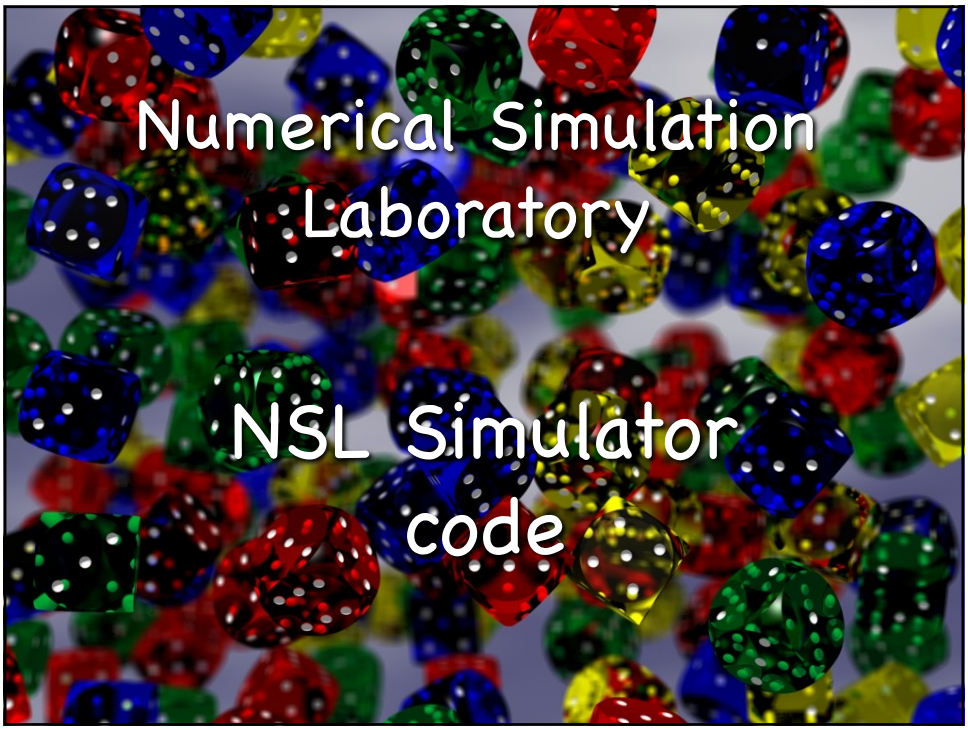


1

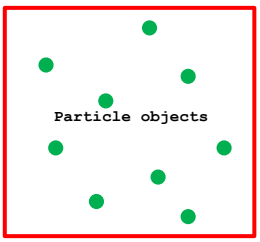
NSL Simulator code: system.h 1.

```
class System {

private:
  const int _ndim = 3;  // Dimensionality of the system
  bool _restart;        // Flag indicating if the simulation is restarted
  int _sim_type;        // Type of simulation (e.g., Lennard-Jones, Ising)
  int _npart;           // Number of particles
  int _nblocks;         // Number of blocks for block averaging
  int _nsteps;          // Number of simulation steps in each block
  int _nattempts;       // Number of attempted moves
  int _naccepted;       // Number of accepted moves
  double _temp, _beta;  // Temperature and inverse temperature
  double _rho, _volume; // Density and volume of the system
  double _r_cut;        // Cutoff radius for pair interactions
  double _delta;        // Displacement step for particle moves
  double _J, _H;        // Parameters for the Ising Hamiltonian
  vec _side;            // Box dimensions
  vec _halfside;        // Half of box dimensions
  Random _rnd;          // Random number generator
  field <Particle> _particle; // Field of particle objects representing the system
  vec _fx, _fy, _fz;    // Forces on particles along x, y, and z directions

  // Properties
  int _nprop; // Number of properties being measured
  bool _measure_penergy, _measure_kenergy, _measure_tenergy;// Flags for measuring different energies
  bool _measure_temp, _measure_pressure, _measure_gofr;     // Flags for measuring temp, pressure, radial dist. function
  bool _measure_magnet, _measure_cv, _measure_chi;          // Flags for measuring magnetization, specific heat, susceptibility
  int _index_penergy, _index_kenergy, _index_tenergy;       // Indices for accessing energy properties in vec _measurement
  int _index_temp, _index_pressure,  index_gofr;            // Indices for accessing temp, pressure, and radial dist. function
  int _index_magnet, _index_cv,  index_chi;                 // Indices for accessing magnetization, specific heat, susceptibility
  int _n_bins;          // Number of bins for radial distribution function
  double _bin_size;     // Size of bins for radial distribution function
  double _vtail, _ptail; // Tail corrections for energy and pressure
  vec _block_av;        // Block averages of properties
  vec _global_av;       // Global averages of properties
  vec _global_av2;      // Squared global averages of properties
  vec _average;         // Average values of properties
  vec _measurement;     // Measured values of properties
```

