FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF SCIENCES

Molecular Dynamics Program

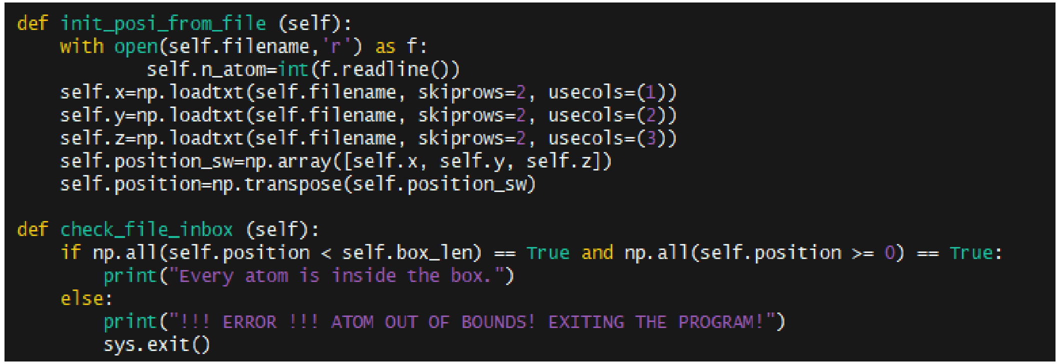
Azad Kirsan, 01.07.2021

1. Aims of the program

- Simulate Argon gas particles
- Potential: Lennard-Jones / hard spheres
- Use Velocity-Verlet-algorithm as integrator
- Periodic boundary conditions and hard walls
- Visualize the simulation

2.1. Initialization of coordinates from file

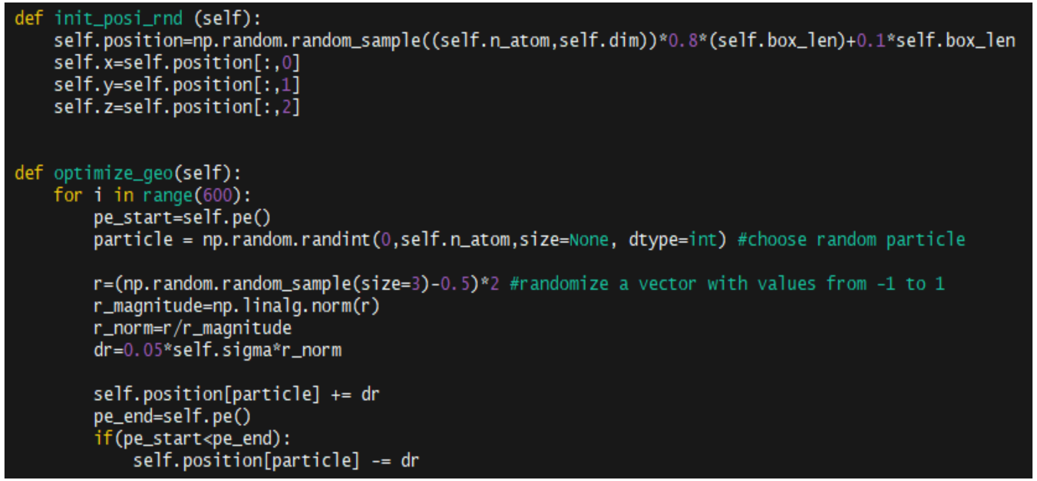
- Load atom coordinates into an array from a given file
- Check if all atoms are inside the box



```python
def init_posi_from_file (self):
    with open(self.filename,'r') as f:
            self.n_atom=int(f.readline())
    self.x=np.loadtxt(self.filename, skiprows=2, usecols=(1))
    self.y=np.loadtxt(self.filename, skiprows=2, usecols=(2))
    self.z=np.loadtxt(self.filename, skiprows=2, usecols=(3))
    self.position_sw=np.array([self.x, self.y, self.z])
    self.position=np.transpose(self.position_sw)

def check_file_inbox (self):
    if np.all(self.position < self.box_len) == True and np.all(self.position >= 0) == True:
        print("Every atom is inside the box.")
    else:
        print("!!! ERROR !!! ATOM OUT OF BOUNDS! EXITING THE PROGRAM!")
        sys.exit()
```

2.2. Initialization of random coordinates

- Randomize atom positions inside the box
- Optimize the geometry so no atoms are on top of each other



