Molecular Dynamics

The molecular dynamics technique simulates a classical system of molecules in the most natural way: it integrates the governing equations of motion through time. The sequence of configurations generated could be run as a movie and looks very realistic (or at least what we envision as realistic based on our everyday experiences with classical mechanics). Properties are collected as a time average. We encountered molecular dynamics at the very outset of this text, where we derived the algorithm for tracing the motion of hard-sphere atoms. For such potentials the algorithm entails repeatedly advancing the system to the next pair-collision and handling the ensuing collision dynamics. Hard potentials are not sufficiently realistic for many applications, and when we move to more accurate, soft potentials we can no longer apply the very efficient collision-detection algorithm. Instead we must apply a more traditional numerical technique of the type encountered in other fields of numerical simulation and analysis. This is one of the primary topics of the present chapter. However, as we will see, there is much more to molecular dynamics than the mere application of an off-the-shelf numerical technique to the equations of motion. Before entering a discussion of the algorithmic matters, it is worthwhile to review the standard formulations of classical mechanics.

## Classical mechanics

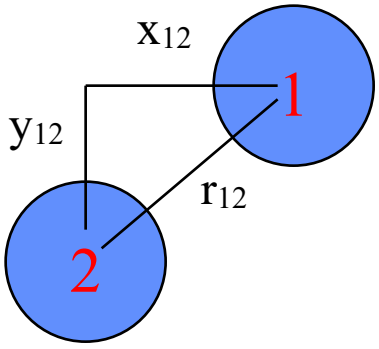
(this discussion is already in the web textbook)

From the Hamiltonian formulation of mechanics, the equations of motion are written as a set of first-order differential equations

$$\begin{aligned} \frac{d\mathbf{r}_j}{dt} &= \frac{\mathbf{p}_j}{m} \\ \frac{d\mathbf{p}_j}{dt} &= \mathbf{F}_j \end{aligned} \tag{1.1}$$

## Calculating forces

The forces needed to integrate Eq. (1.1) are derived from the intra-atomic potential model. The force is the gradient of the potential energy. In the simplest case the potential is pairwise additive and spherically symmetric. For this situation the geometry



of the force calculation is depicted in Illustration 1 (this 2D example is easily converted to the 3D situation). The (vector) force that atom 2 exerts on atom 1 is

$$\begin{aligned}\mathbf{F}_{2\to 1} &= -\nabla_{\mathbf{r}_1} u(r_{12}) \\ &= -\frac{\partial u(r_{12})}{\partial x_1}\mathbf{e}_x - \frac{\partial u(r_{12})}{\partial y_1}\mathbf{e}_y \\ &= -\frac{du(r_{12})}{dr_{12}}\left[\frac{\partial r_{12}}{\partial x_1}\mathbf{e}_x + \frac{\partial r_{12}}{\partial y_1}\mathbf{e}_y\right] \\ &= -\frac{f(r_{12})}{|\mathbf{r}_{12}|}\left[x_{12}\mathbf{e}_x + y_{12}\mathbf{e}_y\right]\end{aligned}$$

where $f(r_{12}) = -\left.\frac{du(r)}{dr}\right|_{r=r_{12}}$ with $r_{12} = |\mathbf{r}_{12}|$. Here $\mathbf{r}_{12}$ is the vector difference $\mathbf{r}_2 - \mathbf{r}_1$; in particular

$$\begin{aligned}\mathbf{r}_{12} &= (x_2 - x_1)\mathbf{e}_x + (y_2 - y_1)\mathbf{e}_y \\ &= x_{12}\mathbf{e}_x + y_{12}\mathbf{e}_y\end{aligned}$$

so we can write (in a form applicable to any dimension)

$$\begin{aligned}\mathbf{F}_{2\to 1} &= -f(r_{12})\frac{\mathbf{r}_{12}}{|\mathbf{r}_{12}|} \\ &= -f(r_{12})\hat{\mathbf{r}}_{12}\end{aligned}$$