System object:
A field i.e. a vector of <Particles>
in a box with p.b.c.



2

2

NSL Simulator code: system.h 2.

```
public: // Function declarations

  int get_nbl();              // Get the number of blocks
  int get_nsteps();           // Get the number of steps in each block
  void initialize();          // Initialize system properties
  void initialize_properties();// Initialize properties for measurement
  void finalize();            // Finalize system and clean up
  void write_configuration(); // Write final system configuration to XYZ file
  void write_XYZ(int nconf);  // Write system configuration in XYZ format on the fly
  void write_velocities();    // Write final particle velocities to file
  void read_configuration();  // Read system configuration from file
  void initialize_velocities();// Initialize particle velocities
  void step();                // Perform a simulation step
  void block_reset(int blk);  // Reset block averages
  void measure();             // Measure properties of the system
  void averages(int blk);     // Compute averages of properties
  double error(double acc, double acc2, int blk); // Compute error
  void move(int part);        // Move a particle
  bool metro(int part);       // Perform Metropolis acceptance-rejection step
  double pbc(double position, int i); // Apply periodic boundary conditions for coordinates
  int pbc(int i);             // Apply periodic boundary conditions for spins
  void Verlet();              // Perform Verlet integration step
  double Force(int i, int dim); // Calculate force on a particle along a dimension
  double Boltzmann(int i, bool xnew); // Calculate Boltzmann factor for Metropolis acceptance

};
```

3

3

NSL Simulator code: main

```
#include <iostream>
#include "system.h"

using namespace std;

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

  int nconf = 1;
  System SYS;
  SYS.initialize();
  SYS.initialize_properties();
  SYS.block_reset(0);

  for(int i=0; i < SYS.get_nbl(); i++){ //loop over blocks
    for(int j=0; j < SYS.get_nsteps(); j++){ //loop over steps in a block
      SYS.step();
      SYS.measure();
      if(j%10 == 0){
//        SYS.write_XYZ(nconf); //Write actual configuration in XYZ format //Commented to avoid "filesystem full"!
        nconf++;
      }
    }
    SYS.averages(i+1);
    SYS.block_reset(i+1);
  }
  SYS.finalize();

  return 0;
}
```

4

4

NSL Simulator code: .initialize()

```
#include <iostream>
#include "system.h"


using namespace std;


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


  int nconf = 1;
  System SYS;
  SYS.initialize();
  SYS.initialize_properties();
  SYS.block_reset(0);


  for(int i=0; i < SYS.get_nbl(); i++){ //loop over blocks
    for(int j=0; j < SYS.get_nsteps(); j++){ //loop over steps in a block
      SYS.step();
      SYS.measure();
      if(j%10 == 0){
//        SYS.write_XYZ(nconf); //Write actual configuration in XYZ format //Commented to avoid "filesystem full"!
        nconf++;
      }
    }
    SYS.averages(i+1);
    SYS.block_reset(i+1);
  }
  SYS.finalize();


  return 0;
}
```

5

5

System :: initialize()

