Symulacje dynamiki molekularnej - ruchy planet.

Rafał Kornel

Wstęp

Problem - rozwiązać numerycznie problem dwóch ciał.

$$\mathbf{F}_j = m_j \frac{d^2\mathbf{r}_j}{dt^2} = \sum_{i=1, i\neq j}^{N} \mathbf{F}_{ij} \tag{1}$$

Do rozwiązania problemu zostały użyte trzy metody numeryczne:

Algorytm Eulera

Wynika z rozwinięcia w szereg Taylora:

$$\mathbf{r}(t+\delta t) \approx \mathbf{r}(t) + \dot{\mathbf{r}}(t)\delta t + \frac{1}{2!}\ddot{\mathbf{r}}(t)\delta t^2 \tag{2}$$

Stąd wynikają wzory na położenie oraz pęd:

$$r(t+\delta t) = r(t) + p(t)\delta t + \frac{1}{2}F(t)\delta t^2$$

$$p(t+\delta t) = p(t) + F(t)\delta t$$

Kod poniżej przedstawia klasę, która została użyta do przechowywania danych.

```
class Planet:
    def __init__(self, pos, mom, m, static = False):
        self.pos = pos                  # position
        self.mom = mom                  # momentum
        self.m = m                      # mass
        self.pos_history = []
        self.vel_history = []
        self.pot = []                   # potential energy
        self.kin = []                   # kinetic energy

        self.is_static = static
        self.r = 0.3 if static else 0.2
        self.fig = pl.Circle( self.pos, self.r, color=pl.cm.summer(self.m))
```

Kod poniżej przedstawia implementacje algorytmu Euler'a.

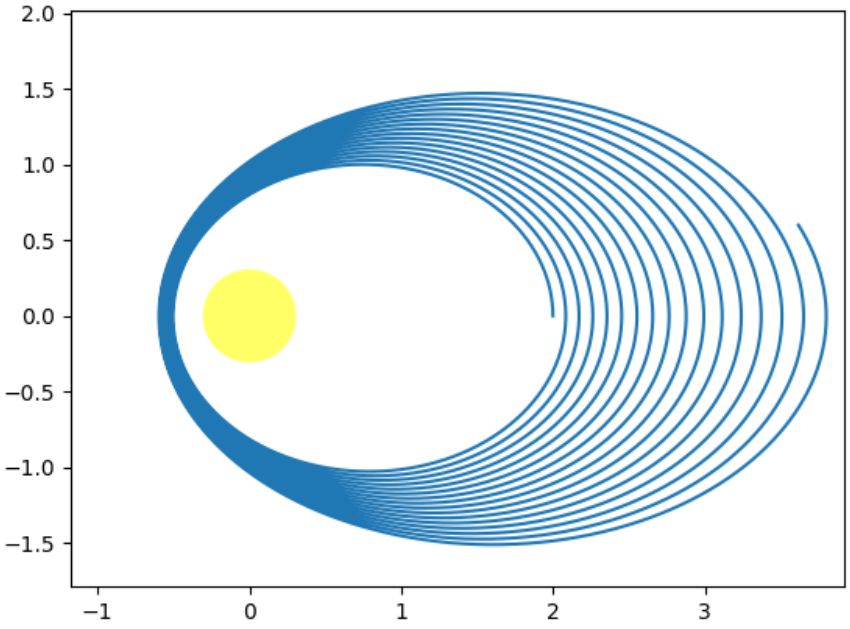
```
def evolve_euler(self):
        for p in self.planets:
            f = self.calculate_force(p)

            p.pos_history.append(p.pos)

            kin, pot = self.calculate_energy(p)
            p.kin.append(kin)
            p.pot.append(pot)

            p.pos = p.pos + p.mom * self.dt/p.m + 1/(2*p.m) * f * (self.dt**2)
            p.mom = p.mom + f * self.dt
```

Dla $N = 100000$ iteracji, orbita planety o warunkach początkowych

$$\mathbf{r} = (2, 0)$$

$$\mathbf{p} = (0, 0.1)$$

wygląda następująco:



Rys. 1: Orbita planety w metodzie Eulera.