Of course this relation satisfies Newton's third law

$$\mathbf{F}_{2\to 1} = -\mathbf{F}_{1\to 2}$$

As an example, let's consider the Lennard-Jones model

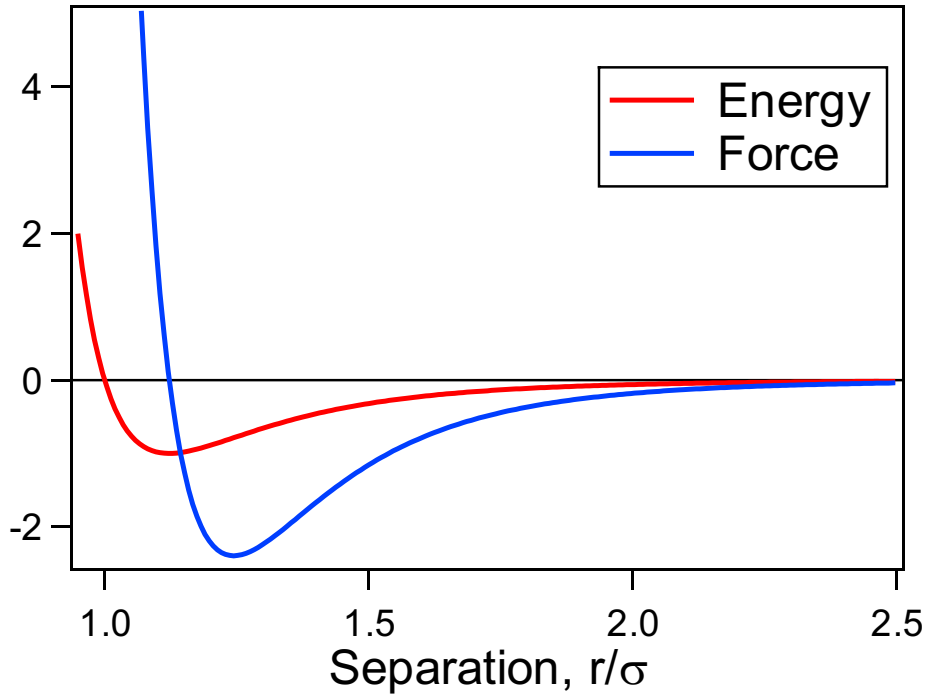
$$u(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

$$\begin{aligned}f(r) &= -\frac{du}{dr} \\ &= +\frac{48\varepsilon}{\sigma}\left[\left(\frac{\sigma}{r}\right)^{13} - \frac{1}{2}\left(\frac{\sigma}{r}\right)^{7}\right]\end{aligned}$$

$$\mathbf{F}_{2\to 1} = -\frac{48\varepsilon}{\sigma^2}\left[\left(\frac{\sigma}{r_{12}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{12}}\right)^{8}\right]\left[x_{12}\mathbf{e}_x + y_{12}\mathbf{e}_y\right]$$

It is good that the force, like the potential, is given in terms of even powers of $r_{12}$; this means that the force can be computed while avoiding an expensive square-root calculation.

The LJ potential and the corresponding force (*i.e.*, its magnitude) are presented in Illustration 2.




## Integration algorithms

The usual approach to integrating a set of first-order differential equations is to advance the system variables through a discrete step in time δt by approximating the action of the derivative via finite differences. Methods vary in several ways. Some make use of the prior evolution of the trajectory to approximate the effects of higher-order derivatives. There are choices to be made about when and how to apply information obtained upon evaluating the right-hand side of the governing differential equations. According to Eq. (1.1), in MD these function evaluations require computation of the forces acting on each atom. In an MD simulation the force calculation consumes an overwhelming amount of the total CPU time (as much as 90%), and it is essential that it be performed no more than once per time step. Thus, most standard methods are entirely inappropriate for MD simulation because they do not economize on the force calculation (*e.g.*, Runge-Kutta would require it be evaluated four times per time step). One could justify multiple force evaluations per time step if it led to a commensurate increase in the step size (*i.e.,* two

force evaluations would permit more than doubling the time step). In most MD calculations this proportion cannot be achieved because the forces are very rapidly changing nonlinear functions, particularly in the region where the atoms repel each other.