6

```
void System :: initialize(){ //Initialize the System object according to the content of the input files

  int p1, p2; // Read from ../INPUT/Primes a pair of numbers to be used to initialize the RNG
  ifstream Primes("../INPUT/Primes");
  Primes >> p1 >> p2 ;
  Primes.close();
  int seed[4]; // Read the seed of the RNG
  ifstream Seed("../INPUT/seed.in");
  Seed >> seed[0] >> seed[1] >> seed[2] >> seed[3];
  _rnd.SetRandom(seed,p1,p2);

  ofstream couta("../OUTPUT/acceptance.dat"); // Set the heading line in file ../OUTPUT/acceptance.dat
  couta << "#   N_BLOCK:   ACCEPTANCE:" << endl;
  couta.close();

  ifstream input("../INPUT/input.dat"); // Start reading ../INPUT/input.dat
  ofstream coutf;
  coutf.open("../OUTPUT/output.dat");
  string property;
  double mass, delta;
  while ( !input.eof() ){
    input >> property;
    if( property == "SIMULATION_TYPE" ){
      input >> _sim_type;
      if(_sim_type > 1){
        input >> _J;
        input >> _H;
      }
      if(_sim_type > 3){
        cerr << "PROBLEM: unknown simulation type" << endl;
        exit(EXIT_FAILURE);
      }
      if(_sim_type == 0)      coutf << "LJ MOLECULAR DYNAMICS (NVE) SIMULATION"  << endl;
      else if(_sim_type == 1) coutf << "LJ MONTE CARLO (NVT) SIMULATION"         << endl;
      else if(_sim_type == 2) coutf << "ISING 1D MONTE CARLO (MRT^2) SIMULATION" << endl;
      else if(_sim_type == 3) coutf << "ISING 1D MONTE CARLO (GIBBS) SIMULATION" << endl;
    } else if( property == "RESTART" ){
      input >> _restart;
```

Input.dat

```
SIMULATION_TYPE     1
RESTART             0
TEMP                1.1
NPART               108
RHO                 0.8
R_CUT               2.5
DELTA               0.1
NBLOCKS             20
NSTEPS              2000

ENDINPUT
```

... etc.etc. ...

6

NSL Simulator code: .initialize_properties()

```
#include <iostream>
#include "system.h"

using namespace std;

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

  int nconf = 1;
  System SYS;
  SYS.initialize();
  SYS.initialize_properties();
  SYS.block_reset(0);

  for(int i=0; i < SYS.get_nbl(); i++){ //loop over blocks
    for(int j=0; j < SYS.get_nsteps(); j++){ //loop over steps in a block
      SYS.step();
      SYS.measure();
      if(j%10 == 0){
//        SYS.write_XYZ(nconf); //Write actual configuration in XYZ format //Commented to avoid "filesystem full"!
        nconf++;
      }
    }
    SYS.averages(i+1);
    SYS.block_reset(i+1);
  }
  SYS.finalize();

  return 0;
}
```

7

7

System :: initialize_properties()

8

```
void System :: initialize_properties(){ // Initialize data members used for measurement of properties

  string property;
  int index_property = 0;
  _nprop = 0;

  _measure_penergy  = false; //Defining which properties will be measured
  _measure_kenergy  = false;
  _measure_tenergy  = false;
  _measure_pressure = false;
  _measure_gofr     = false;

//… etc.etc. …

  ifstream input("../INPUT/properties.dat");
  if (input.is_open()){
    while ( !input.eof() ){
      input >> property;
      if( property == "POTENTIAL_ENERGY" ){

//… etc.etc. …

      } else if( property == "GOFR" ){
        ofstream coutgr("../OUTPUT/gofr.dat");
        coutgr << "# DISTANCE:     AVE_GOFR:        ERROR:" << endl;
        coutgr.close();
        input>>_n_bins;
        _nprop+=_n_bins;
        _bin_size = (_halfside.min() )/(double)_n_bins;
        _measure_gofr = true;
        _index_gofr = index_property;
        index_property+= _n_bins;

//… etc.etc. …
```

properties.dat

```
TOTAL_ENERGY
POTENTIAL_ENERGY
KINETIC_ENERGY
TEMPERATURE
PRESSURE
GOFR                _n_bins
```

