MolDyn

February 11, 2020

1 Molecular Dynamics Project

1.1 1 Introduction

1.1.1 a) Understanding the potential

i) Program som plotter potetialet:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

def U(r,  ,  ):
    return 4* *(( /r)**12 - ( /r)**6)

r_ray = np.linspace(0.9, 3, 100)
U_ray = U(r_ray, 1, 1)

plt.figure(figsize=(8, 6))
plt.plot(r_ray, U_ray)
plt.show()
```

```
<Figure size 800x600 with 1 Axes>
```

ii) Når $r < \sigma$ er $\frac{\sigma}{r} > 1$, så $(\frac{\sigma}{r})^{12}$ dominerer, mens hvis $r > \sigma$ så er $\frac{\sigma}{r} < 1$, og $(\frac{\sigma}{r})^6$ blir relativt mye større enn $(\frac{\sigma}{r})^{12}$. Dette fører til at grafen vokser stert når $r < \sigma$, og synker når $r > \sigma$.

iii) Når r går mot 0 går U mot ∞, når r går mot σ går U mot 0, når r går mot $\sqrt[6]{2}$ har U et bunnpunkt, og når r blir stor går U mot 0.

iv) Hvis vi har to atomer med en avstand på 1.5 blir U et negativt tall, så de vil bli tiltrukket til hverandre. Etter hvert vil r bli så lavt at U blir et positivt tall, og de vlil frastøte hverandre. Ettersom total energi er bevart bør ikke atomene stoppe opp, så de vil veksle rundt likevektspunktet i evig tid. Hvis de starter med en avstand på 0.95 vil de starte med å frastøte hverandre, før de vil tiltrekke hverande og igjen gå frem og tilbake i evig tid.

v) Rett til venste for likevektspunktet er kraften positiv, mens til høyre for likevektspunktet er kraften negativ. Kan jeg tenke på andre krefter med lignende oppførsel?

1.1.2 b) Forces and equations of motion

i) Vi har to atomer, i og j. Disse atomene har positioner $\vec{r}_i$ og $\vec{r}_j$. Avstanden mellom atomene blir da $r = |\vec{r}_i - \vec{r}_j|$.
For å gjøre om potensialet til en kraft har man at kraften er minus den deriverte av potentialet med hensyn til r.
Deriverer med wolfram alpha:
$\frac{dU}{dr} = 24\varepsilon\left(\left(\frac{\sigma}{r}\right)^6 - 2\left(\frac{\sigma}{r}\right)^{12}\right)\frac{1}{r}$ Setter så inn formelen til $r = |\vec{r}_i - \vec{r}_j|$ for å få kraften på atom i fra atom j:

$$F = -24\varepsilon\left(\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^6 - 2\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^{12}\right)\frac{1}{|\vec{r}_i - \vec{r}_j|}$$

ii) For å finne en formel for bevegelsen til atom i må vi bruke newtons andre lov, $F = ma$. Først vil vi gjøre om kraften vi fant i forigje deloppgave til en kraftvektor. Dette er enkelt, ettersom kraften på atom i kommer fra atom j, altså er retningsvektoren $\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|}$. Ganger vi dette med formelen for kraften, får vi det endelige uttrykket for kraftvektoren:

$$F\cdot\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|} = -24\varepsilon\left(\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^6 - 2\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^{12}\right)\frac{1}{|\vec{r}_i - \vec{r}_j|}\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|} = 24\varepsilon\left(2\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^{12} - \left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^6\right)\frac{i}{|\vec{r}}$$

Ettersom $F = ma$ får i en akselerasjon på $\frac{F}{m} = \frac{24\varepsilon}{m}\left(2\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^{12} - \left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^6\right)\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^2}$. For å generalisere denne formelen til flere enn to atomer må vi summere over alle andre atomer, så formelen for bevegelse for atom i er:

$$\frac{24\varepsilon}{m}\sum_{j\neq i}\left(2\left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^{12} - \left(\frac{\sigma}{|\vec{r}_i - \vec{r}_j|}\right)^6\right)\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^2}$$

1.1.3 c) Units

i)

$$\vec{r}_i\,' = \vec{r}_i/\sigma \implies \vec{r}_i = \vec{r}_i\,'\sigma \tag{1}$$

$$\implies |\vec{r}_i - \vec{r}_j| = |\vec{r}_i\,'\sigma - \vec{r}_i\,'\sigma| = |\sigma(\vec{r}_i\,' - \vec{r}_i\,')| = \sigma|\vec{r}_i\,' - \vec{r}_i\,'| \text{ (ettersom } \sigma \text{ er positiv)} \tag{2}$$

Bruker dette for å skrive likningen på skalert form:

$$\frac{d^2\vec{r}_i}{dt^2} = \frac{24\varepsilon}{m}\sum_{j\neq i}\left(2\left(\frac{\sigma}{|\vec{r}_i-\vec{r}_j|}\right)^{12}-\left(\frac{\sigma}{|\vec{r}_i-\vec{r}_j|}\right)^{6}\right)\frac{\vec{r}_i-\vec{r}_j}{|\vec{r}_i-\vec{r}_j|^2} \tag{3}$$

$$\frac{d^2\vec{r}_i{}'\sigma}{dt^2} = \frac{24\varepsilon}{m}\sum_{j\neq i}\left(2\left(\frac{\sigma}{\sigma|\vec{r}_i{}'-\vec{r}_i{}'|}\right)^{12}-\left(\frac{\sigma}{\sigma|\vec{r}_i{}'-\vec{r}_i{}'|}\right)^{6}\right)\frac{\sigma(\vec{r}_i{}'-\vec{r}_i{}')}{\sigma^2|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{4}$$

$$\frac{d^2\vec{r}_i{}'\sigma}{dt^2} = \frac{24\varepsilon}{m}\sum_{j\neq i}\left(2\left(\frac{1}{|\vec{r}_i{}'-\vec{r}_i{}'|}\right)^{12}-\left(\frac{1}{|\vec{r}_i{}'-\vec{r}_i{}'|}\right)^{6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{\sigma|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{5}$$

$$\frac{d^2\vec{r}_i{}'}{dt^2} = \frac{24\varepsilon}{m\sigma^2}\sum_{j\neq i}\left(2|\vec{r}_i{}'-\vec{r}_i{}'|^{-12}-|\vec{r}_i{}'-\vec{r}_i{}'|^{-6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{6}$$

Hvis vi også innfører $t' = t/\tau$ kan vi forenkle uttrukket:

$$\frac{d^2\vec{r}_i{}'}{dt^2} = \frac{24\varepsilon}{m\sigma^2}\sum_{j\neq i}\left(2|\vec{r}_i{}'-\vec{r}_i{}'|^{-12}-|\vec{r}_i{}'-\vec{r}_i{}'|^{-6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{7}$$

$$\frac{d^2\vec{r}_i{}'}{dt'^2\tau^2} = \frac{24\varepsilon}{m\sigma^2}\sum_{j\neq i}\left(2|\vec{r}_i{}'-\vec{r}_i{}'|^{-12}-|\vec{r}_i{}'-\vec{r}_i{}'|^{-6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{8}$$

$$\frac{d^2\vec{r}_i{}'}{dt'^2} = 24\tau^2\frac{\varepsilon}{m\sigma^2}\sum_{j\neq i}\left(2|\vec{r}_i{}'-\vec{r}_i{}'|^{-12}-|\vec{r}_i{}'-\vec{r}_i{}'|^{-6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{9}$$

$$\tag{10}$$

For å forenkle uttrykket kan vi fjærne $\frac{\varepsilon}{m\sigma^2}$ ved å velge en τ slik at $\tau^2 \cdot \frac{\varepsilon}{m\sigma^2} = 1$:

$$\tau^2 \cdot \frac{\varepsilon}{m\sigma^2} = 1 \tag{11}$$

$$\tau^2 = \frac{m\sigma^2}{\varepsilon} \tag{12}$$

$$\tau = \sqrt{\frac{m\sigma^2}{\varepsilon}} \tag{13}$$

Hvis vi putter dette inn i likningen får vi vår skalerte likning:

$$\frac{d^2\vec{r}_i{}'}{dt'^2} = 24\tau^2\frac{\varepsilon}{m\sigma^2}\sum_{j\neq i}\left(2|\vec{r}_i{}'-\vec{r}_i{}'|^{-12}-|\vec{r}_i{}'-\vec{r}_i{}'|^{-6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{14}$$

$$\frac{d^2\vec{r}_i{}'}{dt'^2} = 24\sum_{j\neq i}\left(2|\vec{r}_i{}'-\vec{r}_i{}'|^{-12}-|\vec{r}_i{}'-\vec{r}_i{}'|^{-6}\right)\frac{\vec{r}_i{}'-\vec{r}_i{}'}{|\vec{r}_i{}'-\vec{r}_i{}'|^2} \tag{15}$$

ii) Den karakteristiske tidsskalaen τ for argon er:

$$\tau = \sqrt{\frac{m\sigma^2}{\varepsilon}} \tag{16}$$

$$= \sqrt{\frac{39.95u \cdot 3.405^{22}}{1.0318 \cdot 10{-}2eV}} \tag{17}$$

$$= \sqrt{\frac{(39.95 \cdot 1.66 \cdot 10^{-27})\ kg \cdot (3.405 \cdot 1 \cdot 10^{-10})^2\ m^2}{(1.0318 \cdot 10^{-2} \cdot 1.602 \cdot 10^{-19})\ J}} \tag{18}$$

$$= 2.16 \cdot 10^{-12} \tag{19}$$

Som er 2.16 picosekunder.
Enheten til τ er $\sqrt{\frac{kgm^2}{J}} = \sqrt{\frac{kgm^2}{kgm^2/s^2}} = \sqrt{s^2} = s$, som gir mening.

1.2 2 Two-atom simulations

1.2.1 a) Implementation

i) Skrev et program:

[2]:
```
import itertools
from functools import reduce
import time
import matplotlib.pyplot as plt
from vector import Vector3