So one of the desirable features of an MD integrator is to minimize the need for the force calculation. Also the integrator should be stable. This means that any small departure of the integration from the correct trajectory will not tend to even greater departures. One might wish that the integrator also be accurate, in the sense that over long times it produces a trajectory that deviates from the correct trajectory by an amount as small as possible. However, such an ambition is misplaced. The detailed dynamics of multiatomic systems is complex and chaotic. An infinitesimal change in the position of one atom's coordinate or momentum will propagate quickly to all other atoms, and ultimately (and perhaps suddenly) lead to a large, strongly disproportionate, deviation in the trajectory. Since all atom positions and momenta are kept to a finite precision in the computer, such deviations are inevitable no matter what algorithm is applied. While this situation might seem troubling, we should remember that at this point we have already introduced a much more severe approximation by using classical mechanics in lieu of the correct quantum treatment. Much more important than absolute accuracy of the trajectories is adherence of the whole system to conservation of energy and momentum. Failure in this regard implies failure to sample the correct statistical mechanical ensemble. Small fluctuations in energy conservation can be tolerated, but there should be no systematic drift. Another desirable (but not essential) feature of an integrator is that it be time-reversible. This means that if at some instant all the velocities were reversed, the system would in principle backtrack over its prior trajectory. As we proceed we will show examples of how different algorithms do or do not meet this criterion. Finally, it is good for an integrator to be symplectic. This means that it preserves the volume of phase space it is attempting to sample. We will delve into this issue more at a later point.

## Verlet algorithms

Work by Verlet has led to a class of algorithms that are simple and effective, and consequently very popular. The original Verlet algorithm is based on a simple expansion of the atomic coordinates forward and backward in time by a step δt

$$\mathbf{r}(t+\delta t) = \mathbf{r}(t) + \tfrac{1}{m}\mathbf{p}(t)\delta t + \tfrac{1}{2m}\mathbf{F}(t)\delta t^2 + \tfrac{1}{3!}\dddot{\mathbf{r}}(t)\delta t^3 + O(\delta t^4)$$
$$\mathbf{r}(t-\delta t) = \mathbf{r}(t) - \tfrac{1}{m}\mathbf{p}(t)\delta t + \tfrac{1}{2m}\mathbf{F}(t)\delta t^2 - \tfrac{1}{3!}\dddot{\mathbf{r}}(t)\delta t^3 + O(\delta t^4)$$

Addition of these formulas yields

$$\mathbf{r}(t+\delta t) + \mathbf{r}(t-\delta t) = 2\mathbf{r}(t) + \tfrac{1}{m}\mathbf{F}(t)\delta t^2 + O(\delta t^4)$$

which upon rearrangement gives a prescription for the position at the next step in time

$$\mathbf{r}(t+\delta t) = 2\mathbf{r}(t) - \mathbf{r}(t-\delta t) + \tfrac{1}{m}\mathbf{F}(t)\delta t^2 + O(\delta t^4)$$

Note that the position at the previous step is saved and used to project the position at the next step. Remarkably, the positions are updated without ever consulting the velocities. In fact, a small drawback to the method is that the momenta are never computed. However, if it is desired to know them (for example, to compute the momentum temperature), they can be estimated by a finite difference

$$\mathbf{p}(t) = \frac{m}{2\delta t}\left[\mathbf{r}(t+\delta t) - \mathbf{r}(t-\delta t)\right] + O(\delta t^2)$$

It is helpful to introduce in lieu of the previous-step position **r**(t-δt) a quantity Δ**r** that holds the position change, and to break the advancement into two steps

$$\begin{aligned} \Delta\mathbf{r}^{new} &= \Delta\mathbf{r}^{old} + \tfrac{1}{m}\mathbf{F}(t)\delta t^2 \\ \mathbf{r}^{new} &= \mathbf{r}^{old} + \Delta\mathbf{r}^{new} \end{aligned} \qquad (1.2)$$

Since Δ**r** is a quantity of order δt, one advantage of this reformulation is that we never add terms that differ by more than one order in δt; this can help avoid errors associated with the machine precision. Another advantage is that the effects of the periodic boundaries are handled more naturally. In the original formulation, if periodic boundaries are invoked after the atom is moved, one must be careful to work with the minimum image when evaluating the difference $\mathbf{r}(t) - \mathbf{r}(t-\delta t)$. In the reformulation one can apply periodic boundaries to $\mathbf{r}^{new}$ without affecting Δ**r**, so this programming error is not likely to be made.

