Introducing periodic boundary conditions in a Molecular Dynamics simulation

Alexander Wagner

February 5, 2018

Abstract

We show how in a molecular simulation particles can be contained in a bulk-like environment by using periodic boundary conditions. In this report we focus on how such boundary conditions are implemented in detail in a Molecular Dynamics code.

1 Introduction

In this report we are focusing on simulations of molecular systems. The particles move under classical Newtonian dynamics and are represented as coordinates and velocities. Typically only a small part of such a system can be represented in a simulation. To simulate such a bulk system, it is important that particles in a simulation are contained. The most obvious solution would be to define a container that keeps the particles in place. However, doing so would introduce special boundary effects, that can differ substantially from bulk behavior. Periodic boundary conditions are an effective way of containing the particles without introducing boundary effects. The rest of the report is structured as follows: first we introduce the theory behind periodic boundary conditions, then we explain how this can be implemented in a code. We then demonstrate that the code manages to contain the particles. We also comment on situations in which this code can fail. We conclude that, as long as sensible parameters are chosen that will prevent very large jumps of particles, our code performs well. The full code is presented in an appendix.

2 Periodic boundary conditions

The idea behind the periodic boundary conditions are shown in Figure 1. The system is seen as extending to infinity, but in an entirely periodic manner. Particles interact not only within the simulation cell, but also with all periodic images of these particles. For this to be practical it is usually assumed that the interactions fall of quickly with distance, so that only one interaction with the

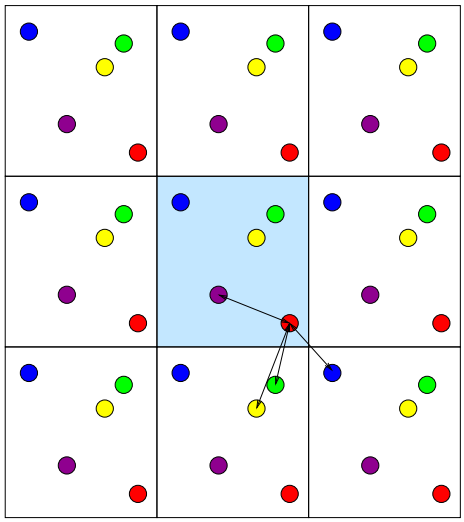

Figure 1: A sketch of the idea behind periodic boundary conditions. The modeled system consists only of the slightly darker middle square. However, particles will interact with the closest periodic image, in stead of the actual particle if those are closer.

closes image has to be taking into account. This also means that only one shell of periodic images around the original system has to be taken into account.

Our molecular dynamics code implements Newton's equations

$$\dot{\mathbf{x}}_i = \mathbf{v}_i \tag{1}$$

$$\dot{\mathbf{v}}_i = \frac{\mathbf{F}_i}{m} \tag{2}$$

where $\mathbf{x}_i$ is the position vector of the ith particle, and $\mathbf{v}_i$ is its velocity. The dot indicates a time derivative, as usual. We discretized this using the Verlet algorithm as

$$\mathbf{v}(t + \Delta t/2) = \mathbf{v}(t - \Delta t/2) + \frac{\mathbf{F}(t)}{m}\Delta t \tag{3}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \Delta t/2) \tag{4}$$

The force is given as a sum of pair forces as

$$\mathbf{F}_i = \frac{\partial V(\{\mathbf{x}_n\})}{\partial \mathbf{x}_i} \tag{5}$$

where the partial derivative here is the same as the nabla operator for the coordinates of the ith particle. V is the potential energy of the system, that depends on the coordinates of all the particles. An explicit implementation of this algorithm is given in the appendix.

Explicitly the function F in the code in the appendix calculates the distances in each dimension, and only retains information about the shortest distance:

```
double dr[D],dR=0;
for (int d=0; d<D; d++){
  dr[d]=x[m][d]-(x[n][d]-L);
  double ddr;
  ddr=x[m][d]-(x[n][d]);
  if (fabs(ddr)<fabs(dr[d])) dr[d]=ddr;
  ddr=x[m][d]-(x[n][d]+L);
  if (fabs(ddr)<fabs(dr[d])) dr[d]=ddr;
  dR+=dr[d]*dr[d];
}
```

where $dr[d]$ encodes the dth component of the displacement vector between the nth and the mth particle and dR is the absolute squared of the **dr** vector.

This code fragment shows how we determine the shortest distance between the nth and the mth particle and their periodic images in D dimensions. This shortest distance is then used in the usual force routine.

The second part of the code that needs to consider periodic boundary conditions is the part where the particle positions are updated. Since we now insist that the particles are contained in a domain from $x_{i\alpha} \in [0, L]$, where α refers to any of the dimensions considered in the code. When a particle leaves this domain one of its periodic images will enter the simulation domain at the same time. The position coordinates are now replaced with the coordinates of the periodic image that happens to reside inside the computational domain. In code, this is accomplished by

```
x[n][d]+=v[n][d]*dt;
if (x[n][d]<0) x[n][d]+=L;
else if (x[n][d]>=L) x[n][d]-=L;
```

This completes the implementation of periodic boundary conditions in our Molecular Dynamics code.