class Atom:
        def __init__(self, pos, vel=None):
                self.force = Vector3()
                self.acc = Vector3()
                if vel is None:
                        self.vel = Vector3()
                self.pos = pos

        def update(self, dt, func="chromer"):
                if func == "chromer":
                        self.update_chromer(dt)
                elif func == "euler":
                        self.update_euler(dt)
                elif func == "verlet":
                        self.update_verlet(dt)
                else:
                        print(">:[")

        def update_euler(self, dt):
                """
```

```
                r[i+1] = r[i] + v[i]*dt
                v[i+1] = v[i] + a[i]*dt
                """
                acc = self.force
                self.pos += self.vel*dt
                self.vel += acc*dt
                self.force.set(0, 0, 0)

        def update_chromer(self, dt):
                """
                v[i+1] = v[i] + a[i]*dt
                r[i+1] = r[i] + v[i+1]*dt
                """
                acc = self.force
                self.vel += acc*dt
                self.pos += self.vel*dt
                self.force.set(0, 0, 0)

        def update_verlet(self, dt):
                """
                v[i] = v[i-1] + 0.5*(a[i-1] + a[i])*dt
                r[i+1] = v[i]*dt + 0.5*a[i]*dt^2
                """
                acc_prev = self.acc
                acc = self.force.copy()
                self.vel += 0.5*(acc_prev + acc)*dt
                self.pos += self.vel*dt + 0.5*acc*dt**2
                self.acc = acc
                self.force.set(0, 0, 0)

        def length_to(self, atom):
                return (self.pos - atom.pos).length

        def copy(self):
                return Atom(self.pos.copy())

        def __repr__(self):
                return str(self.pos)

def deep_copy(list):
        return [item.copy() for item in list]

def get_force(atom1, atom2):
        between_vec = atom1.pos - atom2.pos
        r_sqrd = between_vec.get_length_sqrd()

        direction_vec = between_vec/r_sqrd
```

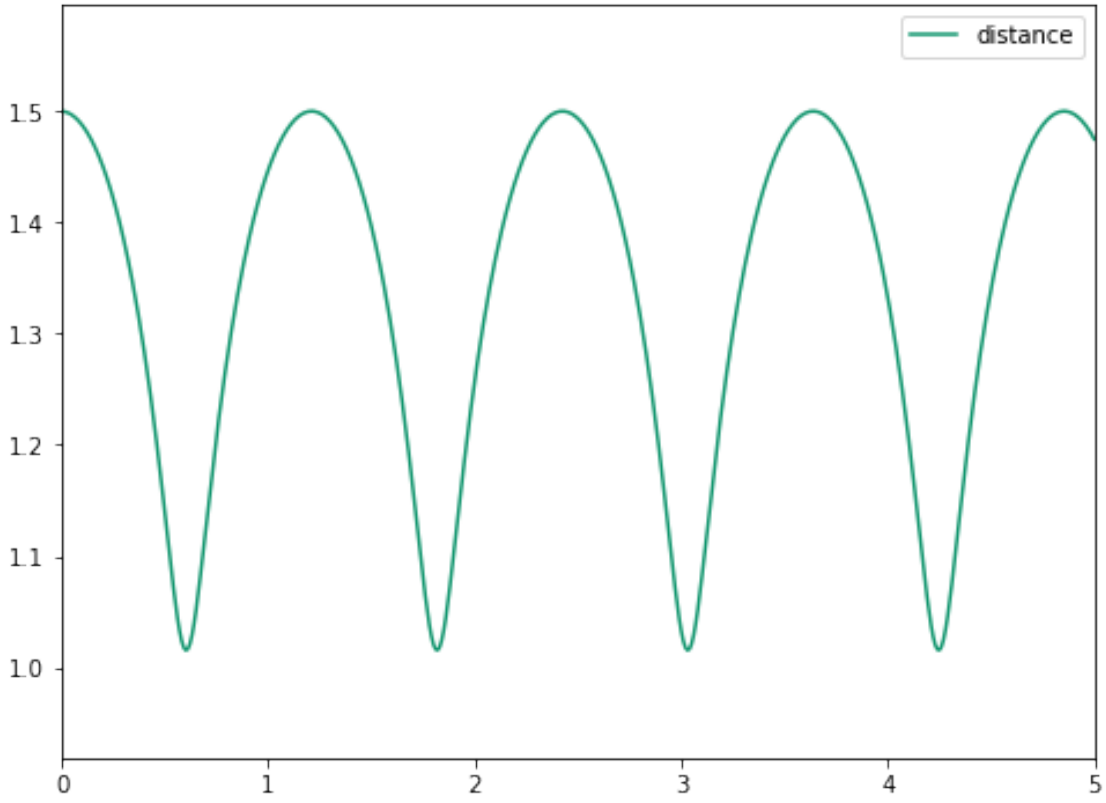
```
        force = 24*(2*r_sqrd**-6 - r_sqrd**-3)

        return direction_vec*force

def step(dt, update_func):
        global atoms

        # Add the force acting on the particles efficiently using pairs
        for atom1, atom2 in itertools.combinations(atoms, 2):
                force = get_force(atom1, atom2)
                atom1.force += force
                atom2.force -= force

        # Update the atoms position using given method
        for atom in atoms:
                atom.update(dt, update_func)

def simulate(dt, t_max, update_func):
        global atoms

        t_list = [0]
        r_list = []
        start_atoms = deep_copy(atoms)

        while t_list[-1] < t_max:
                r_list.append(atoms[0].length_to(atoms[1]))
                step(dt, update_func)
                t_list.append(t_list[-1] + dt)

