The heat is on

- **Topic:** temperature, pressure, thermostats, measurements
- **Task:**
  1. Implement the Berendsen thermostat.
  2. Run a simulation using `fcc_tet` with and without the thermostat and with different values of τ. Study how temperature and energy behave in both cases.
  3. Measure the pressure. Study how the calculated pressure and its error estimate depend on (1) thermalization time, (2) total simulation time and (3) sample length.
  4. Study how the pressure depends on temperature. Also calculate the pressure in `fcc_compress.txt` and `fcc_stretch.txt` and compare to previous results.

- **Template:** temp.py
- **Data:**
  - fcc.txt
  - fcc_compress.txt
  - fcc_stretch.txt
- **Further reading:**
  - https://en.wikipedia.org/wiki/Nosé-Hoover_thermostat
  - https://en.wikipedia.org/wiki/Langevin_dynamics
  - https://en.wikipedia.org/wiki/Berendsen_thermostat

temp.py

class **temp.Atom**(*position*, *velocity*, *mass=1.0*) [source]

A point like object.

An atom has a position (a 3-vector), a velocity (3-vector) and a mass (a scalar).

Parameters:
- **position** (*array*) – coordinates $[x, y, z]$
- **velocity** (*array*) – velocity components $[v_x, v_y, v_z]$
- **mass** (*float*) – mass m

accelerate(*force*, *dt*) [source]

Set a new velocity for the particle as

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{2m}\vec{F}\Delta t$$

Parameters:
- **force** (*array*) – force acting on the planet $[F_x, F_y, F_z]$
- **dt** (*float*) – time step Δt

move(*force*, *dt*) [source]

Move the atom.

Parameters:
- **shift** (*array*) – coordinate change $[\Delta x, \Delta y, \Delta z]$

save_position() [source]

Save the current position of the particle in the list 'trajectory'.

Note: in a real large-scale simulation one would never save trajectories in memory. Instead, these would be written to a file for later analysis.

class **temp.PeriodicBox**(*lattice*) [source]

Class representing a simulation box with periodic boundaries.

The box is orthogonal, i.e., a rectangular volume. As such, it is specified by the lengths of its edges (lattice constants).

Parameters:
- **lattice** (*array*) – lattice constants

distance_squared(*particle1*, *particle2*) [source]

Calculates and returns the square of the distance between two particles.

$$r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2.$$

In a periodic system, each particle has an infinite number of periodic copies. Therefore the distance between two particles is not unique. The function returns the shortest such distance, that is, the distance between the the periodic copies which are closest ot each other.

Parameters:
- **particle1** (*Molecule* – the first body
- **particle2** (*Molecule* – the second body

Returns: the squared distance r_{ij}^2

Return type: float

shift_inside_box(*position*) [source]

If the given position (3-vector) is outside the box, it is shifted by multiple of lattice vectors until the new position is inside the box. That is, the function transforms the position vector to an equivalen position inside the box.

Parameters: **position** (*array*) – the position to be shifted

vector(*particle1*, *particle2*) [source]

Returns the vector pointing from the position of particle1 to the position of particle2.

$$\vec{r}_{1\to 2} = \vec{r}_2 - \vec{r}_1$$

In a periodic system, each particle has an infinite number of periodic copies. Therefore the displacement between two particles is not unique. The function returns the shortest such displacement vector.

Parameters:
- **particle1** (*Molecule* – the first body
- **particle2** (*Molecule* – the second body

Returns: components of $[x_{1\to 2}, y_{1\to 2}, z_{1\to 2}]$

Return type: array

temp.**animate**(*particles*, *box*, *multiply=(3, 3)*) [source]

Animates the simulation.

Parameters:
- **particles** (*list* – list of `temp.Atom` objects
- **box** (`temp.PeriodicBox`) – supercell
- **multiply** (*array*) – number of periodic images to draw in x and y directions

temp.**assign_maxwell_boltzmann_velocities**(*particles*, *temperature*) [source]

Randomly pick velocities for all particles. The velocities are chosen according to the Maxwell–Boltzmann distribution.

temp.**berendsen_thermostat**(*particles*, *dt*, *tau=5.0*, *T0=0.2*) [source]

Implements the velocity scaling of the Berendsen thermostat.

