Computational Physics (PHYS6350)

Lecture 14: Classical molecular dynamics

$$m\ddot{\mathbf{r}}_\mathbf{i} = -\sum_j \nabla_i V_{\mathrm{LJ}}^{ij}(|\mathbf{r_i} - \mathbf{r_j}|)$$

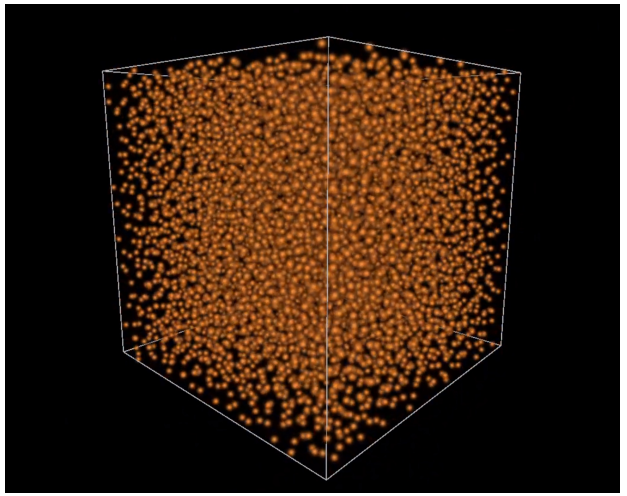
March 3, 2025

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics/tree/spring2025

Molecular dynamics (MD)

- System of N particles with a pair potential



- Newton's equations of motion (classical *N*-body problem)

$$m\ddot{\mathbf{r}}_\mathbf{i} = -\sum_j \nabla_i V_{\mathrm{LJ}}^{ij}(|\mathbf{r}_\mathbf{i} - \mathbf{r}_\mathbf{j}|)$$

- Box simulation
  - Periodic boundary conditions
  - Minimum-image convention

- If *N* is large enough, system can be characterized by macroscopic parameters
  - Energy-Volume-Number (UVN), microcanonical ensemble
  - Temperature-Volume-Number (TVN), canonical ensemble

- MD simulations give access to the **equation of state**

Molecular dynamics equations

Have to solve Newton's equations of motion

$$m_i \ddot{\mathbf{r}}_i = -\sum_{j \neq i} \nabla_i V(\mathbf{r}_i, \mathbf{r}_j)$$

- Desired properties
  - Stability (long simulations)
  - Energy conservation
  - Time-reversibility

- Rewrite as a system of first-order ODEs

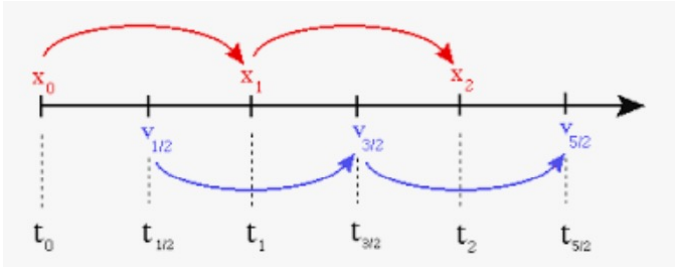
$$\begin{aligned} \dot{\mathbf{r}}_i &= \mathbf{v}_i, \\ \dot{\mathbf{v}}_i &= -(m_i)^{-1} \sum_{j \neq i} \nabla_i V(\mathbf{r}_i, \mathbf{r}_j), \end{aligned}$$

and use the **leapfrog method**

Velocity Verlet method

We have system of equations

$$\frac{dx}{dt} = v,$$
$$\frac{dv}{dt} = f(x,t).$$



The **leapfrog scheme** applied to this system of equations:

$$x(t+h) = x(t) + hv(t+h/2),$$
$$v(t+3h/2) = v(t+h/2) + hf[x(t+h), t+h)],$$

- Coordinates are evaluated at full steps
- Velocities are evaluated at half-steps

Leapfrog method applied to molecular dynamics proble is called ***Velocity Verlet method***.