If we introduce what is basically a change of notation, writing Δ**r** in terms of the momentum as (**p**/m)δt, then Eq. (1.2) can be written

$$\begin{aligned} \mathbf{p}(t+\tfrac{1}{2}\delta t) &= \mathbf{p}(t-\tfrac{1}{2}\delta t) + \mathbf{F}(t)\delta t \\ \mathbf{r}(t+\delta t) &= \mathbf{r}(t) + \tfrac{1}{m}\mathbf{p}(t+\tfrac{1}{2}\delta t)\delta t \end{aligned} \qquad (1.3)$$

These equations constitute the so-called Verlet leapfrog algorithm. Although the momentum now appears explicitly, its evaluation at time *t* (*i.e.*, for the same time that the positions are known) requires interpolating the values at the surrounding half-intervals

$$\mathbf{p}(t) = \frac{1}{2}\left[\mathbf{p}(t+\tfrac{1}{2}\delta t) + \mathbf{p}(t-\tfrac{1}{2}\delta t)\right]$$

This suggests that an algorithm in which the velocities are integrated using a time step that is half as large as the time step for advancing the positions. Such an approach is systematized in the velocity-Verlet algorithm. The working formulas are

$$\begin{aligned}
\mathbf{p}(t+\tfrac{1}{2}\delta t) &= \mathbf{p}(t) + \tfrac{1}{2}\mathbf{F}(t)\delta t \\
\mathbf{r}(t+\delta t) &= \mathbf{r}(t) + \tfrac{1}{m}\mathbf{p}(t+\tfrac{1}{2}\delta t)\delta t \\
\mathbf{p}(t+\delta t) &= \mathbf{p}(t+\tfrac{1}{2}\delta t) + \tfrac{1}{2}\mathbf{F}(t+\delta t)\delta t
\end{aligned} \tag{1.4}$$

Between the second and third steps the forces are computed for the positions obtained in the first step. Note that the force added to the momentum in the third step is the same force used to increment the momentum in the first step at the next time increment. Performing this addition all at once gives us instead just the first equation of the leapfrog algorithm, so it is easy to see that they yield identical trajectories. The advantage of the velocity Verlet treatment is the availability of the momenta at the same time as the positions.

### Time reversibility

The Verlet algorithms are time reversible. This means that upon changing the sign of the time increment δt, the algorithm will (in principle) retrace the steps it just followed. Note that we are not saying that time reversibility involves a reversal of the algorithm. For example, testing for time reversibility does not entail running the steps in Eq. (1.4) from the third one to the first one. Rather, we simply change δt to -δt, and then run the algorithm forward as written. An example of a time-irreversible algorithm is a simple forward Euler approach (which is, by the way, a terrible algorithm)

$$\begin{aligned}
\mathbf{r}(t+\delta t) &= \mathbf{r}(t) + \tfrac{1}{m}\mathbf{p}(t)\delta t + \tfrac{1}{2m}\mathbf{F}(t)\delta t^2 \\
\mathbf{p}(t+\delta t) &= \mathbf{p}(t) + \mathbf{F}(t)\delta t
\end{aligned} \tag{1.5}$$

Assuming we have advanced from at time $t_0$ to $t_0$ + δt, we can look to see where we end up if we subsequently apply the algorithm with δt replaced by -δt. We'll use a subscript 'f' to indicate a coordinate/momentum obtained during the forward traversal, and an 'r' to indicate those obtained upon reversing the time

$$\begin{aligned}
\mathbf{r}_r(t_o+\delta t-\delta t) &= \mathbf{r}_f(t_o+\delta t) + \tfrac{1}{m}\mathbf{p}_f(t_o+\delta t)(-\delta t) + \tfrac{1}{2m}\mathbf{F}(t_o+\delta t)(-\delta t)^2 \\
\mathbf{p}_r(t_o+\delta t-\delta t) &= \mathbf{p}_f(t_o+\delta t) + \mathbf{F}(t_o+\delta t)(-\delta t)
\end{aligned} \tag{1.6}$$