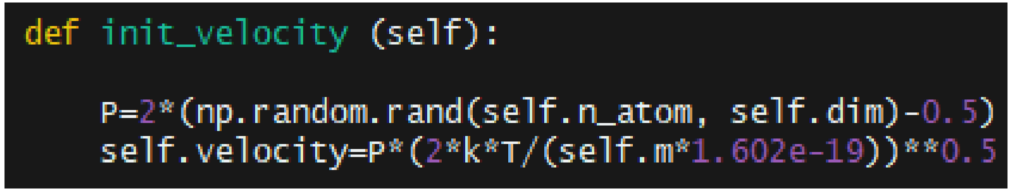
```
def init_posi_rnd (self):
    self.position=np.random.random_sample((self.n_atom,self.dim))*0.8*(self.box_len)+0.1*self.box_len
    self.x=self.position[:,0]
    self.y=self.position[:,1]
    self.z=self.position[:,2]


def optimize_geo(self):
    for i in range(600):
        pe_start=self.pe()
        particle = np.random.randint(0,self.n_atom,size=None, dtype=int) #choose random particle

        r=(np.random.random_sample(size=3)-0.5)*2 #randomize a vector with values from -1 to 1
        r_magnitude=np.linalg.norm(r)
        r_norm=r/r_magnitude
        dr=0.05*self.sigma*r_norm

        self.position[particle] += dr
        pe_end=self.pe()
        if(pe_start<pe_end):
            self.position[particle] -= dr
```

2.3. Initialization of velocities

Initialize random velocities relating to the temperature

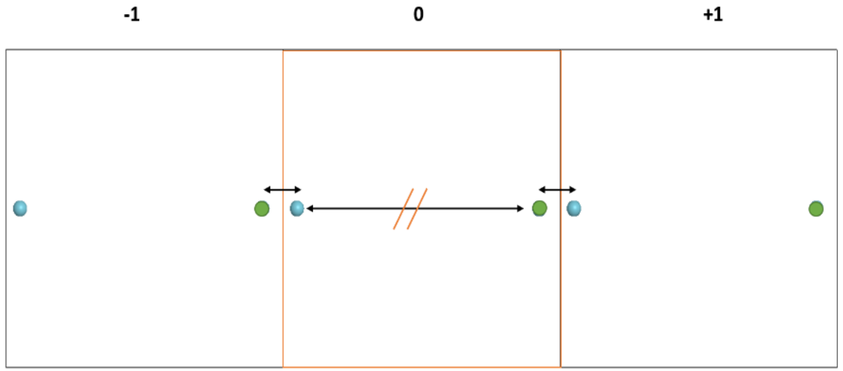
- Array of random numbers from -1 to 1 in a uniform distribution

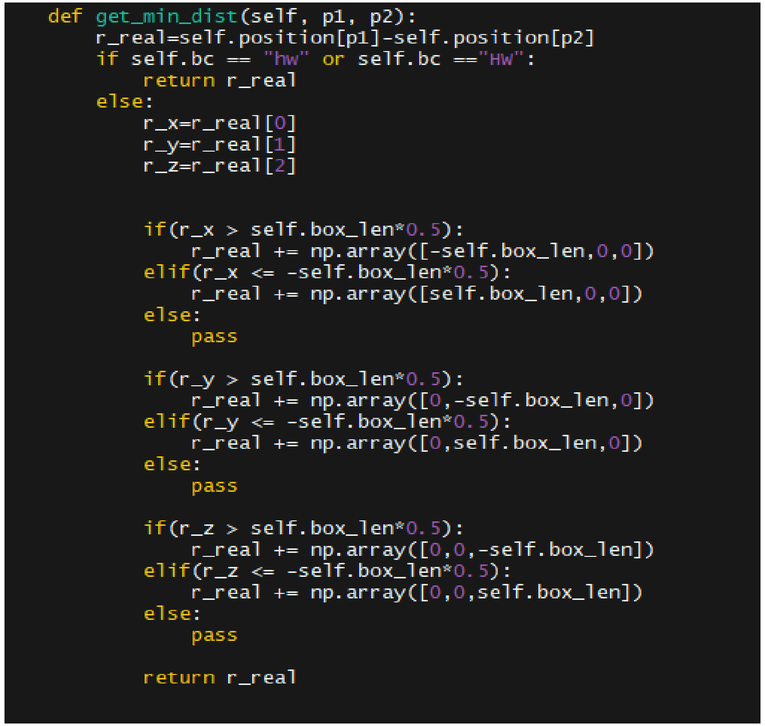
- Multiply array by the most probable velocity according to the Maxwell-Boltzmann distribution



```
def init_velocity (self):

    P=2*(np.random.rand(self.n_atom, self.dim)-0.5)
    self.velocity=P*(2*k*T/(self.m*1.602e-19))**0.5
```