Velocity Verlet:

$$v(t+h/2) = v(t) + \frac{h}{2} f[x(t), t],$$
$$x(t+h) = x(t) + hv(t+h/2),$$
$$v(t+h) = v(t+h/2) + \frac{h}{2} f[x(t+h), t+h)].$$

Velocity Verlet method

$$\dot{\mathbf{r}}_i = \mathbf{v}_i,$$
$$\dot{\mathbf{v}}_i = -(m_i)^{-1} \sum_{j \neq i} \nabla_i V(\mathbf{r}_i, \mathbf{r}_j),$$

```
# Apply the velocity verlet time step
# Returns the tuple of new positions, velocities, accelerations (forces), potential energy and pressure
def velocity_verlet(positions, velocities, accelerations,
                    time_step, potential, potential_gradient):
    # Update positions
    positions += velocities*time_step + 0.5*accelerations*time_step**2
    positions = positions - box_length*np.floor(positions/box_length)
    # Update velocities
    velocities_half = velocities + 0.5*accelerations*time_step
    # Compute new forces and potential energy
    accelerations, potential_energy, pressure = compute_forces(accelerations, positions, potential, potential_gradient)
    # Update velocities using new accelerations
    velocities = velocities_half + 0.5*accelerations*time_step
    # Add ideal gas contribution to the pressure
    kinetic_temperature = compute_kinetic_temperature(velocities)
    pressure += density * kinetic_temperature
    return positions, velocities, accelerations, potential_energy, pressure
```

Forces

We will assume all masses are equal to unity, $m_i = 1$ (dimensionless) and that the pair potential depends on the distance only. Then

$$\ddot{\mathbf{r}}_i = -\sum_{j \neq i} \frac{dV(r_{ij})}{dr_{ij}} \frac{\mathbf{r}_i - \mathbf{r}_j}{r_{ij}} .$$

```python
# Computes forces for a given vector of positions, interaction potential and its gradient
# Return a tuple: positions, total potential energy, and the virial part of the pressure
def compute_forces(forces, positions, potential, potential_gradient):
    # forces = np.zeros_like(positions)
    forces.fill(0.)
    potential_energy = 0.0
    virial = 0.0
    for i in range(n_particles):
        for j in range(i+1, n_particles):
            # Vector of relative distance
            r_ij = positions[i] - positions[j]
            # Periodic boundary conditions (minimum-image convention)
            r_ij = r_ij - box_length*np.round(r_ij/box_length)

            r_sq = np.sum(r_ij**2)
            f_ij = -potential_gradient(r_sq) * r_ij

            forces[i] += f_ij
            forces[j] -= f_ij
            potential_energy += potential(r_sq)
            virial += np.dot(f_ij, r_ij)

    virial = virial/(3.0*box_length**3)
    return forces, potential_energy, virial
```

Example: Lennard-Jones fluid

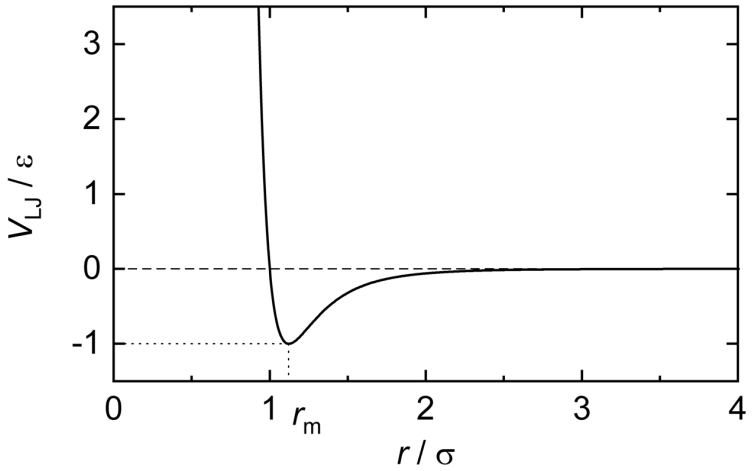
$$V_{\mathrm{LJ}}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