Substituting in from Eq. (1.5) for the corresponding terms on the right-hand side of Eq. (1.6)

$$\begin{aligned}
\mathbf{r}_r(t_o) &= \left[\mathbf{r}_f(t_o) + \tfrac{1}{2m}\mathbf{F}(t_o)\delta t^2\right] + \tfrac{1}{m}\left[\mathbf{F}(t_o)\delta t\right](-\delta t) + \tfrac{1}{2m}\mathbf{F}(t_o+\delta t)(-\delta t)^2 \\
\mathbf{p}_r(t_o) &= \left[\mathbf{p}_f(t_o) + \mathbf{F}(t_o)\delta t\right] + \mathbf{F}(t_o+\delta t)(-\delta t)
\end{aligned}$$

And now simplifying both sides

$$\mathbf{r}_r(t_o) = \mathbf{r}_f(t_o) + \tfrac{1}{2m}\left[\mathbf{F}(t_o + \delta t) - \mathbf{F}(t_o)\right]\delta t^2$$
$$\mathbf{p}_r(t_o) = \mathbf{p}_f(t_o) - \left[\mathbf{F}(t_o + \delta t) - \mathbf{F}(t_o)\right]\delta t$$

The lack of equality in these equations indicates that the formulas are not time reversible. This situation comes about because the forces encountered at the beginning and end of each time step do not enter in a symmetric way. In contrast, consider the same analysis of the velocity-Verlet algorithm. After proceeding one time step according to Eq. (1.4) we reverse time and obtain, first for the momentum at the half-time increment

$$\mathbf{p}_r(t_o + \delta t - \tfrac{1}{2}\delta t) = \mathbf{p}_f(t_o + \delta t) + \tfrac{1}{2}\mathbf{F}(t_o + \delta t)(-\delta t)$$

which, upon substitution from Eq. (1.4) and simplifying

$$\mathbf{p}_r(t_o + \delta t - \tfrac{1}{2}\delta t) = \left[\mathbf{p}_f(t_o + \tfrac{1}{2}\delta t) + \tfrac{1}{2}\mathbf{F}(t_o + \delta t)\delta t\right] + \tfrac{1}{2}\mathbf{F}(t_o + \delta t)(-\delta t)$$
$$\mathbf{p}_r(t_o + \tfrac{1}{2}\delta t) = \mathbf{p}_f(t_o + \tfrac{1}{2}\delta t)$$

which shows that the same momentum is obtained at the half-time step. The algorithm next updates the positions

$$\mathbf{r}_r(t_o + \delta t - \delta t) = \mathbf{r}_f(t_o + \delta t) + \tfrac{1}{m}\mathbf{p}_r(t_o + \delta t - \tfrac{1}{2}\delta t)(-\delta t)$$

Substituting and simplifying as before

$$\mathbf{r}_r(t_o + \delta t - \delta t) = \left[\mathbf{r}_f(t_o) + \tfrac{1}{m}\mathbf{p}_f(t_o + \tfrac{1}{2}\delta t)\delta t\right] + \tfrac{1}{m}\mathbf{p}_r(t_o + \delta t - \tfrac{1}{2}\delta t)(-\delta t)$$
$$\mathbf{r}_r(t_o) = \mathbf{r}_f(t_o) + \tfrac{1}{m}\left(\mathbf{p}_f(t_o + \tfrac{1}{2}\delta t) - \mathbf{p}_r(t_o + \tfrac{1}{2}\delta t)\right)\delta t$$
$$\mathbf{r}_r(t_o) = \mathbf{r}_f(t_o)$$

Finally, the momentum is again updated

$$\mathbf{p}_r(t_o + \delta t - \delta t) = \mathbf{p}_r(t_o + \delta t - \tfrac{1}{2}\delta t) + \tfrac{1}{2}\mathbf{F}(t_o + \delta t - \delta t)(-\delta t)$$

which becomes

$$\mathbf{p}_r(t_o) = \mathbf{p}_r(t_o + \tfrac{1}{2}\delta t) - \tfrac{1}{2}\mathbf{F}(t_o)\delta t$$
$$\mathbf{p}_r(t_o) = \mathbf{p}_f(t_o + \tfrac{1}{2}\delta t) - \tfrac{1}{2}\mathbf{F}(t_o)\delta t$$
$$\mathbf{p}_r(t_o) = \left[\mathbf{p}_f(t_o) + \tfrac{1}{2}\mathbf{F}(t_o)\delta t\right] - \tfrac{1}{2}\mathbf{F}(t_o)\delta t$$
$$\mathbf{p}_r(t_o) = \mathbf{p}_f(t_o)$$