        atoms = start_atoms
        return t_list[:-1], r_list
```

Programmet jeg har skrevet er objekt-orientert, jeg har brukt atom-objekter og looper gjennom en liste med disse objektene. Dette gjør at ting ser litt andreledes ut, men jeg mener ihvertfall at det gir mer mening og er mer leselig enn hvis man bare hadde positioner i et stort array.
Jeg bruker itertools.combinations og newtons 3. lov for å gjøre programmet mer effektivt. (Jeg la senere merke til at det er en del av oppgave 3a, men men).
Jeg har også endret kraft-funksjonen litt. Jeg bruker $r_{sqrd} = |\vec{r}_i\,' - \vec{r}_i\,'|^2$ istedenfor $r = |\vec{r}_i\,' - \vec{r}_i\,'|$, ettersom dette unngår å finne kvadratroten av mange tall, som er relativt treigt. Det gjør at formelen for kraft mellom to atomer blir $\left(2\left(r_{sqrd}\right)^{-6} - \left(r_{sqrd}\right)^{-3}\right)\frac{\vec{r}_i\,' - \vec{r}_i\,'}{r_{sqrd}}$.

1.2.2 b) Motion

i) og ii) Skrev et programm

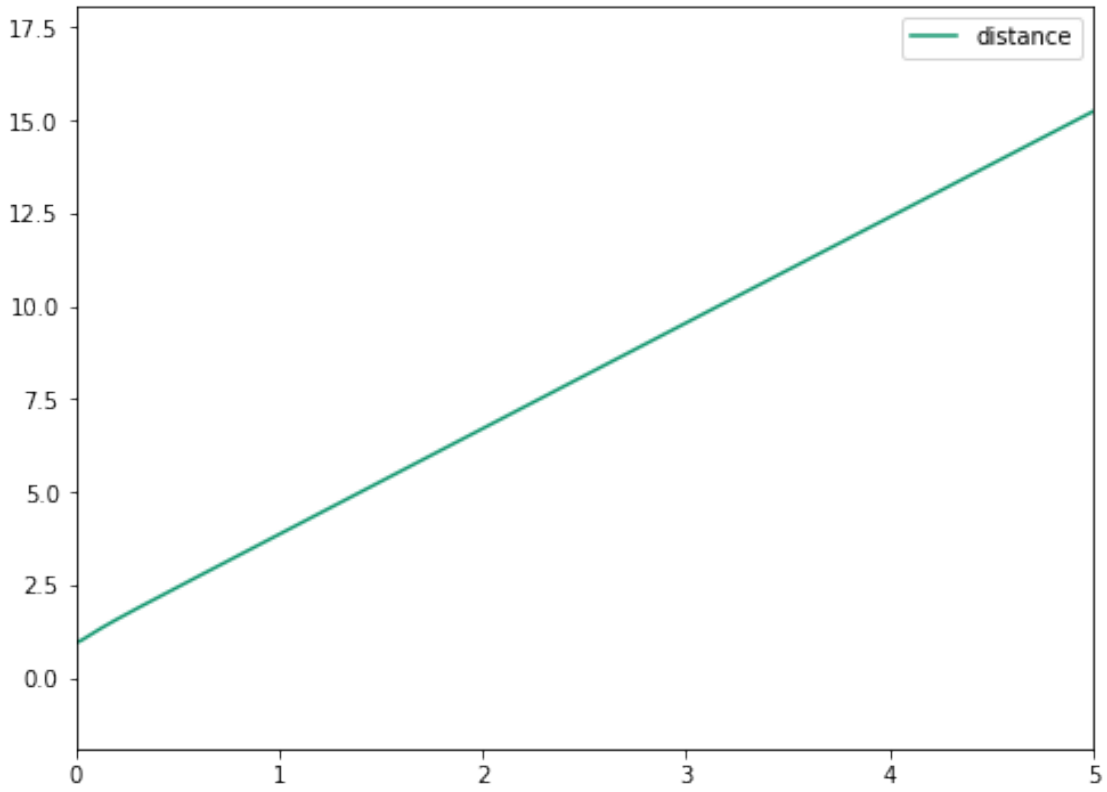
[3]:
```
atoms = [
        Atom(Vector3(0, 0, 0)),
        Atom(Vector3(1.5, 0, 0))
]

def main():
        dt = 0.01
        length = 5
        t_list, r_list = simulate(dt, length, "chromer")
        plt.figure(figsize=(8, 6))
        plt.plot(t_list, r_list, color="#11966e", label="distance")
        plt.legend()

        min_r, max_r = min(r_list), max(r_list)
        padding = (max_r-min_r)/5
        plt.axis([0, length, min_r-padding, max_r+padding])

        plt.show()
main()
```



iii) Bevegelsen er som forventet, kulene tiltrekker hverandre helt fram til avstanden nærmer seg σ, og så frastøter de hverandre. Den totale energien er bevart, så denne bevegelsen vil fortsette i all evighet.

iv) Skrev et programm:

[4]:
```
atoms = [
        Atom(Vector3(0, 0, 0)),
        Atom(Vector3(0.95, 0, 0))
]

main()
```



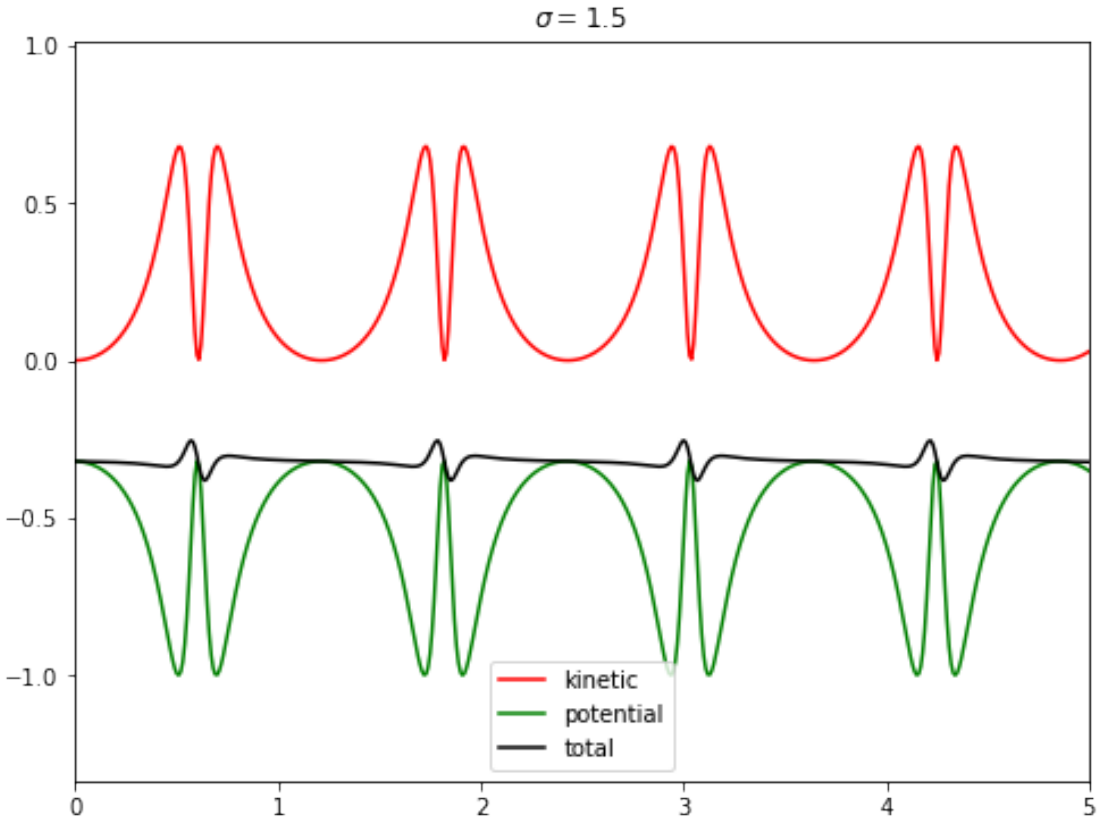
Her flyr atomene fra hverandre ettersom den potensielle energien kom over 0.

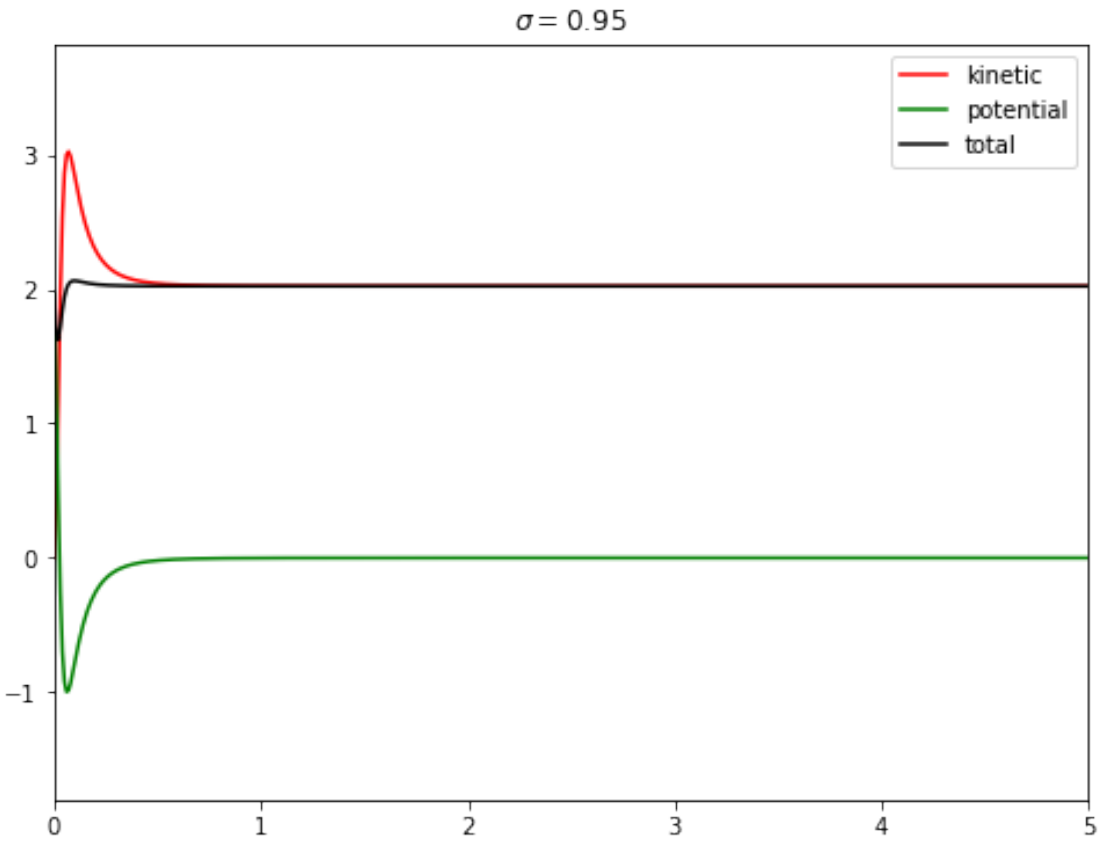
1.2.3 c) Energy

i) Den kinetiske energien er $E_k = \frac{1}{2}mv^2$, den potensielle energien er $E_p = U(r)$, og den totale energien er $E = E_k + E_p$.
Endret på programmet slik at det lagrer de ulike energiformene og plotter disse verdiene:

[5]:
```
def get_kinetic(speed):
    return 0.5*speed**2

