# Build a Toy Model for Exploring Molecular / Atomic Interactions

## What you are doing

You are given the chance to build a toy that shows how atoms might jostle, bounce, and push each other in a small box. This is **not full-scale Molecular Dynamics**, but a playful starting point. You will:

- create arrays for positions and velocities,
- update them using Newton's laws,
- add forces between pairs,
- reflect from walls,
- plot energy and final positions.

**Goal.** At the end, you will have a small simulator you can tweak — faster steps, stronger springs, more particles — and see what happens.

## 1. Setting up the world

Install the tools we need (once). If already installed, ignore this step.:

```
pip install numpy matplotlib
```

Now start a new Python file or Jupyter notebook and set up particle positions and velocities:

```python
import numpy as np
np.random.seed(0)

N = 12            # number of particles
L = 10.0          # box side length
dt = 0.01         # time step
T  = 1000         # number of steps to simulate

# TODO: create pos, vel, and mass arrays
# Hint: pos should be N x 2 random numbers (x, y) scaled to fit inside box
# vel (vx, vy) should be random velocities
# mass should be all ones
```

**TODO**

Fill in the code for pos, vel, and mass. Print them once to check shapes: `print(pos.shape, vel.shape)`.

## 2. Wall reflections (bounce off walls)

Right now, a particle could fly straight out of the box. You need to make it bounce back.

At the end of every step:

1. Look at every particle.
2. If its $x$-position is $< 0$, put it back at $x = 0$ and flip its $x$-velocity (multiply by $-1$).
3. If its $x$-position is $> L$, put it back at $x = L$ and flip its $x$-velocity.
4. Do the same for $y$-position and $y$-velocity.

The order matters: **first** detect it crossed the wall, **then** flip its velocity, and **then** clamp its position inside. This prevents particles from getting stuck outside the box or gaining energy by accident.

Here is a clear function you can fill in:

```python
def walls_reflect(pos, vel, L):
    # Left wall (x < 0)
    left_hits = pos[:, 0] < 0
    vel[left_hits, 0] *= -1        # flip x velocity
    pos[left_hits, 0] = 0.0        # push back inside

    # Right wall (x > L)
    right_hits = pos[:, 0] > L
    vel[right_hits, 0] *= -1
    pos[right_hits, 0] = L

    # Bottom wall (y < 0)
    bottom_hits = pos[:, 1] < 0
    vel[bottom_hits, 1] *= -1
    pos[bottom_hits, 1] = 0.0

    # Top wall (y > L)
    top_hits = pos[:, 1] > L
    vel[top_hits, 1] *= -1
    pos[top_hits, 1] = L

    return pos, vel
```

**TODO**

Test this function: Create a particle just outside the right wall (`pos = [[L+0.1, 5]]`) with a positive x-velocity. Call `walls_reflect`. Check that it ends up at exactly x=L and its velocity becomes negative.

**Hint.** If you see energy growing after a bounce, check the order: update position first, then reflect. Never flip velocity before you know where the particle is.

## 3. Pair forces: springs between particles

You are given this formula:

$$\mathbf{F}_{ij} = -k\,(r - r_0)\,\frac{\mathbf{r}}{\|\mathbf{r}\|}$$

where $k$ is stiffness, $r_0$ is preferred spacing, $r$ is current distance, and $\mathbf{r}$ is displacement vector.

```python
k  = 15.0
r0 = 0.8
rc = 2.0    # cutoff distance

def pair_forces(pos):
    F = np.zeros_like(pos)
    # TODO: double loop over i,j (j>i)
    # compute distance, apply spring force if r < rc
    # add equal and opposite forces to F[i], F[j]
    return F
```

**TODO**

Implement the double loop version first. Print F for a few particles to see if it makes sense (zero if far apart).