A thermostat is an algorithm which couples the simulated system to an external, fictious heat bath at some constant temperature T_0. If the system is hotter than this, the thermostat removes energy from the system. And vice versa, if the system is cooler than the heat bath, energy is brought in the system.

The Berendsen thermostat aims at scaling the temperature T of the system according to

$$\frac{dT}{dt} = \frac{1}{\tau}(T_0 - T),$$

where t is time and τ is a time constant. This makes T approach T_0 exponentially.

In practice, the temperature is changed by scaling all velocities with a scaling factor λ using `scale_velocities()`. In time step Δt one expects approximately

$$\Delta T = \frac{\Delta t}{\tau}(T_0 - T)$$

and solving for λ from the definition of kinetic temperature yields.

$$\lambda = \sqrt{1 + \frac{\Delta t}{\tau}\left[\frac{T_0}{T} - 1\right]}.$$

Note

This function is incomplete!

Parameters:
- **dt** (*float*) – timestep Δt
- **tau** (*float* – time constant τ
- **t0** (*float* – external temperature T_0

temp.**calculate_average_and_error**(*values*, *start=0*) [source]

Calculates the average and standard error of mean of a sequence.

The values in the sequence are assumed to be uncorrelated.

If the beginning of the sequence cannot be used in the analysis (equilibrium has not yet been reached), one can ignore the early values by specifying a starting index.

Parameters:
- **values** (*array*) - values to analyse
- **start** (*int*) - index of the first value to be included in the analysis

temp.**calculate_forces**(*particles*, *box*, *sigma=1.0*, *epsilon=0.1*, *cutoff=2.5*) [source]

Calculates the total force applied on each atom.

The forces are returned as a numpy array where each row contains the force on an atom and the columns contain the x, y and z components.

```
[ [ fx0, fy0, fz0 ],
  [ fx1, fy1, fz1 ],
  [ fx2, fy2, fz2 ],
  ...                ]
```

The function also calculates the virial,

$$\sum_{i<j} \vec{U}(\vec{r}_{ij})\vec{r}_{ij},$$

which is needed for pressure calculation.

Parameters:
- **atoms** (*list* – a list of `temp.Atom` objects
- **box** (`temp.PeriodicBox`) – supercell
- **sigma** (*float* - Lennard-Jones parameter σ
- **epsilon** (*float* - Lennard-Jones parameter ϵ
- **cutoff** (*float* - maximum distance for interactions

Returns: forces, virial

Return type: array, float

temp.**calculate_kinetic_energy**(*particles*) [source]

Calculates the total kinetic energy of the system.

$$K_{total} = \sum_i \frac{1}{2} m_i v_i^2.$$

Parameters: **particles** (*list* – a list of `temp.Atom` objects

Returns: kinetic energy K

Return type: float

temp.**calculate_momentum**(*particles*) [source]

Calculates the total momentum of the system.

$$\vec{p}_{total} = \sum_i \vec{p}_i = \sum_i m_i \vec{v}_i$$

Parameters: **particles** (*list* – a list of `Planet` objects

Returns: momentum components $[p_x, p_y, p_z]$

Return type: array

temp.**calculate_potential_energy**(*particles*, *box*, *sigma=1.0*, *epsilon=0.1*, *cutoff=2.5*) [source]

Calculates the total potential energy of the system.

The potential energy is calculated using the Lennard-Jones model

$$U = \sum_{i<j} 4\epsilon \left(\frac{\sigma^{12}}{r_{ij}^{12}} - \frac{\sigma^6}{r_{ij}^6} \right).$$

Parameters:
- **particles** (*list* – a list of `temp.Atom` objects
- **box** (`temp.PeriodicBox`) – supercell
- **sigma** (*float* - Lennard-Jones parameter σ
- **epsilon** (*float* - Lennard-Jones parameter ϵ
- **cutoff** (*float* - maximum distance for interactions

Returns: potential energy U

Return type: float

temp.**calculate_pressure**(*particles*, *box*, *virial*, *temperature*) [source]

Calculate the current pressure.