… etc.etc. …

8

NSL Simulator code: .step()

```
#include <iostream>
#include "system.h"

using namespace std;

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

  int nconf = 1;
  System SYS;
  SYS.initialize();
  SYS.initialize_properties();
  SYS.block_reset(0);

  for(int i=0; i < SYS.get_nbl(); i++){ //loop over blocks
    for(int j=0; j < SYS.get_nsteps(); j++){ //loop over steps in a block
      SYS.step();
      SYS.measure();
      if(j%10 == 0){
//        SYS.write_XYZ(nconf); //Write actual configuration in XYZ format //Commented to avoid "filesystem full"!
        nconf++;
      }
    }
    SYS.averages(i+1);
    SYS.block_reset(i+1);
  }
  SYS.finalize();
  return 0;
}
```

```
void System :: step(){ // Perform a simulation step
  if(_sim_type == 0) this->Verlet();  // Perform a MD step
  else for(int i=0; i<_npart; i++) this->move(int(_rnd.Rannyu()*_npart)); // Perform a MC step
  _nattempts += _npart; //update number of attempts performed on the system
  return;
}
```

9

System :: move(int)


```
void System :: move(int i){ // Propose a MC move for particle i
  if(_sim_type == 3){ //Gibbs sampler for Ising
    // TO BE FIXED IN EXERCISE 6
  } else {           // M(RT)^2
    if(_sim_type == 1){       // LJ system
      vec shift(_ndim);       // Will store the proposed translation
      for(int j=0; j<_ndim; j++){
        shift(j) = _rnd.Rannyu(-1.0,1.0) * _delta; // uniform distribution in [-_delta;_delta)
      }
      _particle(i).translate(shift, _side);  //Call the function Particle::translate
      if(this->metro(i)){ //Metropolis acceptance evaluation
        _particle(i).acceptmove();
        _naccepted++;
      } else _particle(i).moveback(); //If translation is rejected, restore the old configuration
    } else {                  // Ising 1D
      if(thi
        _par
        _nac
      }
    }
  }
  return;
}
```

```
bool System :: metro(int i){ // Metropolis algorithm
  bool decision = false;
  double delta_E, acceptance;
  if(_sim_type == 1) delta_E = this->Boltzmann(i,true) - this->Boltzmann(i,false);
  else …
  acceptance = exp(-_beta*delta_E);
  if(_rnd.Rannyu() < acceptance ) decision = true; //Metropolis acceptance step
  return decision;
}
```

```
double System :: Boltzmann(int i, bool xnew){
  double energy_i=0.0;
  double dx, dy, dz, dr;
  for (int j=0; j<_npart; j++){
    if(j != i){
      dx = this->pbc(_particle(i).getposition(0,xnew) - _particle(j).getposition(0,1), 0);
      dy = this->pbc(_particle(i).getposition(1,xnew) - _particle(j).getposition(1,1), 1);
      dz = this->pbc(_particle(i).getposition(2,xnew) - _particle(j).getposition(2,1), 2);
      dr = dx*dx + dy*dy + dz*dz;
      dr = sqrt(dr);
      if(dr < _r_cut) energy_i += 1.0/pow(dr,12) - 1.0/pow(dr,6);
    }
  }
  return 4.0 * energy_i;}
```

10

NSL Simulator code: .measure()

```
#include <iostream>
#include "system.h"


using namespace std;


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


  int nconf = 1;
  System SYS;
  SYS.initialize();
  SYS.initialize_properties();
  SYS.block_reset(0);


  for(int i=0; i < SYS.get_nbl(); i++){ //loop over blocks
    for(int j=0; j < SYS.get_nsteps(); j++){ //loop over steps in a block
      SYS.step();
      SYS.measure();
      if(j%10 == 0){
//        SYS.write_XYZ(nconf); //Write actual configuration in XYZ format //Commented to avoid "filesystem full"!
        nconf++;
      }
    }
    SYS.averages(i+1);
    SYS.block_reset(i+1);
  }
  SYS.finalize();


  return 0;
}
```