3 Results

We tested the algorithm by setting up particles in a regular two-dimensional lattice and giving the particles on the left side an upward velocity of 1 and the right side a downward velocity of -1. The results are shown in Fig. 2.

These simulations show that particles reenter at the boundaries of the simulation and that no spurious effects are obvious at the periodic boundaries.

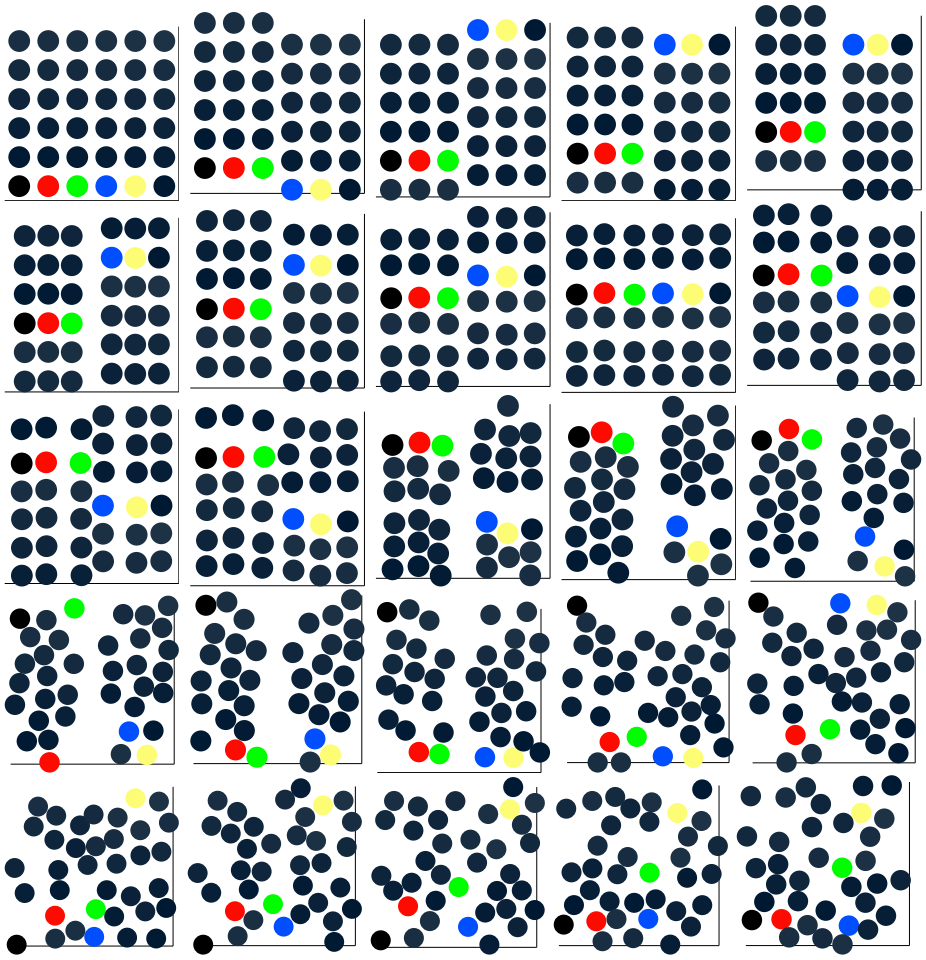

Figure 2: A sample output for a system with $L = 8$ and 36 particles arranged on a regular square lattice. The particles move in unison for a while until computational errors break the initial symmetry. The time-step was $\Delta t = 0.01$ and we show subsequent results after 50 iterations.

4 Conclusions

In this report we have shown how to implement periodic boundary conditions in a Molecular Dynamics simulation. We have verified that our algorithm is free from obvious errors by setting up a periodic system and observing that it shows no spurious edge effects.