def get_potential(r_sqrd):
    return 4*(r_sqrd**-6 - r_sqrd**-3)

def simulate(dt, t_max, update_func):
        global atoms

        t_list = [0]
        kin_list, pot_list, tot_list = ([] for i in range(3))
        start_atoms = deep_copy(atoms)

        while t_list[-1] < t_max:
                r_sqrd = (atoms[0].pos - atoms[1].pos).get_length_sqrd()

                kin_list.append(sum(get_kinetic(atom.vel.length) for atom in␣
 ↪atoms))
                pot_list.append(get_potential(r_sqrd))
                tot_list.append(kin_list[-1] + pot_list[-1])

                step(dt, update_func)
                t_list.append(t_list[-1] + dt)

        atoms = start_atoms
        return t_list[:-1], kin_list, pot_list, tot_list

atoms = [
        Atom(Vector3(0, 0, 0)),
        Atom(Vector3(1.5, 0, 0))
]

def main(title):
        dt = 0.01
        length = 5
        t_list, kin_list, pot_list, tot_list = simulate(dt, length, "chromer")
        data_list = [
                ("kinetic", "r", kin_list),
                ("potential", "g", pot_list),
                ("total", "k", tot_list),
        ]
        plt.figure(figsize=(8, 6))
        for name, color, values in data_list:
                plt.plot(t_list, values, color=color, label=name)
        plt.legend()
        min_value = min(pot_list)
        max_value = max(kin_list)
```

```
        buffer = (max_value - min_value)/5
        plt.axis([0, length, min_value-buffer, max_value+buffer])
        plt.title(f"$\\sigma=${title}")
        plt.show()
main("1.5")

atoms = [
        Atom(Vector3(0, 0, 0)),
        Atom(Vector3(0.95, 0, 0))
]
main("0.95")
```





ii) Den totale energien bør bevares, energien til atomene går ikke over til noen andre former (det er ikke noe luftmotstand eller liknende), og ettersom energi ikke kan forsvinne må den bevares.

1.2.4 iii)

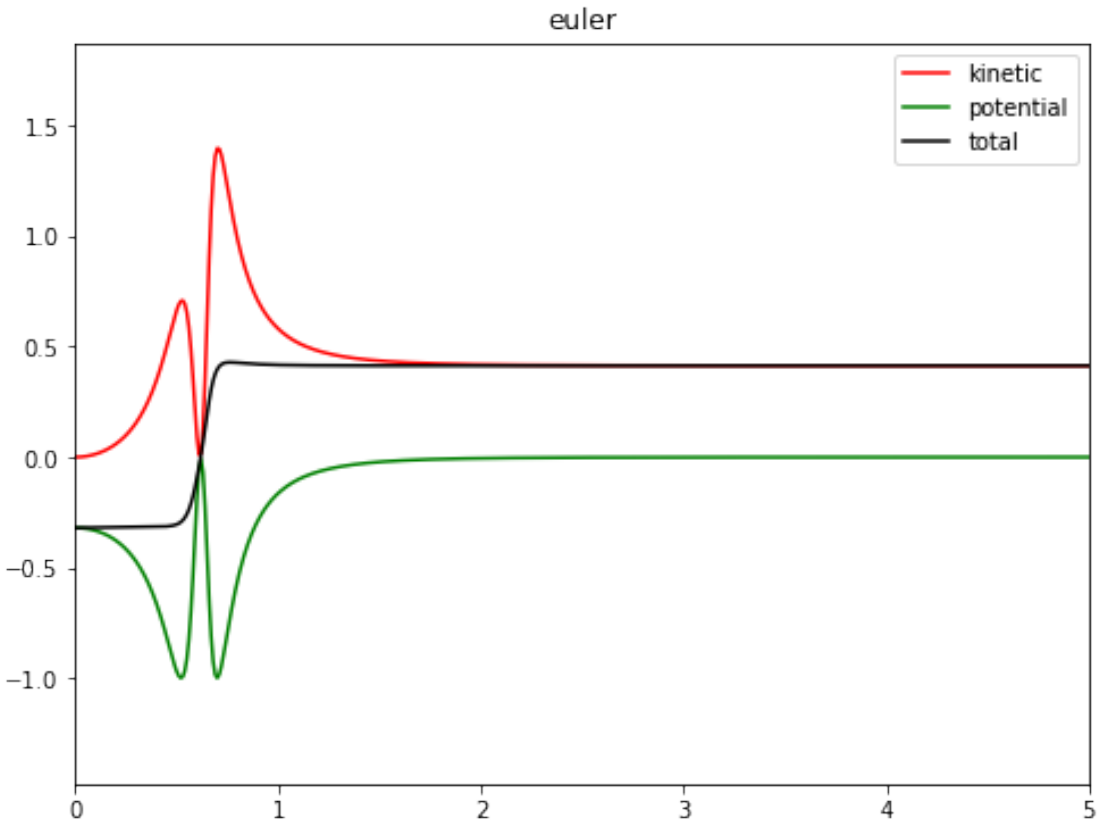
Programmet mitt gjør ikke dette med en dt på 0.01, men med lavere verdier er den totale energien konstant. Nyaktig hva som går galt med for høy dt vet jeg ikke, men det er ikke uvanlig at høy dt fører til små unyaktigheter.

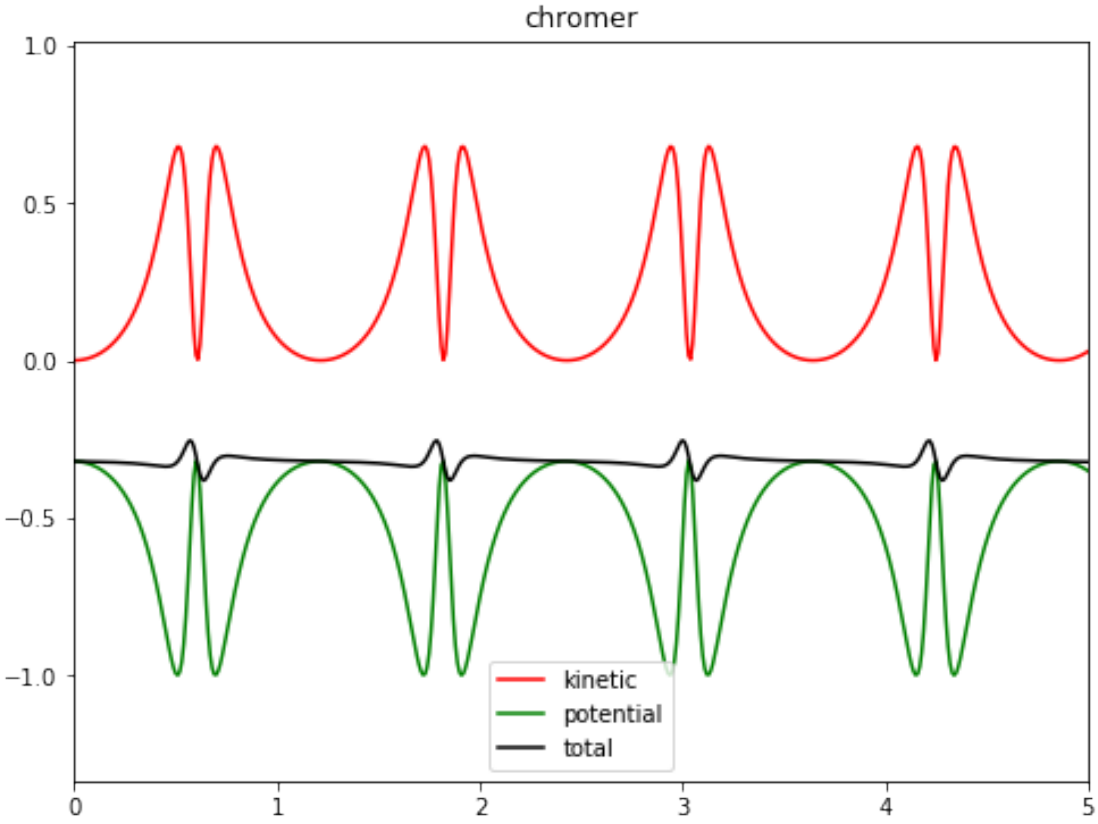
1.2.5 iv)

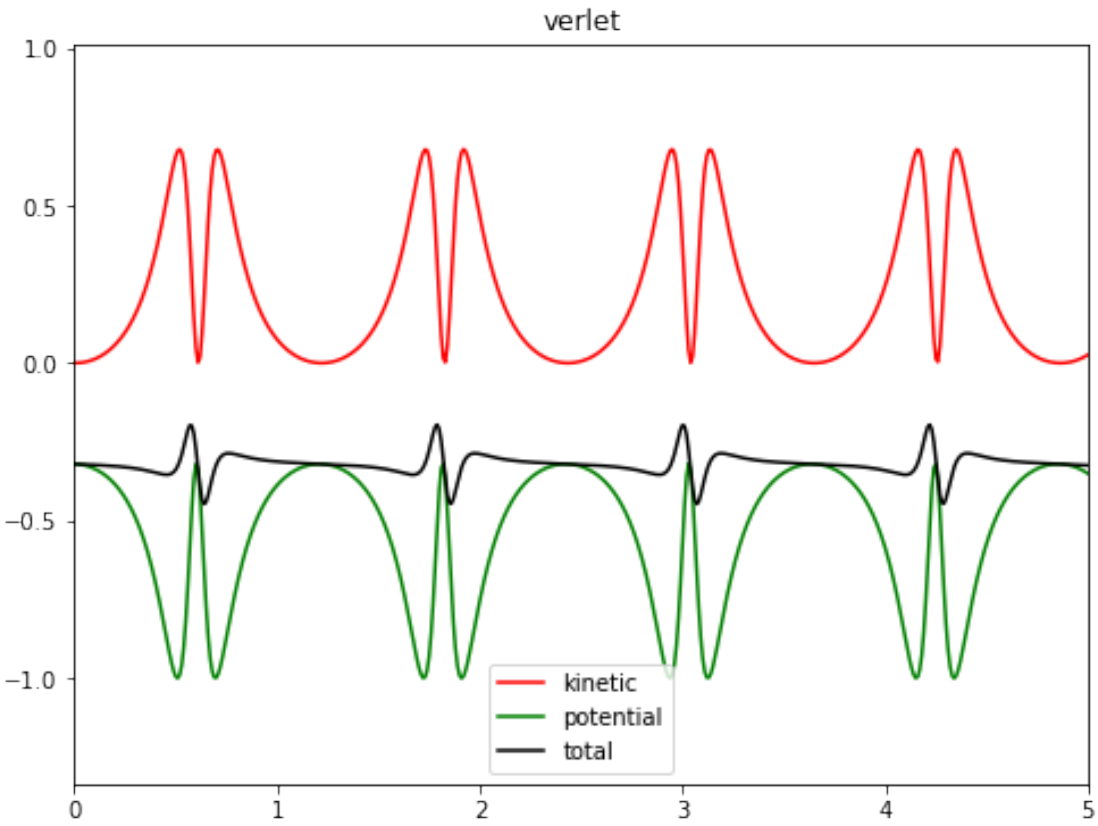
Skrev et programm som plotter med Euler, Euler Chromer og Velocity Verlet:

[6]:
```
atoms = [
        Atom(Vector3(0, 0, 0)),
        Atom(Vector3(1.5, 0, 0))
]
```