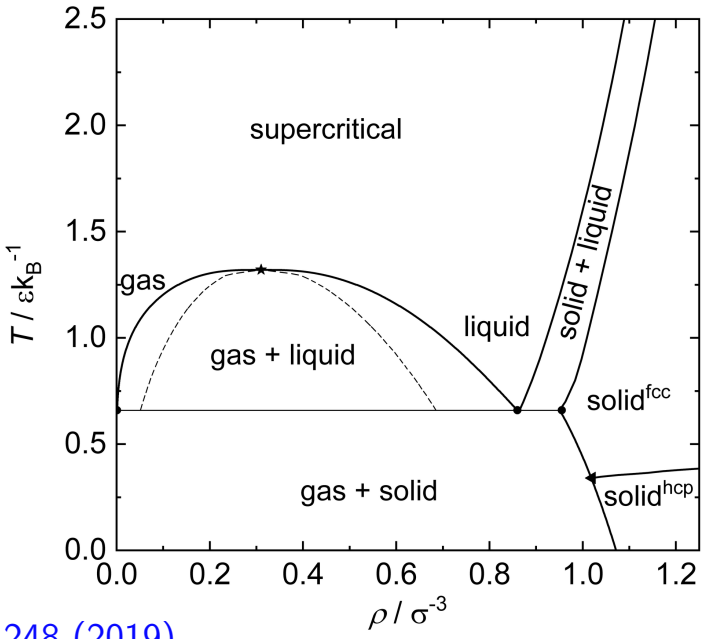
Reduced variables:

$$\tilde{r} = r/\sigma \qquad \tilde{T} = T/(k_B\varepsilon) \qquad \tilde{n} = n\sigma^3$$

Properties:

- Multiple phase transitions, including critical point
- Cannot be solved analytically
- Tractable with molecular dynamics simulations





S. Stephan, M. Thol, J. Vrabec, H. Hasse, Journal of Chemical Information and Modeling 59, 4248 (2019)

Example: Lennard-Jones fluid

$$\ddot{\mathbf{r}}_i = -\sum_{j \neq i} \frac{dV(r_{ij})}{dr_{ij}} \frac{\mathbf{r}_i - \mathbf{r}_j}{r_{ij}} .$$

```
# Lennard-Jones potential as a function of squared distance
def lj_potential(r_sq):
    r6 = r_sq**3
    r12 = r6**2
    return 4.0*(1./r12 - 1./r6)

# The grandient term dV/dr / r in the rhs of Newton's equations
# for the LJ potential
def lj_potential_gradient(r_sq):
    r6 = r_sq**3
    r12 = r6**2
    return -24.0*(2./r12 - 1./r6) / r_sq
```

Both the potential and the gradient term can be expressed in terms of $|r_i - r_j|^2$, saves the unnecessary computation of the square root

Simulation: Initial conditions

We have to initialize the system with initial positions and velocities

- Coordinates
  - Put particles in a grid
  - Avoids particle overlap (mind the r^{-12} term)

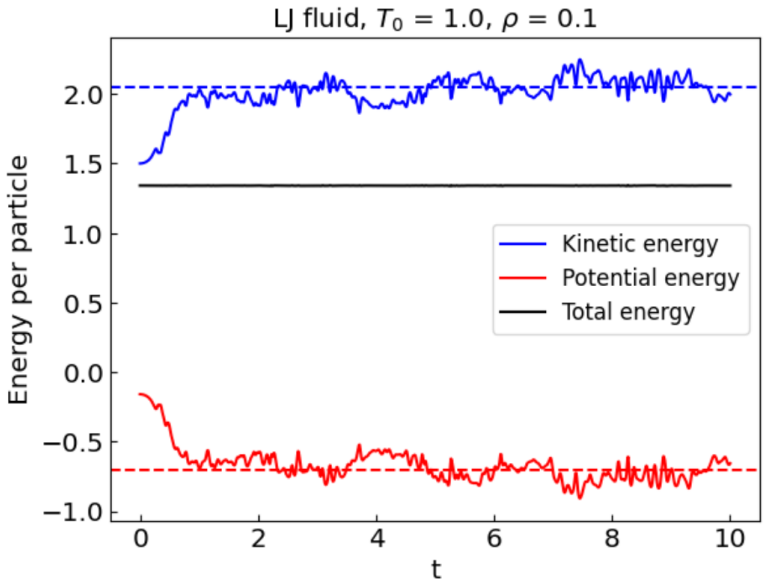
```
def initial_positions():
    ret = np.zeros((n_particles,3))
    Nsingle = np.ceil(n_particles**(1/3.))
    dL = box_length / Nsingle
    for i in range(n_particles):
        ix = i % Nsingle
        iy = (np.trunc(i / Nsingle)) % Nsingle
        iz = np.trunc(i / (Nsingle * Nsingle))
        ret[i][0] = (ix + 0.5) * dL;
        ret[i][1] = (iy + 0.5) * dL;
        ret[i][2] = (iz + 0.5) * dL;
    return ret
```

- Velocities
  - Sample each component from Gaussian (Maxwell-Boltzmann) distribution

