Stadium billiard with soft repulsive walls

Soft repulsive (Weeks-Chandler-Andersen) potential

\[ u(R) = \begin{cases} 
4 \varepsilon \left[ \left( \frac{\sigma}{R} \right)^{12} - \left( \frac{\sigma}{R} \right)^{6} \right] + \varepsilon & \text{for } R \leq r_c \\
0 & \text{for } R > r_c 
\end{cases} \]

Pair potentials, \( \varepsilon = 1 \)

Force on the ball near the wall of a stadium with "soft" walls

Magnitude of the force:

\[ F(R) = -\frac{dU}{dR} = \frac{48}{\sigma^2} \left[ \left( \frac{\sigma}{R} \right)^{14} - \frac{1}{2} \left( \frac{\sigma}{R} \right)^{8} \right] R \]

Direction of the force: along (inward pointing) normal vectors

\[ F_x = F(R)n_x \]
\[ F_y = F(R)n_y \]
Calculate the force components:

In general:

\[ F_x = -\frac{\partial U}{\partial x} \quad \text{Here: } U = U(R) \]

\[ F_y = -\frac{\partial U}{\partial y} \]

\[ \Rightarrow n_x = \frac{\partial R}{\partial x} \]

\[ n_y = \frac{\partial R}{\partial y} \]

We need an expression for the distance \( R \) (the distance of the ball from the wall) in terms of the \( x \) and \( y \) coordinates of the ball.

The components \( n_x \) and \( n_y \) can then be calculated by taking a derivative.

Region 2 \(-\alpha r < x < \alpha r\)

\[ R = r - y \quad \text{for } y > 0 \]

\[ R = r + y \quad \text{for } y < 0 \]

Region 1 \(-(\alpha + 1)r < x < -\alpha r\)

\[ (r - R)^2 = (x + \alpha r)^2 + y^2 \]

\[ \Rightarrow R = r - \sqrt{(x + \alpha r)^2 + y^2} \]

Region 3 \(\alpha r < x < (\alpha + 1)r\)

\[ (r - R)^2 = (x - \alpha r)^2 + y^2 \]

\[ \Rightarrow R = r - \sqrt{(x - \alpha r)^2 + y^2} \]

Taking the derivatives, we find for \( y > 0 \)

\[ n_x = \frac{\partial R}{\partial x} = \begin{cases} 
\frac{x + \alpha r}{\sqrt{(x + \alpha r)^2 + y^2}} & \text{Region 1} \\
0 & \text{Region 2} \\
\frac{x - \alpha r}{\sqrt{(x - \alpha r)^2 + y^2}} & \text{Region 3}
\end{cases} \]

For \( y < 0 \), \( n_x \) stays the same but \( n_y \) changes sign.
• Molecular dynamics (MD) simulations are a method for solving Newton’s equation of motion for classical particles interacting with each other, the walls of the container, external fields, etc.

• In MD simulations, just like in other numerical methods for solving differential equations, time is a discrete variable that may be labeled by an integer, say \( k \).

• One solves Newton’s equation of motion with a numerical (finite difference) algorithm to go from time \( t_k \) to time \( t_{k+1} = t_k + \delta t \), where \( \delta t \) is the step size.

• One such method is the Verlet algorithm, others include the “velocity Verlet” algorithm, the “leap-frog” method, and “predictor-corrector” methods. The methods differ in ease of implementation, speed and accuracy, where all these properties depend on the system that is simulated (see, for example, J. M. Haile, *Molecular Dynamics Simulation*, Wiley, New York, 1992, chapter 4).

• It is always a good idea to monitor quantities that are expected to be conserved in order to recognize when the algorithm/parameters lead to systematic errors.

All finite difference methods are based on Taylor expansions of the coordinates.