```
def main(method):
        dt = 0.01
        length = 5
        t_list, kin_list, pot_list, tot_list = simulate(dt, length, method)
        data_list = [
                ("kinetic", "r", kin_list),
                ("potential", "g", pot_list),
                ("total", "k", tot_list),
        ]
        plt.figure(figsize=(8, 6))
        for name, color, values in data_list:
                plt.plot(t_list, values, color=color, label=name)
        plt.legend()
        min_value = min(pot_list)
        max_value = max(kin_list)
        buffer = (max_value - min_value)/5
        plt.axis([0, length, min_value-buffer, max_value+buffer])
        plt.title(method)
        plt.show()
main("euler")
main("chromer")
main("verlet")
```




chromer
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
kinetic
potential
total



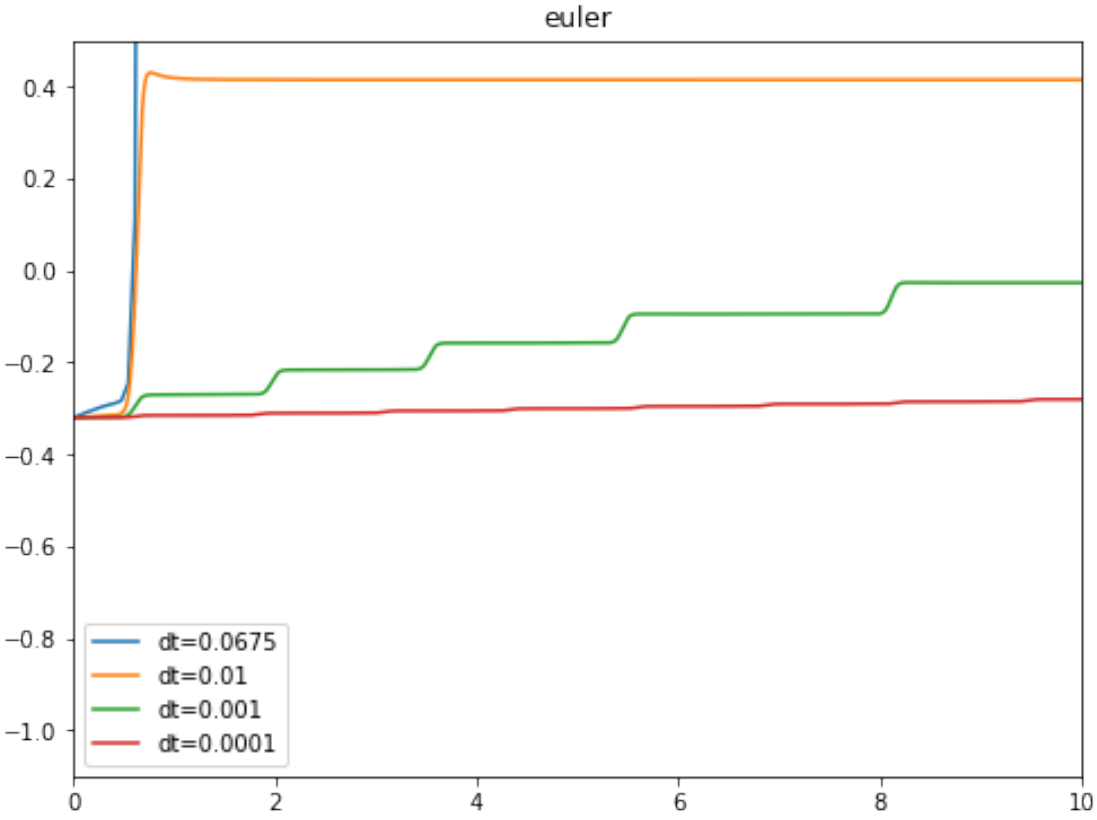
Eulers metode er helt på bærtur, Euler Chromer er best, og velocity verlet er ganske grei.

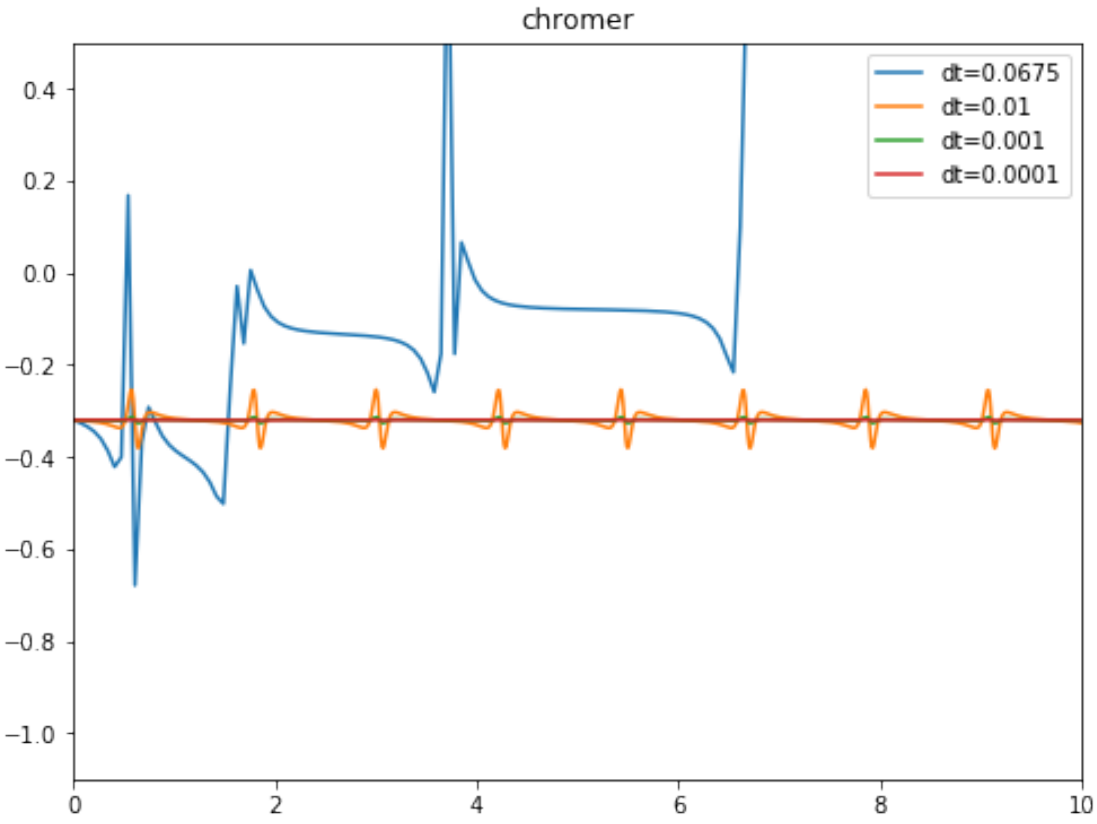
1.2.6 v)

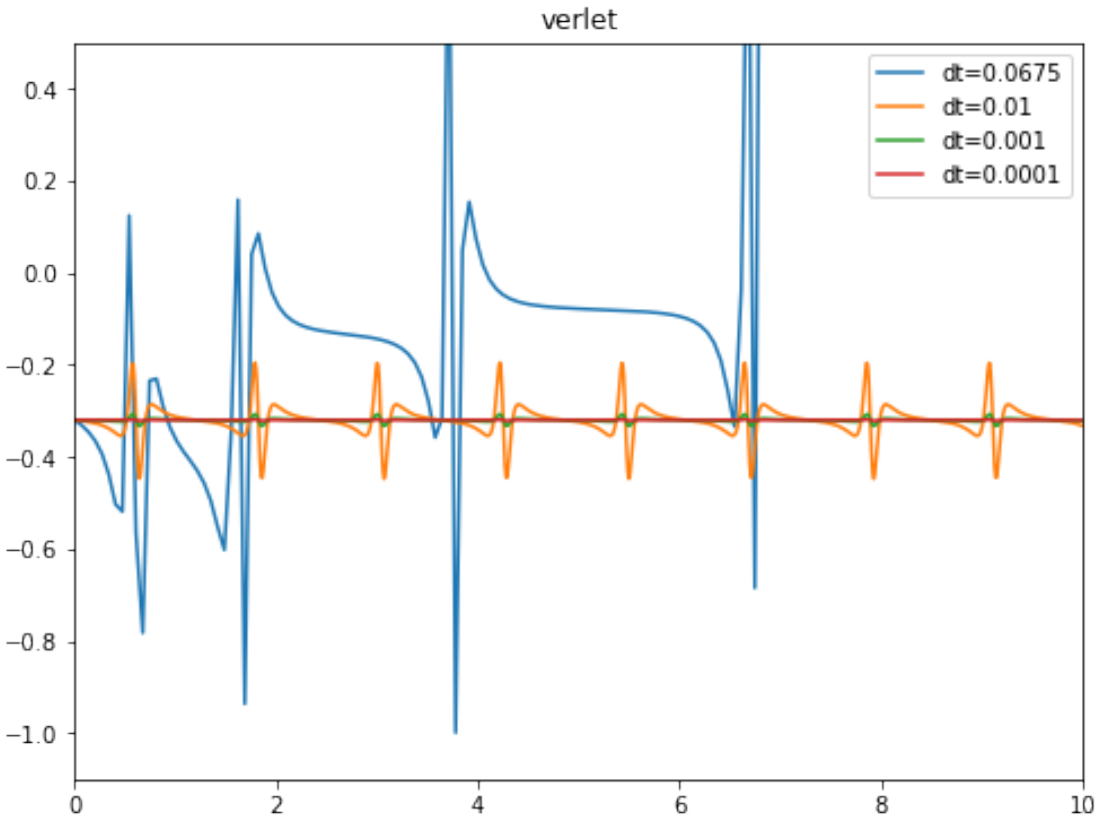
Plotter for forskjellige dt verdier:

[7]:
```
def main(method):
        length = 10