11

11

System :: measure()

12

```
void System :: measure(){ // Measure properties
  _measurement.zeros();
  // POTENTIAL ENERGY, VIRIAL, GOFR ////////////////////////////////////////////
  int bin;
  vec distance;
  distance.resize(_ndim);
  double penergy_temp=0.0, dr; // temporary accumulator for potential energy
  double kenergy_temp=0.0; // temporary accumulator for kinetic energy
  double tenergy_temp=0.0;
  double magnetization=0.0;
  double virial=0.0;
  if (_measure_penergy or _measure_pressure or _measure_gofr) {
    for (int i=0; i<_npart-1; i++){
      for (int j=i+1; j<_npart; j++){
        distance(0) = this->pbc( _particle(i).getposition(0,true) - _particle(j).getposition(0,true), 0);
        distance(1) = this->pbc( _particle(i).getposition(1,true) - _particle(j).getposition(1,true), 1);
        distance(2) = this->pbc( _particle(i).getposition(2,true) - _particle(j).getposition(2,true), 2);
        dr = sqrt( dot(distance,distance) );
        // GOFR ... TO BE FIXED IN EXERCISE 7
        if(dr < _r_cut){
          if(_measure_penergy)  penergy_temp += 1.0/pow(dr,12) - 1.0/pow(dr,6); // POTENTIAL ENERGY
          // PRESSURE ... TO BE FIXED IN EXERCISE 4
        }
      }
    }
  }

//... Etc.etc. ...

  _block_av += _measurement; //Update block accumulators

  return;
}
```

12

NSL Simulator code: .averages(int)

```
#include <iostream>
#include "system.h"


using namespace std;


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


  int nconf = 1;
  System SYS;
  SYS.initialize();
  SYS.initialize_properties();
  SYS.block_reset(0);


  for(int i=0; i < SYS.get_nbl(); i++){ //loop over blocks
    for(int j=0; j < SYS.get_nsteps(); j++){ //loop over steps in a block
      SYS.step();
      SYS.measure();
      if(j%10 == 0){
       // SYS.write_XYZ(nconf); //Write actual configuration in XYZ format //Commented to avoid "filesystem full"!
        nconf++;
      }
    }
    SYS.averages(i+1);
    SYS.block_reset(i+1);
  }
  SYS.finalize();


  return 0;
}
```

13

13

System :: averages(int)

14

```
void System :: averages(int blk){

  ofstream coutf;
  double average, sum_average, sum_ave2;

  _average    = _block_av / double(_nsteps);
  _global_av  += _average;
  _global_av2 += _average % _average; // % -> element-wise multiplication

  // POTENTIAL ENERGY /////////////////////////////////////////////////////////
  // KINETIC ENERGY ///////////////////////////////////////////////////////////
  // TOTAL ENERGY /////////////////////////////////////////////////////////////
  // TEMPERATURE //////////////////////////////////////////////////////////////
  // PRESSURE ////////////////////////////////////////////////TO BE FIXED IN EXERCISE 4
  // GOFR ////////////////////////////////////////////////TO BE FIXED IN EXERCISE 7
  // MAGNETIZATION ///////////////////////////////////////TO BE FIXED IN EXERCISE 6
  // SPECIFIC HEAT ///////////////////////////////////////TO BE FIXED IN EXERCISE 6
  // SUSCEPTIBILITY //////////////////////////////////////TO BE FIXED IN EXERCISE 6
  // ACCEPTANCE ///////////////////////////////////////////////////////////////
  double fraction;
  coutf.open("../OUTPUT/acceptance.dat",ios::app);
  if(_nattempts > 0) fraction = double(_naccepted)/double(_nattempts);
  else fraction = 0.0;
  coutf << setw(12) << blk << setw(12) << fraction << endl;
  coutf.close();

  return;
}
```

```
double System :: error(double acc, double acc2, int blk){
  if(blk <= 1) return 0.0;
  else return sqrt( fabs(acc2/double(blk) - pow( acc/double(blk) ,2) )/double(blk) );
}
```

14