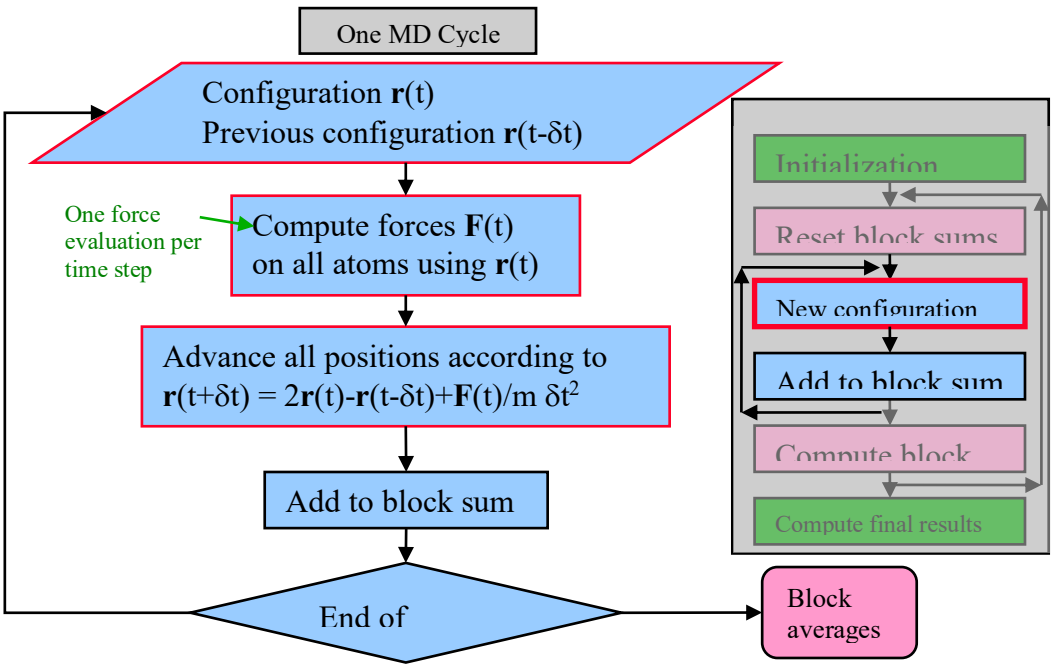
and complete time-reversibility is demonstrated.

## Implementation

Illustration 1 summarizes the Verlet algorithm, and shows how it fits into the structure of a molecular simulation program as presented in a previous chapter. One issue not covered in the algorithm is start up of the simulation. At time zero the momentum at the preceding half-time interval is not available. However it is easily estimated from the forces acting on the atoms in their initial configuration