For a molecular simulation with constant pressure, volume and temperature, one can derive the relation

$$pV = Nk_BT + \frac{1}{d}\left\langle \sum_i \vec{F}_i \cdot \vec{r}_i \right\rangle,$$

where $p, V, N, k_B, T, d, \vec{F}_i, \vec{r}_i$ are, respectively, pressure, volume, number of atoms, Boltzmann constant, temperature, number of dimensions, force acting on atom i and position of atom i.

This function uses this relation to solve the effective instantaneous pressure as

$$p = \frac{1}{V}Nk_BT + \frac{1}{dV}\sum_i \vec{F}_i \cdot \vec{r}_i.$$

where the sum is called the virial.

This is not necessarily the true instantaneous pressure, but calculating the average of this quantity over an extended simulation should converge towards the true pressure.

Parameters:
- **particles** (*list* – list of `temp.Atom` objects
- **box** (`temp.PeriodicBox`) – supercell
- **virial** (*float* - virial
- **temperature** (*float* - temperature

Returns: pressure

Return type: float

temp.**calculate_temperature**(*particles*) [source]

Calculate and return the current temperature.

temp.**draw**(*frame*, *xtraj*, *ytraj*, *ztraj*, *bounds*) [source]

Draws a representation of the particle system as a scatter plot.

Used for animation.

Parameters:
- **frame** (*int* - index of the frame to be drawn
- **xtraj** (*array* - x-coordinates of all particles at different animation frames
- **ytraj** (*array* - y-coordinates at all particles at different animation frames
- **ztraj** (*array* - z-coordinates at all particles at different animation frames
- **bounds** (*array* - list of lower and upper bounds for the plot as [xmin, xmax], [ymin, ymax]]

temp.**expand_supercell**(*particles*, *box*, *multiplier*) [source]

Expands a periodic system.

The periodic system is represented by the particles and the box. This method creates a new, similar system, which is larger by the factor 'multiplier' in x, y, and z directions. That is, the list 'particles' will be expanded by a factor of multiplier^3.

For example, if particles contains Atoms at positions [0.0,0] and [1,1,1], and box is a cube with edge length 2, calling this function with multiplier = 2 will change particles to contain the Atoms at pocitions [0.0,0], [1,1,1], [2,0,0], [1,1,1], [0,2,0], [1,3,1], [0,0,2], [1,1,1], [2,2,0], [3,3,1], [2,0,2], [3,1,3], [0,2,2], [1,3,3], [2,2,2], [3,3,3] and box will be expanded to a square with edge lengths 4.

temp.**main**(*filename*, *external_temperature*, *tau*, *dt*, *sample_interval*, *simulation_time*, *thermalization_time*) [source]

The main program.

The program reads the system from a file, runs the simulation.

Atomic velocities are initialized according to the given temperature. In addition, if a time constant is given, a thermostat is applied to drive temperature towards this temperature.

Parameters:
- **filename** (*str*) - name of the file to read
- **external_temperature** (*float*) - temperature
- **tau** (*float* - thermostat time constant

temp.**print_progress**(*step*, *total*) [source]

Prints a progress bar.

Parameters:
- **step** (*int*) - progress counter
- **total** (*int*) - counter at completion

temp.**read_particles_from_file**(*filename*) [source]

Reads the properties of planets from a file.

Each line should define a single `Planet` listing its position in cartesian coordinates, velocity components and mass, separated by whitespace.

```
x0 y0 z0 vx0 vy0 vz0 m0
x1 y1 z1 vx1 vy1 vz1 m1
x2 y2 z2 vx2 vy2 vz2 m2
x3 y3 z3 vx3 vy3 vz3 m3
...
```

Parameters: **filename** (*str*) – name of the file to read

Returns: list of `temp.Atom` objects

Return type: list

temp.**scale_velocities**(*particles*, *scale_factor*) [source]

Scale the velocities of all particles by the scaling factor.

