double powD;
  powD=x;
  for (d=2;d<=DIMENSION;d++)
    powD*=x;
  return(powD);
}

/* potential enters here!!!!!!! */
double pairpot (double r2)
{
  double rinv6;
  rinv6=1.0/(r2*r2*r2);
  return (4.0*rinv6*(rinv6-1.0));
}
double forcefac (double r2)
{
  double rinv6;
  rinv6=1.0/(r2*r2*r2);
  return (48.0*rinv6*(0.5-rinv6)/r2);
}
```

```
double Epot_fun (unsigned long numparts, double **position, double length2,
   double cutoff, double density)
{
  unsigned long n,m;
  double distsq,sum,*vec,cutoffsq,cutoffc,offset;
  cutoffsq=cutoff*cutoff;
  cutoffc=cutoffsq*cutoff;
  sum=0.0;
  vec = dvector(1,DIMENSION);
  offset=pairpot(cutoffsq);
  for (n=1;n<=numparts;n++)
    for (m=n+1;m<=numparts;m++){
      comp_rel(length2,position[n],position[m],vec,&distsq);
      if (distsq<cutoffsq)
sum+=pairpot(distsq)-offset;
    }
  sum = sum -8.0*M_PI*numparts/3*density/cutoffc; /* tail correction */
  sum = sum -8.0*M_PI*numparts/3*density*(1.0/cutoffc - 1.0/(cutoffc*cutoffc));
                                /* correction for offset of pair potential */
  free_dvector(vec,1,DIMENSION);
  return(sum);
}
```

```
/* computes relative vector r2-r1 (in minimum image convention) */
void comp_rel (double length2, double *vec1, double *vec2, double *relvec,
        double *distsq)
{
  unsigned int d;
  double dx;
  *distsq=0.0;
  for (d=1;d<=DIMENSION; d++){
    dx=vec2[d]-vec1[d];
    if (dx<-length2)
      dx+=2.0*length2;
    else if (dx>length2)
      dx-=2.0*length2;
    relvec[d]=dx;
    *distsq+=dx*dx;
  }
}
```

```
double Ekin_fun (int numparts, double **velocity)
{
  unsigned int d,n;
  double x;
  x=0.0;
  for (n=1;n<=numparts;n++)
    for (d=1;d<=DIMENSION;d++)
      x+=velocity[n][d]*velocity[n][d];
  return(0.5*x);
}
```

```
/* place atoms on regular grid */
void init_positions (int numparts, double **position, double length, int systemsize, char lattice) {
  int n,m,k,d, *ivec, fac, sum;
  if (lattice =='h')
    fac=1;
  else /* lattice == 'f' */
    fac=2;
  n=0;
  ivec=ivector(1,DIMENSION);
  for (m=0;m<fac*numparts;m++){
    k=m;
    ivec[1]=k%systemsize;
    sum=ivec[1];
    for (d=2;d<=DIMENSION;d++){
      k=k/systemsize;
      ivec[d]=k%systemsize;
      sum+=ivec[d];}
    if (sum%fac==0){
      n++;
      for (d=1;d<=DIMENSION;d++)
        position[n][d]=(0.5+ivec[d])*length/systemsize;
    }}
  free_ivector(ivec,1,DIMENSION);}
```

```
/* initialize velocities for target kinetic energy (in 2-step process) */
void init_velocities (int numparts, double **velocity, double Ekin_target)
{
  int n, d;
  double Ekin, *dv, fac;
  dv=dvector(1,DIMENSION);

  /* Seed the random number generator */
  srand48(time(0) + getpid());

  /* first step: unit random velocities */
  for (d=1;d<=DIMENSION;d++){
    dv[d]=0.0;
    for (n=1;n<=numparts;n++){
      velocity[n][d]=drand48()-0.5;
      dv[d]+=velocity[n][d];
    }
    dv[d]=-1.0*dv[d]/numparts;
  }

  /* second step: ensure zero total momentum */
  Ekin=0.0;
  for (n=1;n<=numparts;n++)
```

```
    for (d=1;d<=DIMENSION;d++){
      velocity[n][d]+=dv[d];
      Ekin+=velocity[n][d]*velocity[n][d];
    }
  Ekin=0.5*Ekin;

  /* third step: rescale velocities */
  fac=sqrt(Ekin_target/Ekin);
  for (n=1;n<=numparts;n++)
    for (d=1;d<=DIMENSION;d++)
      velocity[n][d]*=fac;

  free_dvector(dv,1,DIMENSION);
}

void init_paircorr (unsigned long *paircorr, int corrbins)
{
  unsigned int i;
  paircorr[0]=corrbins;
  for (i=1;i<=corrbins;i++)
    paircorr[i]=0;
}
```