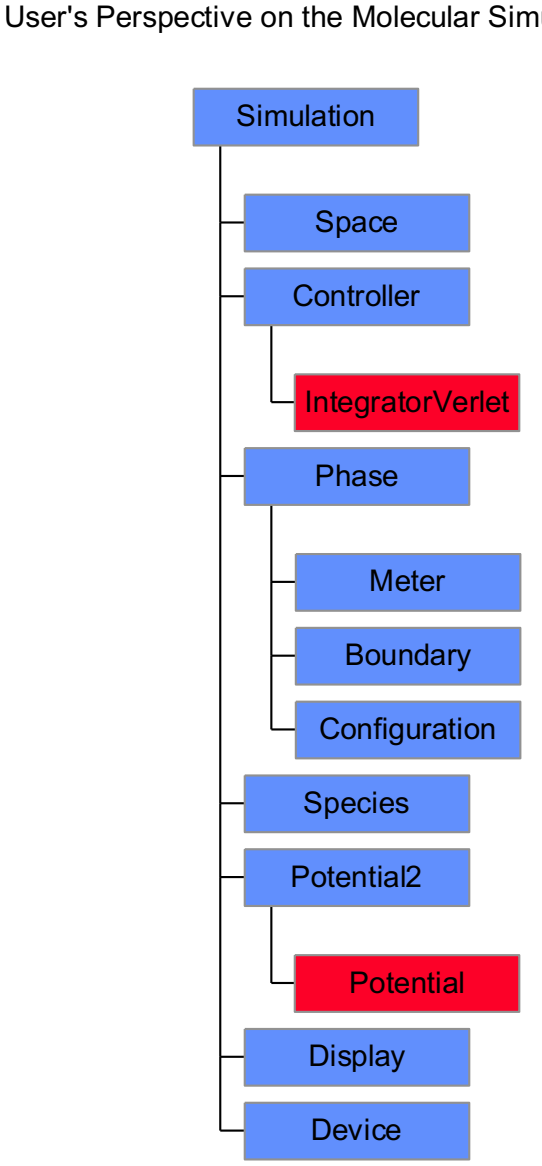
$$\mathbf{p}(t_0 - \tfrac{1}{2}\delta t) = \mathbf{p}(t_0) - \mathbf{F}(t_0)\tfrac{1}{2}\delta t$$

Selection of initial velocities is performed at random, consistent with the desired temperature as described previously in the context of hard-sphere MD.




## Java code

The parts of a molecular simulation particular to molecular dynamics enter into the molecular simulation API through the Integrator and the Potential. The integrator is responsible for implementing the molecular dynamics algorithm, while the potential is responsible for defining the force between atoms. These parts are highlighted in the chart




given in Illustration 4. Java code implementing the force calculation for the LJ model is presented in Illustration 5.

```
//method inside public class PotentialLJ implements PotentialSoft
//Space.Vector used to compute and return a force
private Space.Vector force = Simulation.space.makeVector();

//Given a pair of atoms, computes the force that atom2 exerts on atom1
public Space.Vector force(AtomPair pair) {
  double r2 = pair.r2();          //squared distance between pair of atoms
  if(r2 > cutoffDiameterSquared) {
    force.E(0.0);}                //outside cutoff; no interaction
  }
  else {                          //inside cutoff
    double s2 = sigmaSquared/r2;  // (sigma/r)^2
    double s6 = s2*s2*s2;         // (sigma/r)^6
    force.E(pair.dr());           // f = (x12 ex + y12 ey)  (vector)
    force.TE(-48*s2*s6*(s6-0.5)/sigmaSquared);
                                  // f *= -48*(sigma/r)^8 * [(sigma/r)^6 - 1/2] / sigma^2
  }
  return force;
```

An implementation of the velocity Verlet algorithm is given in Illustration 6. Note that this class defines the Integrator.Agent inner class to have a vector holding the force acting on each atom.

```
//Method inside of public class IntegratorVelocityVerlet extends IntegratorMD

//advances all atom positions by one time step
public void doStep() {

   atomIterator.reset();              //reset iterator of atoms
   while(atomIterator.hasNext()) {    //loop over all atoms
     Atom a = atomIterator.next();  //  advancing positions full step
     Agent agent = (Agent)a.ia;     //  and momenta half step
     Space.Vector r = a.position();
     Space.Vector p = a.momentum();
     p.PEa1Tv1(0.5*timeStep,agent.force);  // p += f(old)*dt/2
     r.PEa1Tv1(timeStep*a.rm(),p);         // r += p*dt/m
     agent.force.E(0.0);
  }
  pairIterator.allPairs(forceSum);    //compute forces on each atom
  atomIterator.reset();
  while(atomIterator.hasNext()) {     //loop over atoms again
     Atom a = atomIterator.next();   //  finishing the momentum step
     a.momentum().PEa1Tv1(0.5*timeStep,((Agent)a.ia).force);  //p += f(new)*dt/2
  }
  return;
}

//class defining the agent for this integrator
public final static class Agent implements Integrator.Agent {  //need public so to use with
instanceof
        public Atom atom;
        public Space.Vector force;

        public Agent(Atom a) {
            atom = a;
            force = Simulation.space.makeVector();
        }
    }
```