        tot_lists = []
        plt.figure(figsize=(8, 6))
        for i, dt in enumerate([0.0675, 0.01, 0.001, 0.0001]):
                t_list, kin_list, pot_list, tot_list = simulate(dt, length,␣
↪method)
                tot_lists.append(tot_list)
                plt.plot(t_list, tot_list, label=f"dt={dt}")
        plt.legend()
        plt.title(method)
        plt.axis([0, length, -1.1, 0.5])
        plt.show()
```

```
main("euler")
main("chromer")
main("verlet")
```




chromer
dt=0.0675
dt=0.01
dt=0.001
dt=0.0001
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
2
4
6
8
10



Euler er det ikke mye håp for, selv med dt=0.0001 er den totale energien ikke konstant. Både Euler-Chromer og Velocity-Verlet går det bedre med, selv med dt=0.01 er den totale energien gjennomsnittlig konstant, den varierer, men variasjonen er periodisk og like stor i begge retninger. Ved dt=0.001 er denne variasjonen veldig lav, og ved dt=0.0001 er den totale energien så godt som konstant. Det vil si at det ikke egentlig finnes et største tidssteg hvor Euler konserverer energi, men med dt=0.0001 er feilen liten nok til at det ikke er så farlig, mens det største tidssteget som konserverer energi er 0.01 for både Euler-Chromer og Velocity-Verlet.

1.2.7 vi)

Det virker som om Euler-Chromer er den beste av de tre numeriske metodene, både i nyaktighet og effektivitet. Euler er lett å regne ut for hvert tidssteg, men har lav nyaktighet, mens Euler-Chromer er like lett å regne ut, du bare bytter om rekkefølgen på fart og positon, mens nyaktigheten er mye høyere. Velocity-Verlet virker ca. like nyaktig som Euler-Chromer, men er mer krevende å regne ut, man må ta vare på forrige akselerasjon og gjøre noen ekstra utregninger. Euler-Chromer har altså bare fordeler og ingen ulemper.

1.2.8 d)

i) La til filskrivingslogikk, det vil så la til en ekstra funksjon i atom-objektet og litt i step-funksjonen og i simulate-funksjonen:

[8]:
```
class Atom:
        def __init__(self, pos, vel=None):
                self.force = Vector3()
                self.acc = Vector3()
                if vel is None:
                        self.vel = Vector3()
                self.pos = pos

        def update(self, dt, func="chromer"):
                if func == "chromer":
                        self.update_chromer(dt)
                elif func == "euler":
                        self.update_euler(dt)
                elif func == "verlet":
                        self.update_verlet(dt)
                else:
                        print(">:[")

        def update_euler(self, dt):
                """
                r[i+1] = r[i] + v[i]*dt
                v[i+1] = v[i] + a[i]*dt
                """
                acc = self.force
                self.pos += self.vel*dt
                self.vel += acc*dt
                self.force.set(0, 0, 0)

        def update_chromer(self, dt):
                """
                v[i+1] = v[i] + a[i]*dt
                r[i+1] = r[i] + v[i+1]*dt
                """
                acc = self.force
                self.vel += acc*dt
                self.pos += self.vel*dt
                self.force.set(0, 0, 0)

        def update_verlet(self, dt):
                """
                v[i] = v[i-1] + 0.5*(a[i-1] + a[i])*dt
                r[i+1] = v[i]*dt + 0.5*a[i]*dt^2
                """
                acc_prev = self.acc
```