For simplicity, we focus on the \( x \)-components (\( y \) and \( z \) components are analogous), drop the particle subscripts and write \( x(t) \) for the position at time \( t \), and \( v(t) \) and \( a(t) \) for the corresponding velocity, and acceleration at time \( t \), respectively.

\[
\begin{align*}
\frac{dx}{dt} &= v(t) \\
\frac{dv}{dt} &= a(t) \\
\frac{da}{dt} &= \theta(t)
\end{align*}
\]

Add (1) and (2), note that the odd terms drop out, and solve for \( x(t + \delta t) \).

\[
x(t + \delta t) = 2x(t) - x(t - \delta t) + \frac{\delta t}{3!} \frac{d^3 x}{dt^3} + O(\delta t)^4
\]  

This is the basis for the Verlet algorithm. Note that the velocity does not appear in (3) and that the truncation error is of order \((\delta t)^4\) even though only 2\(^{nd}\) order derivatives need to be calculated. To arrive at the final equations, drop the error term and write

\[
x = x(t_k), \quad x_{\text{old}} = x(t_k - \delta t), \quad x_{\text{new}} = x(t_k + \delta t), \quad v_x = v_x(t_k), \quad a_x = a_x(t_k).
\]
Verlet algorithm

1. call a function to calculate the acceleration $a_x = f_x/m$, where $f_x$ is the $x$-component of the net force on the particle at time $t_k$
2. calculate the new position
   \[ x_{\text{new}} = 2x - x_{\text{old}} + a_x \times (\delta t)^2 \]
3. calculate the (approximate) velocity
   \[ v_x = \frac{x_{\text{new}} - x_{\text{old}}}{2\delta t} \]
4. reassign positions
   \[ x_{\text{old}} = x \]
   \[ x = x_{\text{new}} \]
Repeat from 1.

The Verlet algorithm is not “self starting”: it requires information about two time steps and the initial state gives information only about a single time step.

We use an Euler-Cromer algorithm to advance the system one time step from the initial state $x_0, v_{x0}$
1. call a function to calculate $a_x$ for the initial state
2. calculate the new velocity
   \[ v_x = v_{x0} + a_x \delta t \]
3. calculate the new position
   \[ x = x_0 + v_x \delta t \]
4. reassign position variable
   \[ x_{\text{old}} = x_0 \]

Note: in systems with “periodic boundary conditions” (typical for materials simulations) the positions have to be corrected after steps 3.

Initial conditions:

\[
\begin{align*}
X0 &= -0.2; 
Y0 &= 0; \quad \% \text{initial position} \\
v0 &= 1.0; \quad \% \text{initial speed} \\
\phi0 &= \pi/4; \quad \% \text{initial velocity angle} \\
vx0 &= v0*cos(\phi0); 
vy0 &= v0*sin(\phi0); \quad \% \text{initial velocity components}
\end{align*}
\]

Euler step:

\[
\begin{align*}
% \text{Starting from the initial conditions, forward the solution one step using} \\
% \text{the Euler method} \\
[ax,ay] &= \text{faccstadium}(x(1),y(1)); \quad \% \text{call faccstadium to calculate the acceleration} \\
vx(2) &= vx(1)+ax*delt; \quad \% \text{calculate the new (time-step 2) x-component of the velocity} \\
x(2) &= x(1)+vx0*delt; \quad \% \text{calculate the new (time-step 2) x-component of the position} \\
vy(2) &= vy(1)+ay*delt; \quad \% \text{calculate the new (time-step 2) y-component of the velocity} \\
y(2) &= y(1)+vy0*delt; \quad \% \text{calculate the new (time-step 2) y-component of the position}
\end{align*}
\]

Verlet algorithm:

\[
\begin{align*}
% \text{Simulation of the motion of the ball with the Verlet algorithm} \\
\text{for } k = 2: \text{Nt-1} \quad \% \text{loop over time steps} \\
% \text{Calculate the components of position and velocity at the next time step} \\
[ax,ay] &= \text{faccstadium}(x(k),y(k)); \quad \% \text{call faccstadium to calculate the acceleration} \\
x(k+1) &= 2*x(k)-x(k-1)+ax*delt*delt; \quad \% \text{advance the x-component of the position} \\
vx(k) &= (x(k+1)-x(k-1))/(2*delt); \quad \% \text{calculate the x-component of the velocity} \\
y(k+1) &= 2*y(k)-y(k-1)+ay*delt*delt; \quad \% \text{advance the y-component of the position} \\
vy(k) &= (y(k+1)-y(k-1))/(2*delt); \quad \% \text{calculate the y-component of the velocity} \\
\end{align*}
\]
(program contains evaluation of Poincare section here)

\[
\begin{align*}
vx(\text{Nt}) &= vx(\text{Nt}-1); \quad \% \text{assign the last values of the velocity components} \\
vx(\text{Nt}) &= vy(\text{Nt}-1);
\end{align*}
\]