```
void update_paircorr (unsigned long *paircorr, double x)
{
  unsigned long bin;
  double xrel;
  xrel=sqrt(x);
  bin=(unsigned long)(xrel*paircorr[0])+1;
  paircorr[bin]++;
}

void print_paircorr (unsigned long *paircorr, double length2, double corrfac) {
  int bin;
  int corrbins;
  double binlD, binhD, invcorrbins;
  corrbins=paircorr[0];
  invcorrbins=1.0/corrbins;
  printf ("## # distance  paircorr(distance)\n");
  for (bin=1;bin<=corrbins;bin++){
    binlD=powD((bin-1)*invcorrbins);
    binhD=powD(bin*invcorrbins);
    printf("## %f  %f\n",(bin-0.5)/corrbins*length2,
    paircorr[bin]/(corrfac*(binhD-binlD)));
  }
}
```

```
void compute_forces (unsigned long numparts, double **position, double **force,
      double length2, double cutoffsq, double *virial,
      unsigned long *paircorr, double time)
{
  unsigned int d;
  unsigned long n,m;
  double *vec,distsq,length2sq,factor;
  vec = dvector(1,DIMENSION);
  length2sq=length2*length2;
  *virial=0.0;
  for (n=1;n<=numparts;n++)
    for (d=1;d<=DIMENSION;d++)
      force[n][d]=0.0;
  for (n=1;n<=numparts;n++)
    for (m=n+1;m<=numparts;m++){
      comp_rel(length2,position[n],position[m],vec,&distsq);
      if (distsq<cutoffsq){
        factor=forcefac(distsq);
for (d=1;d<=DIMENSION;d++){
   force[n][d]+= factor*vec[d];
   force[m][d]-= factor*vec[d];
}
        *virial-=factor*distsq;
```

```
    }
    if ((distsq<length2sq)&&(time>0))
update_paircorr(paircorr,distsq/length2sq);
  }
 free_dvector(vec,1,DIMENSION);
```

```
void velocity_verlet (unsigned long numparts, double **position, double **velocity,
        double **force, double timestep, double length,
        double cutoffsq, double *virial, unsigned long *paircorr,
        double time)
{
  int n,d;
  double ts2;
  ts2=timestep*timestep;
  for (n=1;n<=numparts;n++)
    for (d=1;d<=DIMENSION;d++){
      position[n][d]+=velocity[n][d]*timestep+0.5*force[n][d]*ts2;
      if (position[n][d]<0)
position[n][d]+=length;
      else if (position[n][d]>length)
position[n][d]-=length;
      velocity[n][d]+=0.5*timestep*force[n][d];
    }
  compute_forces (numparts,position,force,0.5*length,cutoffsq,virial,paircorr,time);
  for (n=1;n<=numparts;n++)
    for (d=1;d<=DIMENSION;d++)
      velocity[n][d]+=0.5*timestep*force[n][d];
}
```

```
void printvector (double *vec)
{
  unsigned int d;
  for (d=1;d<=DIMENSION;d++)
    printf("%6.3lf  ",vec[d]);
}

void print_pos_vel_force (unsigned long numparts, double **position,
     double **velocity, double **force)
{
  unsigned long n;
  for (n=1;n<=numparts;n++){
    printf("%ld  ",n);
    printvector(position[n]);
    printf("      ");
    printvector(velocity[n]);
    printf("      ");
    printvector(force[n]);
    printf("\n");
  }
}
```

```
void print_obs_step (unsigned long numparts, double **velocity, double virial,
      double Epot, double length, double time, double cutoff,
      double density, unsigned int head)
{
 double Ekin, temperature, pressure,cutoff3inv;
 cutoff3inv=1.0/(cutoff*cutoff*cutoff);
 if (head==0) /* print header */
  printf("# time        Epot/N     Ekin/N     Etot/N    temperature   pressure\n");
 else {
  Ekin=Ekin_fun(numparts,velocity);
  temperature=2*Ekin/(DIMENSION*numparts);
  pressure=(numparts*temperature + virial/DIMENSION)/powD(length);
  pressure = pressure -16.0/3.0*M_PI*density*density*density*
    cutoff3inv*(1.0-cutoff3inv/1.5); /* tail corrections (for D=3)*/
  printf("%8.4f    %9.6f  %9.6f  %9.6f     %8.6f    %8.6f\n",
  time,Epot/numparts,Ekin/numparts,
 (Epot+Ekin)/numparts,temperature, pressure);
 }
}
```