3. Get the minimum distance



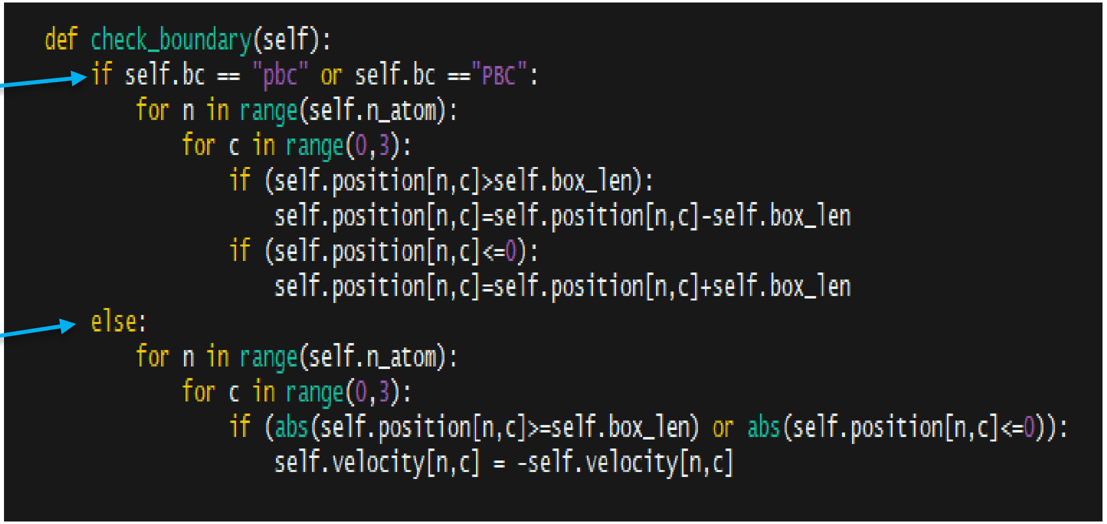


```
def get_min_dist(self, p1, p2):
    r_real=self.position[p1]-self.position[p2]
    if self.bc == "hw" or self.bc =="HW":
        return r_real
    else:
        r_x=r_real[0]
        r_y=r_real[1]
        r_z=r_real[2]


        if(r_x > self.box_len*0.5):
            r_real += np.array([-self.box_len,0,0])
        elif(r_x <= -self.box_len*0.5):
            r_real += np.array([self.box_len,0,0])
        else:
            pass

        if(r_y > self.box_len*0.5):
            r_real += np.array([0,-self.box_len,0])
        elif(r_y <= -self.box_len*0.5):
            r_real += np.array([0,self.box_len,0])
        else:
            pass

        if(r_z > self.box_len*0.5):
            r_real += np.array([0,0,-self.box_len])
        elif(r_z <= -self.box_len*0.5):
            r_real += np.array([0,0,self.box_len])
        else:
            pass

        return r_real
```

4. Boundary conditions

Periodic boundary conditions

Hard walls



```
def check_boundary(self):
    if self.bc == "pbc" or self.bc =="PBC":
        for n in range(self.n_atom):
            for c in range(0,3):
                if (self.position[n,c]>self.box_len):
                    self.position[n,c]=self.position[n,c]-self.box_len
                if (self.position[n,c]<=0):
                    self.position[n,c]=self.position[n,c]+self.box_len
    else:
        for n in range(self.n_atom):
            for c in range(0,3):
                if (abs(self.position[n,c]>=self.box_len) or abs(self.position[n,c]<=0)):
                    self.velocity[n,c] = -self.velocity[n,c]
```