Parameters:
- **particles** (*list* – a list of `temp.Atom` objects
- **scale_factor** (*float* – scaling factor

temp.**show_particles**(*particles*) [source]

Plot a 2D-projection (xy-coordinates) of the system.

Parameters: **particles** (*list* – list of `temp.Atom` objects

temp.**show_trajectories**(*particles*, *box*, *tail=10*, *skip=10*, *multiply=(3, 3)*) [source]

Plot a 2D-projection of the trajectory of the system.

The function creates a plot showing the current and past positions of particles.

Parameters:
- **particles** (*list* - list of `Planet` objects
- **box** (`temp.PeriodicBox`) – supercell
- **tail** (*int* - the number of past positions to include in the plot
- **skip** (*int* – only every nth past position is plotted – skip is the number n, specifying how far apart the plotted positions are in time
- **multiply** (*array* - number of periodic images to draw in x and y directions

temp.**update_positions**(*particles*, *forces*, *dt*) [source]

Update the positions of all particles using `temp.Atom.move()` according to

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}\Delta t + \frac{1}{2m}\vec{F}(\Delta t)^2$$

Parameters:
- **particles** (*list* - a list of `Planet` objects
- **forces** (*array*) - array of forces on all bodies
- **dt** (*float* - time step Δt

temp.**update_positions_no_force**(*particles*, *dt*) [source]

Update the positions of all particles using `temp.Atom.move()` according to

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}\Delta t$$

Parameters:
- **particles** (*list* - a list of `Planet` objects
- **dt** (*float* - time step Δt

temp.**update_velocities**(*particles*, *forces*, *dt*) [source]

Update the positions of all particles using `temp.Atom.accelerate()` according to

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{m}\vec{F}\Delta t$$

Parameters:
- **particles** (*list* - a list of `Planet` objects
- **forces** (*array*) - array of forces on all bodies
- **dt** (*float* - time step Δt

temp.**velocity_verlet**(*particles*, *box*, *dt*, *time*, *trajectory_dt=1.0*, *temperature=0*, *tau=0*) [source]

Verlet algorithm for integrating the equations of motion, i.e., advancing time.

There are a few ways to implement Verlet. The leapfrog version works as follows: First, forces are calculated for all particles and velocities are updated by half a time step, $\vec{v}(t + \frac{1}{2}\Delta t) = \vec{v}(t) + \frac{1}{2m}\vec{F}\Delta t$. Then, these steps are repeated:

- Positions are updated by a full time step using velocities but not forces,

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t + \frac{1}{2}\Delta t)\Delta t.$$

- Forces are calculated at the new positions, $\vec{F}(t + \Delta t)$.
- Velocities are updated by a full time step using the forces

$$\vec{v}(t + \frac{3}{2}\Delta t) = \vec{v}(t + \frac{1}{2}\Delta t) + \frac{1}{m}\vec{F}(t + \Delta t)\Delta t$$

These operations are done using the methods `calculate_forces()`, `update_velocities()` and `update_positions_no_force()`.

Because velocities were updated by half a time step in the beginning of the simulation, positions and velocities are always offset by half a timestep. You always use the one that has advanced further to update the other and this results in a stable algorithm.

Parameters:
- **particles** (*list* – a list of `Planet` objects
- **box** (`temp.PeriodicBox`) – supercell
- **dt** (*float* - time step Δt
- **time** (*float* - the total system time to be simulated
- **trajectory_dt** (*float* - the positions of particles are saved at these these time intervals – does not affect the dynamics in any way

temp.**write_particles_to_file**(*particles*, *box*, *filename*) [source]

Write the configuration of the system in a file.

The format is the same as that specified in `read_particles_from_file()`.

Parameters:
- **particles** (*list* – list of `temp.Atom` objects
- **box** (`temp.PeriodicBox`) – supercell
- **filename** (*str*) - name of the file to write

temp.**write_xyz_file**(*particles*, *filename*) [source]

Write the configuration of the system in a file.

The information is written in so called xyz format, which many programs can parse.

Parameters:
- **particles** (*list* – list of `Planet` objects
- **filename** (*str*) - name of the file to write