```
                acc = self.force
                self.vel += 0.5*(acc_prev + acc)*dt
                self.pos += self.vel*dt + 0.5*acc*dt**2
                self.acc = acc
                self.force = Vector3()

        def save_state(self, file):
                file.write(f"Ar {self.pos.x:f} {self.pos.y:f} {self.pos.z:f}\n")

        def length_to(self, atom):
                return (self.pos - atom.pos).length

        def copy(self):
                return Atom(self.pos.copy())

        def __repr__(self):
                return str(self.pos)
atoms = [
        Atom(Vector3(0, 0, 0)),
        Atom(Vector3(1.5, 0, 0))
]


def step(dt, update_func, datafile):
        global atoms

        # Add the force acting on the particles efficiently using pairs
        for atom1, atom2 in itertools.combinations(atoms, 2):
                force = get_force(atom1, atom2)
                atom1.force += force
                atom2.force -= force

        datafile.write(f"{len(atoms)}\ntype x y z\n")
        # Update the atoms position using given method
        for atom in atoms:
                atom.update(dt, update_func)
                atom.save_state(datafile)


def simulate(dt, t_max, update_func, filename="data.xyz"):
        global atoms

        t_list = [0]
        r_list = []
        start_atoms = deep_copy(atoms)

        datafile = open("data/"+filename, "w")
```

```
        start_time = time.time()
        while t_list[-1] < t_max:
                r_list.append(atoms[0].length_to(atoms[1]))
                step(dt, update_func, datafile)
                t_list.append(t_list[-1] + dt)

        datafile.close()

        atoms = start_atoms
        return t_list[:-1], r_list
```

ii) Nice

1.3 3 Large systems

1.3.1 a) Implementation

i) Løsningen fungerer allerede for N atomer og tar inn startposition og startfart

ii) Løsningen bruker allerede Newtons tredje lov for å halvere kraftkalkulasjoner

1.3.2 iii)

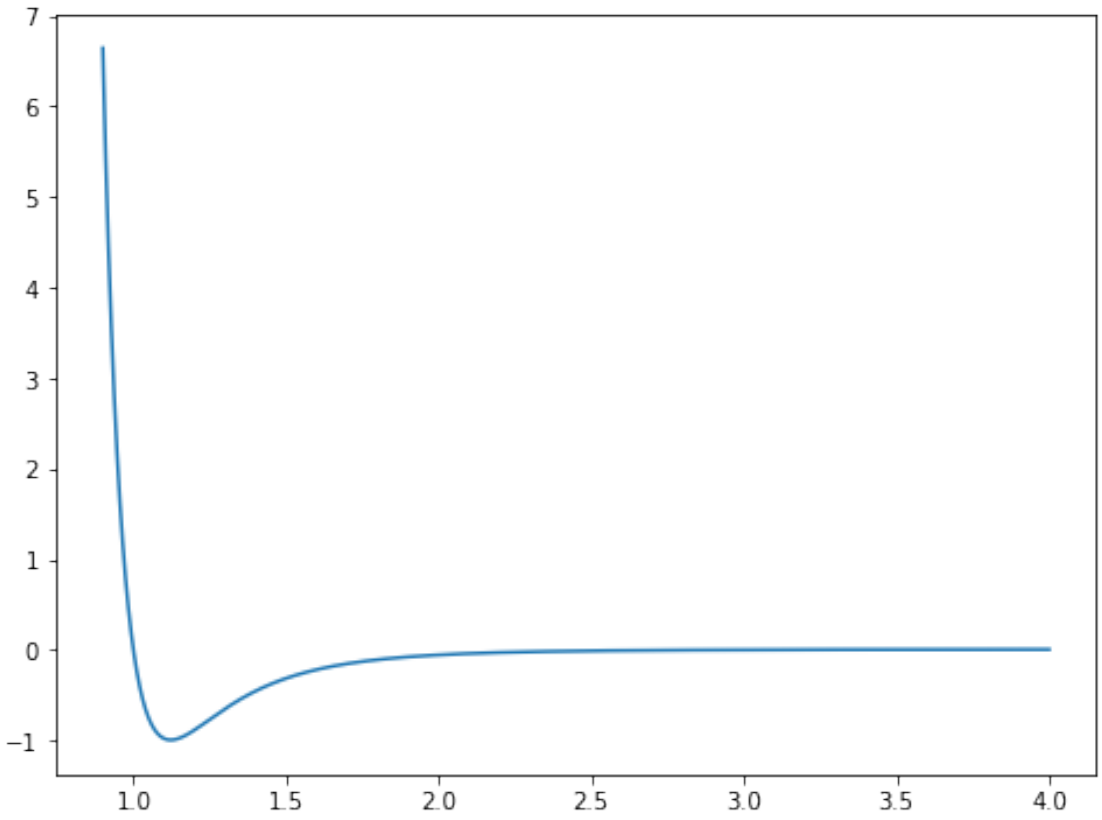
Endret på kraftkalkulasjonen slik at atomer med avstand større enn 3 ikke pårvirker hverandre. Jeg bruker fortsatt r_{sqrd}, så jeg tester heller om r_{sqrd} er større enn 9:

[9]:
```
def get_force(between_vec, r_sqrd):
        direction_vec = between_vec/r_sqrd
        force = 24*(2*r_sqrd**-6 - r_sqrd**-3)

        return direction_vec*force

def step(dt, update_func, datafile):
        global atoms

        # Add the force acting on the particles efficiently using pairs
        for atom1, atom2 in itertools.combinations(atoms, 2):
                between_vec = atom1.pos - atom2.pos
                r_sqrd = between_vec.get_length_sqrd()

                if r_sqrd < 9:
                        force = get_force(between_vec, r_sqrd)
                        atom1.force += force
```

```
                        atom2.force -= force

        datafile.write(f"{len(atoms)}\ntype x y z\n")
        # Update the atoms position using given method
        for atom in atoms:
                atom.update(dt, update_func)
                atom.save_state(datafile)
```

iV) Dette alene ødelegger energibevarelsen, så jeg må legge til en konstant til U. $U(3) = -\frac{2912}{531441}$, og det er bare å legge til -U(3) $= \frac{2912}{531441}$ til U. Plotter den nye potensialfunksjonen:

[10]:
```
def U(r):
    return 4*(r**-12 - r**-6) + 2912/531441

r_ray = np.linspace(0.9, 4, 1000)
U_ray = U(r_ray)

plt.figure(figsize=(8, 6))
plt.plot(r_ray, U_ray)
plt.show()

print(f"U(3) = {U(3)}")
```



```
U(3) = 0.0
```

v) Ettersom kraften er den deriverte av potensialfunksjonen bør ikke dette gjøre noe forskjell

1.3.3 b) Verification

i) Strengt tatt unødvendig ettersom jeg ikke har gjordt noen endringer på implementasjonen.

ii) Endret på atom-arrayet og skjørte simuleringen:

[11]:
```
atoms = [
        Atom(Vector3(1, 0, 0)),
        Atom(Vector3(0, 1, 0)),
        Atom(Vector3(-1, 0, 0)),
        Atom(Vector3(0, -1, 0))
]