## 4. Updating positions and velocities

Now you will make the particles move step by step. At every small time step $\Delta t$, do three things in this order:

1. Find the total force $\mathbf{F}$ on each particle (use your `pair_forces` function).
2. Change velocity using: $\mathbf{v} = \mathbf{v} + (\mathbf{F}/m)\,\Delta t$
3. Change position using: $\mathbf{x} = \mathbf{x} + \mathbf{v}\,\Delta t$
4. Make sure no particle is outside the box. If it is, flip its velocity (your wall bounce function).

In code, that looks like:

```
def step_once(pos, vel, mass, dt, L):
    # 1. Compute forces
    F = pair_forces(pos)

    # 2. Update velocities using F = m a
    acc = F / mass[:, None]
    vel = vel + acc * dt

    # 3. Update positions using new velocities
    pos = pos + vel * dt

    # 4. Bounce from walls
    walls_reflect(pos, vel, L)

    return pos, vel
```

**TODO**

Fill in any missing parts in this function. Then run it in a loop for 50–100 steps and print the first particle's position each time. Do you see it move smoothly inside the box?

## 5. Energy check

Define a function to measure kinetic energy:

$$E_k = \frac{1}{2}\sum mv^2$$

```
def kinetic_energy(vel, mass):
    # TODO: compute and return total KE (use np.sum)
```

Collect KE in a list during the simulation and plot later.

## 6. A smoother update recipe

The simple recipe you just coded works, but if you look at your kinetic energy plot you might notice that it slowly drifts up or down. Here is a slightly improved way to update positions and velocities that usually keeps energy more steady.

At every small time step $\Delta t$, do this:

1. Compute the force $\mathbf{F}$ on each particle.
2. Use half of this force to change the velocity halfway:

$$\mathbf{v}_{\text{half}} = \mathbf{v} + \frac{1}{2}\frac{\mathbf{F}}{m}\Delta t$$

3. Move the particles using this half-updated velocity:

$$\mathbf{x}_{\text{new}} = \mathbf{x} + \mathbf{v}_{\text{half}}\Delta t$$

4. Compute the new forces at these new positions.
5. Use half of these new forces to finish the velocity update:

$$\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{half}} + \frac{1}{2}\frac{\mathbf{F}_{\text{new}}}{m}\Delta t$$

6. Bounce from walls if needed.

In code, this might look like:

```python
def step_smooth(pos, vel, mass, dt, L, F_prev=None):
    # 1. Force at old positions
    if F_prev is None:
        F_prev = pair_forces(pos)
    a_prev = F_prev / mass[:, None]

    # 2. Half velocity update
    vel_half = vel + 0.5 * a_prev * dt

    # 3. Move positions
    pos_new = pos + vel_half * dt
    walls_reflect(pos_new, vel_half, L)

    # 4. New forces
    F_new = pair_forces(pos_new)
    a_new = F_new / mass[:, None]

    # 5. Finish velocity update
    vel_new = vel_half + 0.5 * a_new * dt

    return pos_new, vel_new, F_new
```

**TODO**

Replace your previous update function with this new one and re-run the simulation. Plot kinetic energy again. Is it steadier than before? Try with a bigger `dt` — how does the energy behave?

**TODO**

Explain in one line what you observe in the KE plot. Is energy roughly constant?

## Bonus Pointer: Vectorized Forces

Once your double loop works, try NumPy broadcasting for speed:

```
d = pos[None,:,:] - pos[:,None,:]        # pairwise displacements (N,N,2)
r = np.linalg.norm(d, axis=2)            # pairwise distances (N,N)
mask = (r < rc) & (r > 1e-12)            # ignore self and far pairs
u = d / (r[:,:,None] + 1e-12)            # unit vectors
fmag = -k * (r - r0) * mask
F = np.sum(fmag[:,:,None] * u, axis=1)   # sum forces per particle
```

This replaces the loop with array math. It looks fancy, but it's just the same steps done in parallel.

## Checklist

Particles stay inside the box.

Forces are equal and opposite (check by summing F).

Kinetic Energy (KE) stays roughly constant with the smoother update given in step 6

Smaller `dt` makes result more stable.