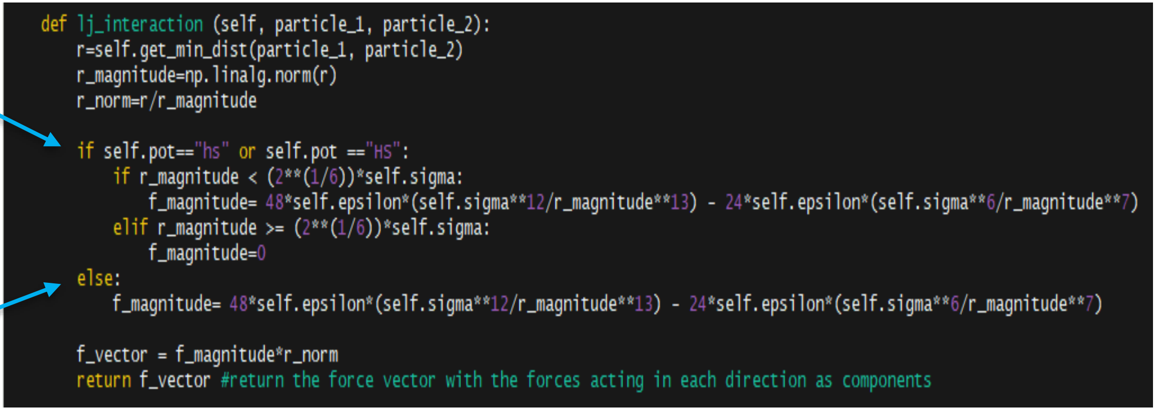
5. Calculating the forces

Weeks-Chandler-Andersen potential (hard spheres)

$$u_{\mathrm{WCA}}(r) = \begin{cases} 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] + \epsilon & r < 2^{\frac{1}{6}}\sigma \\ 0 & r \geq 2^{\frac{1}{6}}\sigma \end{cases}$$

Lennard-Jones potential

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



```
def lj_interaction (self, particle_1, particle_2):
    r=self.get_min_dist(particle_1, particle_2)
    r_magnitude=np.linalg.norm(r)
    r_norm=r/r_magnitude

    if self.pot=="hs" or self.pot =="HS":
        if r_magnitude < (2**(1/6))*self.sigma:
            f_magnitude= 48*self.epsilon*(self.sigma**12/r_magnitude**13) - 24*self.epsilon*(self.sigma**6/r_magnitude**7)
        elif r_magnitude >= (2**(1/6))*self.sigma:
            f_magnitude=0
    else:
        f_magnitude= 48*self.epsilon*(self.sigma**12/r_magnitude**13) - 24*self.epsilon*(self.sigma**6/r_magnitude**7)

    f_vector = f_magnitude*r_norm
    return f_vector #return the force vector with the forces acting in each direction as components
```

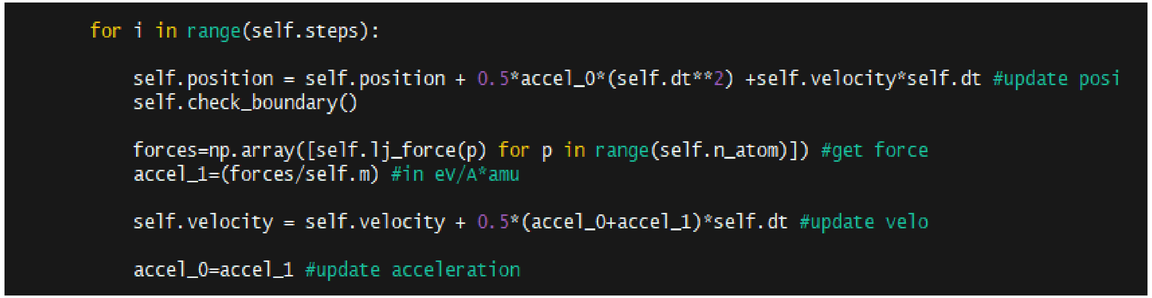
6. Velocity-Verlet Integrator

$$a)\ x \leftarrow x + v\Delta t + \frac{1}{2}a_0\Delta t^2$$

$$b)\ a_1 \leftarrow \frac{F}{m}$$

$$c)\ v \leftarrow v + \frac{1}{2}[a_1 + a_0]\Delta t$$

$$d)\ a_0 \leftarrow a_1$$



```python
for i in range(self.steps):

    self.position = self.position + 0.5*accel_0*(self.dt**2) +self.velocity*self.dt #update posi
    self.check_boundary()

    forces=np.array([self.lj_force(p) for p in range(self.n_atom)]) #get force
    accel_1=(forces/self.m) #in eV/A*amu

    self.velocity = self.velocity + 0.5*(accel_0+accel_1)*self.dt #update velo

    accel_0=accel_1 #update acceleration
```

7. Problems

- Initializing useful random velocities
- Getting the units right
- Visualizing the simulation, since matplotlib was too slow