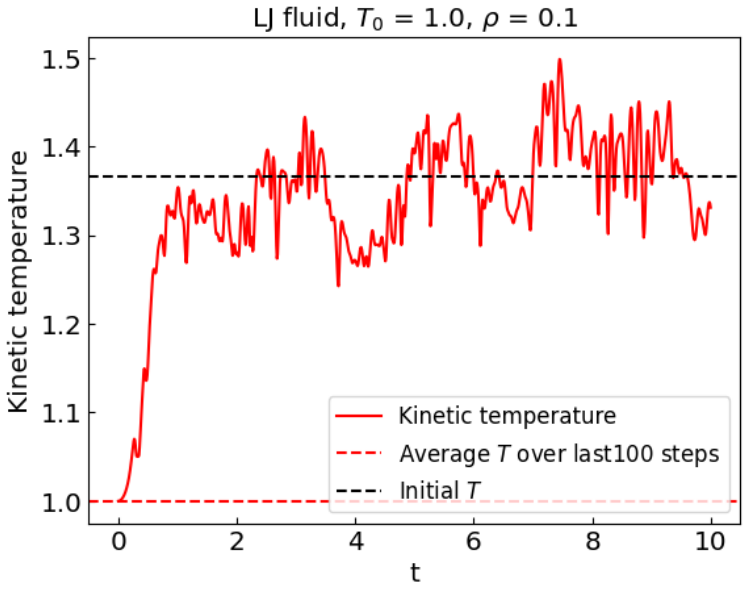
```
positions = initial_positions()
velocities = np.random.normal(loc=0.0, scale=np.sqrt(temperature0), size=(n_particles, 3))
```

Simulation

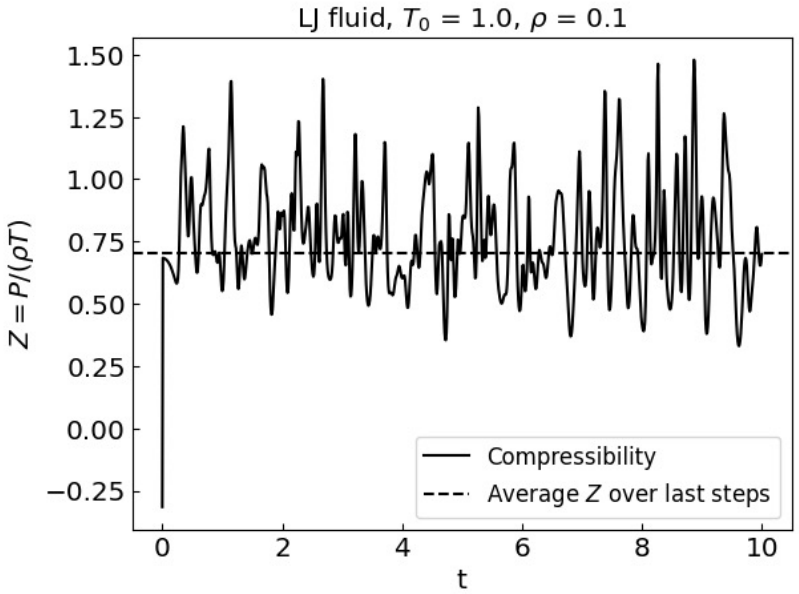
$T = 1,\ \rho = 0.1,\ N = 64$



$T_{\text{kin}} = \langle \mathbf{v}_i^2 \rangle / 3$



$Z = p/(\rho T)$



Kinetic temperature drifts away from initial value!

Reason: system takes time to equilibrate, and temperature is not conserved in microcanonical ensemble