def main(filename):
        dt = 0.01
        length = 5
        simulate(dt, length, "verlet", filename)
main("data3b2.xyz")
```

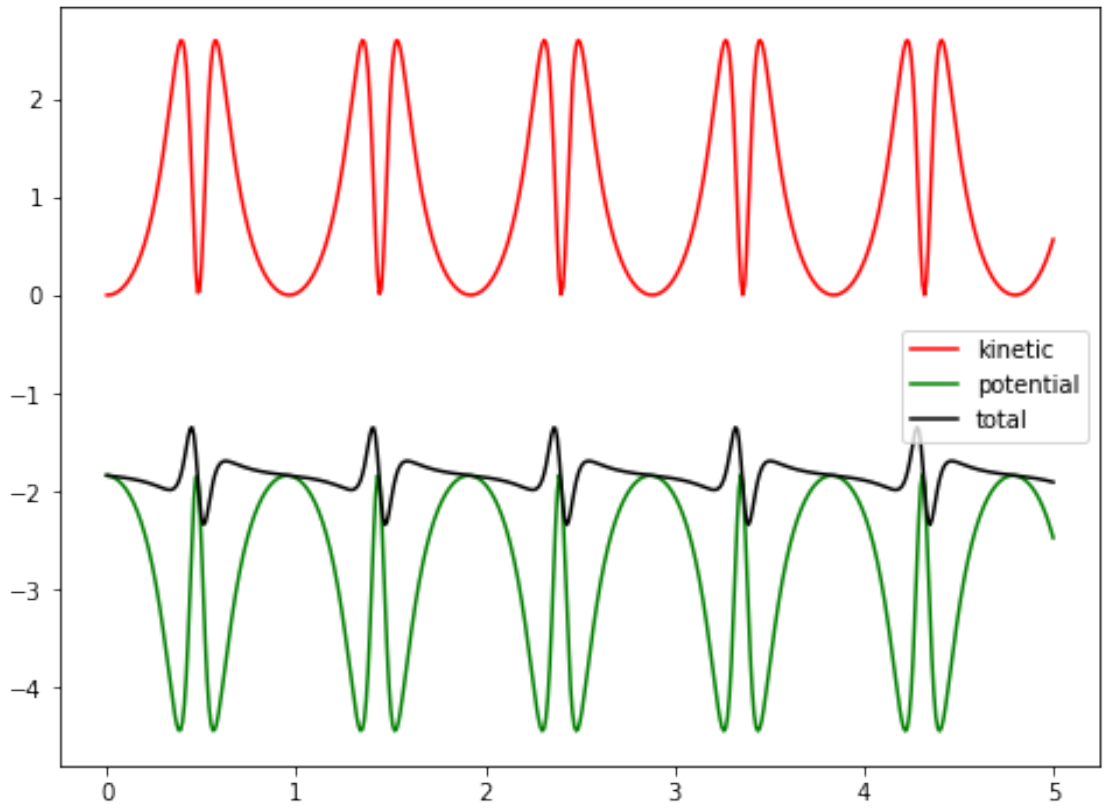
iii) Hvordan skal jeg egentlig vise at jeg har visualisert i Ovito?

Ettersom atomene er plassert i en firkant blir draget fra de to nærmeste atomene lagt sammen til et drag mot atomet som er lengst unna, så alle atomene beveger seg rett frem diagonalt mot atomet på den andre siden. Det er altså fortsatt en syklisk bevegelse.

iv) Skrev en mer generisk energikalkulasjonskode og plottet energien i systemet:

[12]:
```
def U(r_sqrd):
        return 4*(r_sqrd**-6 - r_sqrd**-3) + 2912/531441

def get_energy(atoms):
        # Calculate potential energy:
        potential_energy = 0
        for atom1, atom2 in itertools.combinations(atoms, 2):
                between_vec = atom1.pos - atom2.pos
                r_sqrd = between_vec.get_length_sqrd()
                potential_energy += U(r_sqrd)
```

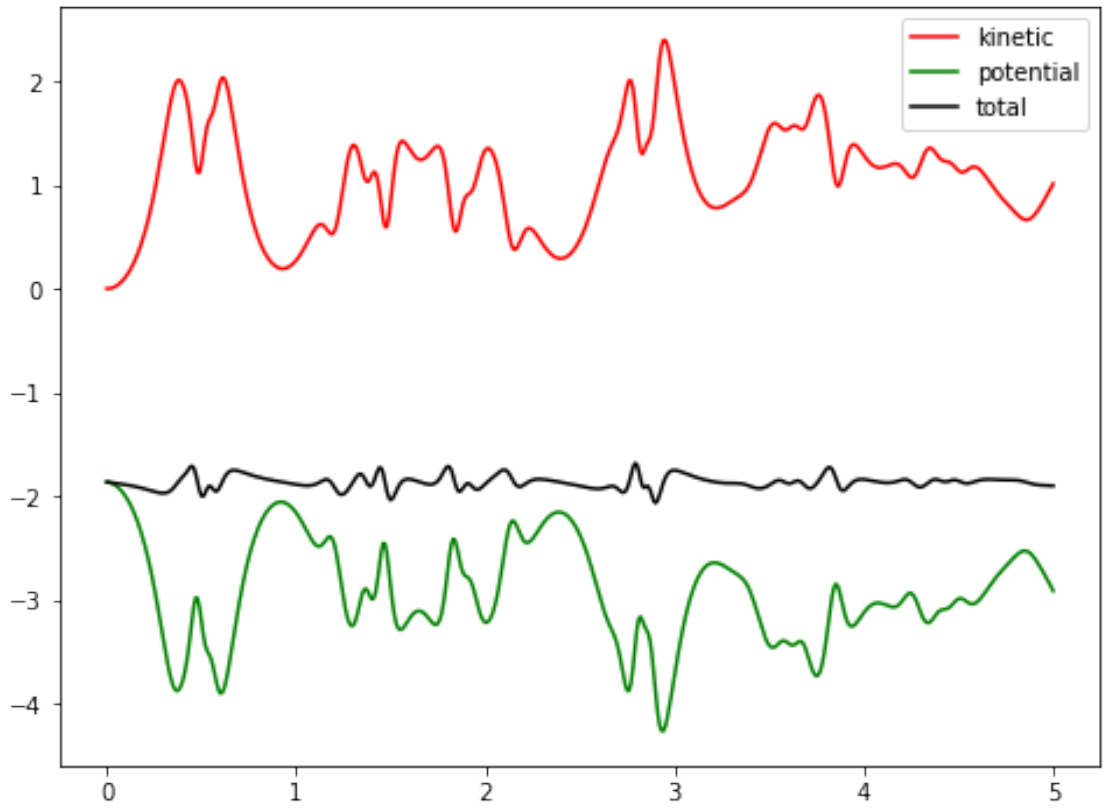
```
        # Calculate kinetic energy:
        kinetic_energy = reduce(lambda accumelator, atom: accumelator + 0.
 ↪5*atom.vel.get_length_sqrd(), atoms, 0)

        return potential_energy, kinetic_energy, potential_energy+kinetic_energy

def simulate(dt, t_max, update_func, filename="data.xyz"):
        global atoms

        t_list = [0]
        pot_list = []
        kin_list = []
        tot_list = []
        start_atoms = deep_copy(atoms)

        datafile = open("data/"+filename, "w")

        while t_list[-1] < t_max:
                pot, kin, tot = get_energy(atoms)
                pot_list.append(pot)
                kin_list.append(kin)
                tot_list.append(tot)
                step(dt, update_func, datafile)
                t_list.append(t_list[-1] + dt)

        datafile.close()

        atoms = start_atoms
        return t_list[:-1], pot_list, kin_list, tot_list

def main(filename):
        dt = 0.01
        length = 5
        t_list, pot_list, kin_list, tot_list = simulate(dt, length, "verlet")
        data_list = [
                ("kinetic", "r", kin_list),
                ("potential", "g", pot_list),
                ("total", "k", tot_list),
        ]
        plt.figure(figsize=(8, 6))
        for name, color, values in data_list:
                plt.plot(t_list, values, color=color, label=name)
        plt.legend()
        plt.show()

main("data3b2.xyz")
```



Den totale energien oppfører seg likt som forventet fra plottene i 2c, den totale energien er ikke konstant, men den er syklisk og feilen øker ikke over tid, så det er ikke et problem.

v) Skjørte simuleringen med litt andre initialverdier:

[13]:
```
atoms = [
        Atom(Vector3(1, 0.1, 0)),
        Atom(Vector3(0, 1, 0)),
        Atom(Vector3(-1, 0, 0)),
        Atom(Vector3(0, -1, 0))
]

main("data3b5.xyz")
```



Selv om atomene startet med nesten samme positon som i ii førte den lille forstyrrelsen til at bevegelsen ble kaotisk veldig fort.
Det er likevel slik at den totale energein er bevart, men energifiguren er mye mer rotete.

[ ]:

1.3.4 c) Initialisation

i) Skrev et programm som returnerer atompositioner i en krystallstruktur:

[14]:
```
def box_positions(n, L):
        d = L/n
        positions = []
        for i in range(0, n):
                for j in range(0, n):
                        for k in range(0, n):
                                positions.append(Vector3(   i,       j,      k ␣
↪  )*d)
                                positions.append(Vector3(   i,    0.5 + j, 0.5␣
↪+ k)*d)
```

```
                                positions.append(Vector3(0.5 + i,    j,    0.5␣
↪+ k)*d)
                                positions.append(Vector3(0.5 + i, 0.5 + j,   k ␣
↪  )*d)
        return positions
```

ii) Skrev et program som lagrer positionsverdiene i en xyz-fil:

[15]:
```
positions = box_positions(3, 20)

with open("data/data3c2.xyz", "w") as outfile:
        outfile.write(f"{len(positions)}\ntype x y z\n")
        for position in positions:
                outfile.write(f"Ar {position.x:f} {position.y:f} {position.z:
↪f}\n")
```

Ser riktig ut det ass

iii) Hvordan regner man ut tetthet???

1.3.5 d)

i) !