A Code

```
#include <math.h>
#include <mygraph.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define Nmax 400
#define D 2
double  L=50; // size of box


double m[Nmax]; // charges of the particles
double x[Nmax][D],v[Nmax][D]; // State of the system

// parameters
double C[D],scalefac=100;
double k=1,x0[Nmax][D],v0[Nmax][D],dt=0.01,vv=1;
int N=Nmax,points=100,iterations=0;

void F(double x[N][D], double v[N][D],double FF[N][D]){

  memset(&FF[0][0],0,N*D*sizeof(double));
  for (int n=0; n<N; n++)
    for (int m=n+1; m<N; m++){
      double dr[D],dR=0;
      for (int d=0; d<D; d++){
        dr[d]=x[m][d]-(x[n][d]-L);
        double ddr;
        ddr=x[m][d]-(x[n][d]);
        if (fabs(ddr)<fabs(dr[d])) dr[d]=ddr;
        ddr=x[m][d]-(x[n][d]+L);
        if (fabs(ddr)<fabs(dr[d])) dr[d]=ddr;
        dR+=dr[d]*dr[d];
      }
      double dR6=dR*dR*dR;
```

```
      double dR12=dR6*dR6;
      double Fabs=12/dR12/dR-6/dR6/dR;
      for (int d=0;d<D; d++){
        FF[n][d]-=Fabs*dr[d];
        FF[m][d]+=Fabs*dr[d];
      }
    }
  return;
}

void iterate(double x[N][D],double v[N][D],double dt){
  double ff[N][D];
  F(x,v,ff);
  if (iterations==0)
    for (int n=0;n<N;n++)
      for (int d=0;d<D;d++)
        v[n][d]+=0.5*ff[n][d]/m[n]*dt;
  else
    for (int n=0;n<N;n++)
      for (int d=0;d<D;d++)
        v[n][d]+=ff[n][d]/m[n]*dt;

  for (int n=0;n<N;n++)
    for (int d=0;d<D;d++){
      x[n][d]+=v[n][d]*dt;
      if (x[n][d]<0) x[n][d]+=L;
      else if (x[n][d]>=L) x[n][d]-=L;
    }
  iterations++;
}

void CM(){
  double cm[D],cmv[D],M=0;
  memset(&cm[0],0,D*sizeof(double));
  memset(&cmv[0],0,D*sizeof(double));
  for (int n=0; n<N; n++) M+=m[n];
  for (int d=0; d<D; d++){
    for (int n=0; n<N; n++){
      cm[d]+=m[n]*x[n][d];
      cmv[d]+=m[n]*v[n][d];
    }
    cm[d]/=M;
    cmv[d]/=M;
  }
  for (int n=0; n<N; n++)
    for (int d=0; d<D; d++){
```

```
      x[n][d]-=cm[d]+L/2;
      v[n][d]-=cmv[d]+L/2;
    }

}

void setup(){
  int M=pow(N,1./D)+1;
  for (int n=0; n<N; n++){
    m[n]=1;
    for (int d=0; d<D; d++){
      int nn=n;
      for (int dd=0; dd<d; dd++) nn/=M;
      x0[n][d]=(nn%M)*L/M;
      if (d==1){
        if (x0[n][0]<L/2)
          v0[n][d]=vv;
        else v0[n][d]=-vv;
      }
      else v0[n][d]=0;
    }
  }
}

void init(){
  for (int n=0; n<N; n++){
    x[n][0]=x0[n][0];
    x[n][1]=x0[n][1];
    v[n][0]=v0[n][0];
    v[n][1]=v0[n][1];
  }
  iterations=0;
}

void draw(int xdim, int ydim){
  int size=xdim;
  if (ydim<size) size=ydim;
  scalefac=size/L;

  mydrawline(1,0,size,size,size);
  mydrawline(1,size,0,size,size);
  for (int n=0; n<N; n++){
    int xx=x[n][0]*scalefac;
    int yy=x[n][1]*scalefac;
    myfilledcircle(n+1,xx,size-yy,0.5*scalefac);
  }
```

```
}

int main(){
  struct timespec ts={0,0};
  int cont=0;
  int sstep=0;
  int repeat=10;
  int done=0;
  char name[50],mname[N][50];

  setup();
  init();

  AddFreedraw("Particles",&draw);
  StartMenu("Newton",1);
  DefineDouble("L",&L);
  DefineDouble("dt",&dt);
  DefineDouble("k",&k);
  StartMenu("init",0);
  for (int n=0; n<N; n++){
  }
  if (N<15)
    for (int n=0; n<N; n++){
      sprintf(mname[n],"Particle %i",n);
      StartMenu(mname[n],0);
      DefineDouble("m",&m[n]);
      for (int d=0; d<D; d++){
        sprintf(name,"x[%i]",d);
        DefineDouble(name,&x0[n][d]);
      }
      for (int d=0; d<D; d++){
        sprintf(name,"v[%i]",d);
        DefineDouble(name,&v0[n][d]);
      }
      EndMenu();
    }
  DefineDouble("vv",&vv);
  DefineFunction("setup",&setup);
  DefineFunction("init",&init);
  DefineFunction("CM",&CM);
  EndMenu();
  for (int d=0; d<D; d++){
    DefineDouble("C",&C[d]);
  }
  DefineDouble("scale",&scalefac);
  DefineInt("points",&points);
```

```
  DefineGraph(freedraw_,"graph");
  DefineInt("repeat",&repeat);
  DefineBool("sstep",&sstep);
  DefineLong("NS_slow",&ts.tv_nsec);
  DefineBool("cont",&cont);
  DefineBool("done",&done);
  EndMenu();
  while (!done){
    Events(1);
    DrawGraphs();
    if (cont||sstep){
      sstep=0;
      for (int i=0; i<repeat; i++) iterate(x,v,dt);
      nanosleep(&ts,NULL);
    }
    else sleep(1);
  }
}
```