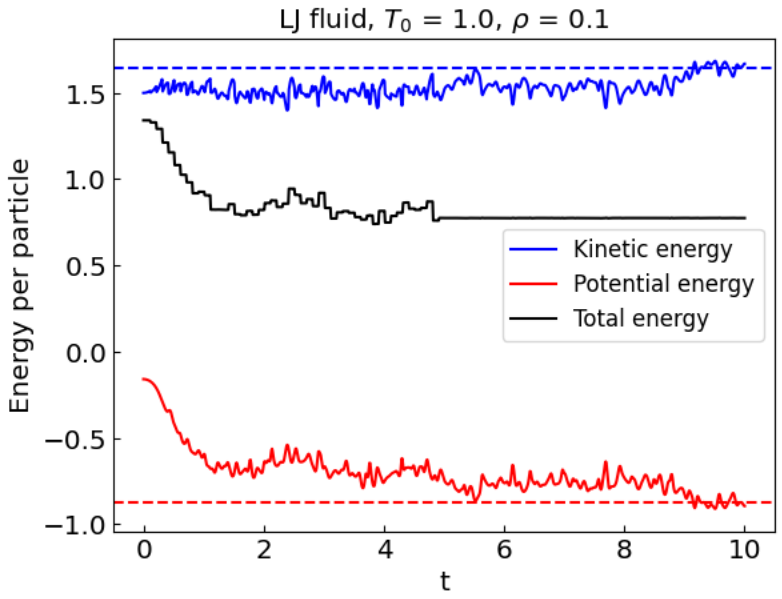
Simulation: Keep the temperature fixed

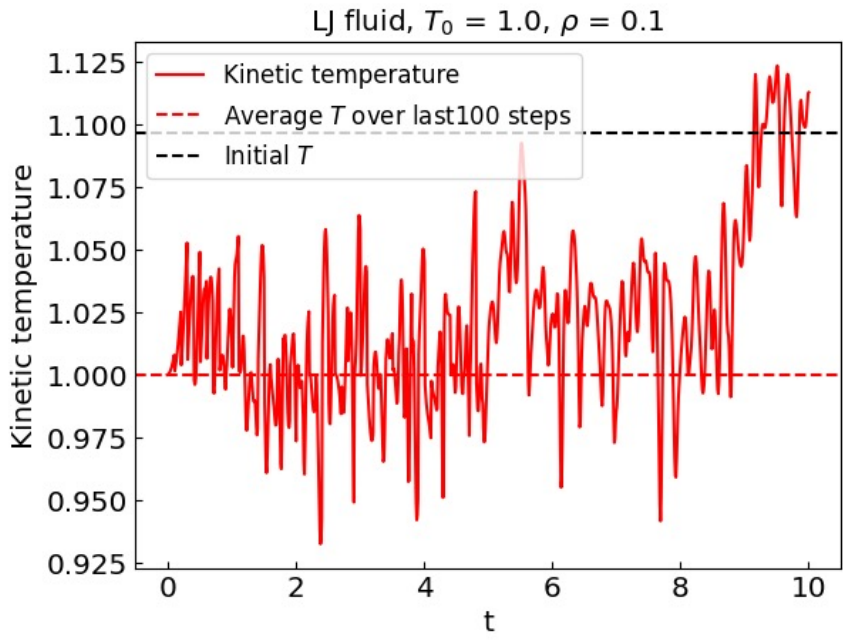
Keep the temperature fixed during the equilibration phase by periodically rescaling the velocities to have desired temperature

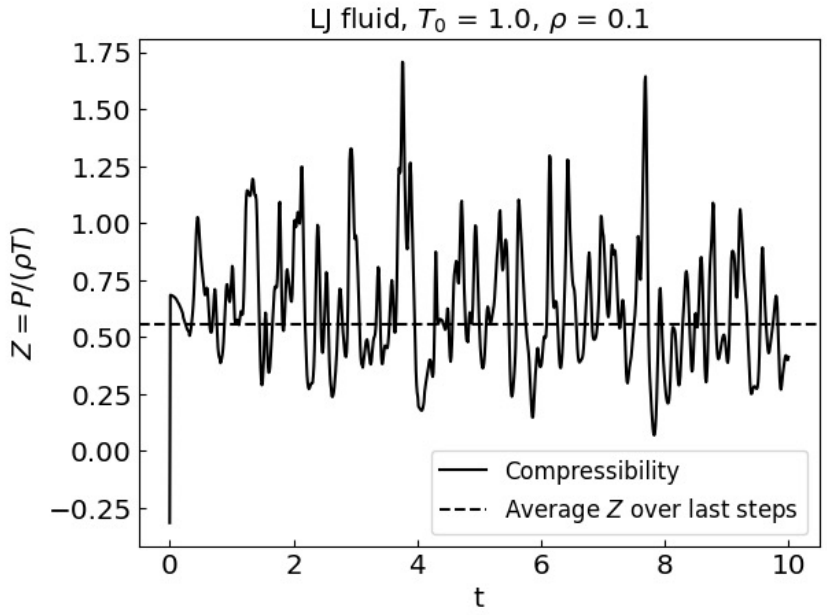
$\mathrm{T} = 1,\ \rho = 0.1,\ \mathrm{N} = 64$

$T_{\mathrm{kin}} = \langle \mathbf{v}_i^2 \rangle / 3$

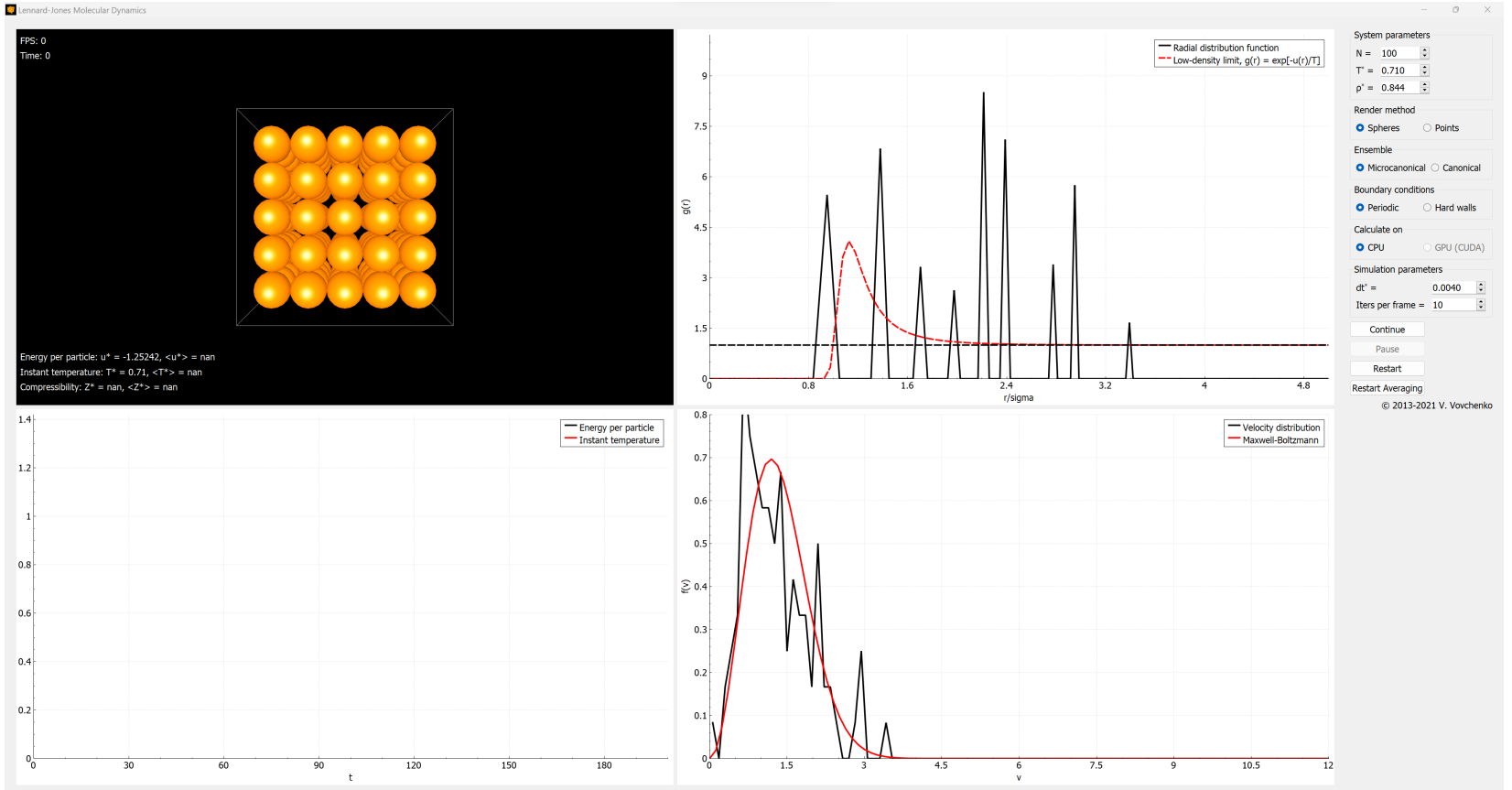
$Z = p/(\rho T)$







Lennard-Jones fluid: C++/GPU implementation



Implementation:

Velocity Verlet integration scheme implemented on CUDA-GPU (x100-200 speed-up*)

open source: https://github.com/vlvovch/lennard-jones-cuda

NVIDIA CUDA