A tak wygląda wykres energii:
Energia całkowita w tej metodzie nie jest zachowana, co widać na przybliżeniu:

Algorytm Verleta

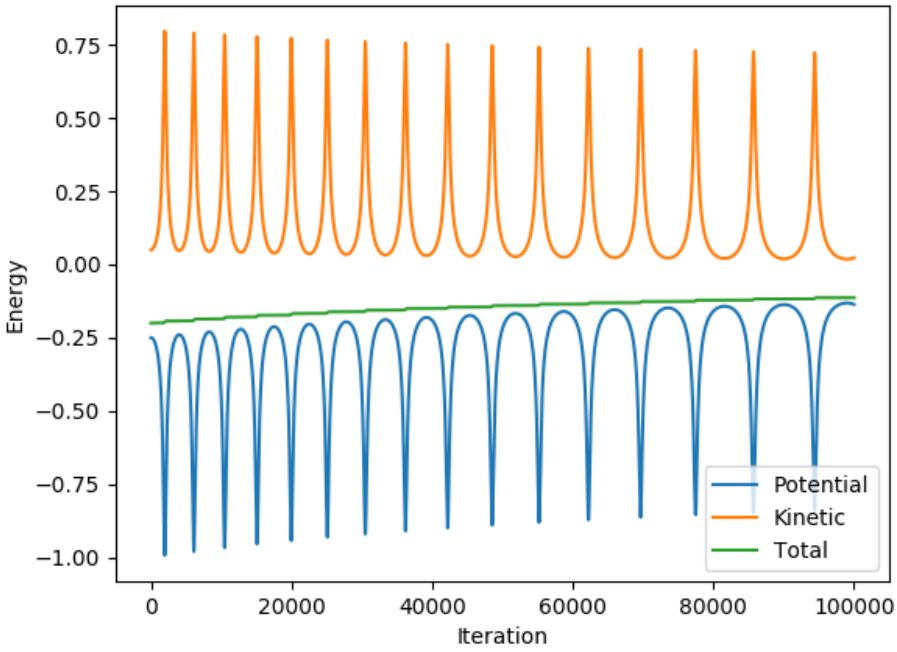
Wzór na położenie w n-tym kroku ma następującą postać:

$$r_{n+1} = 2r_n - r_{n-1} + (\frac{F_n}{m})\delta t^2$$

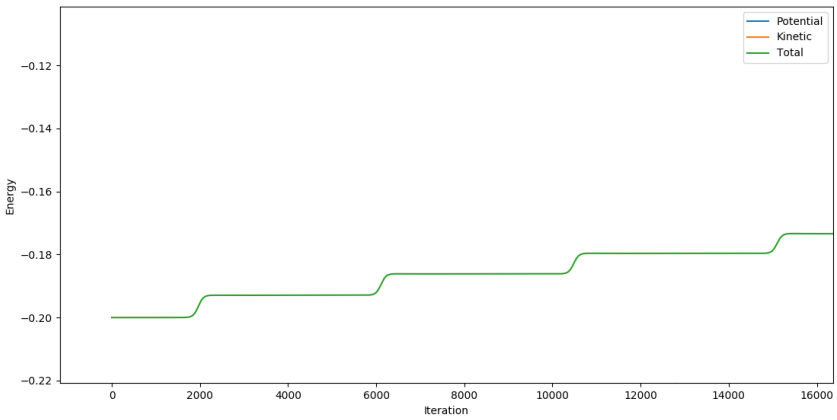
Jak widać, do wyznaczenia n+1-ego kroku potrzebujemy tylko n-tego i n-1-ego położenia. Prędkość nie jest potrzebna do algorytmu Verleta, lecz gdybyśmy chcieli ją policzyć, to wyraża się wzorem:

$$v_n = \frac{r_{n+1} - r_{n-1}}{2\delta t}$$

Tak wygląda implementacja algorytmu Verleta:
Dla takich samych jak wyżej warunków początkowych orbita przy algorytmie Verleta wygląda tak:
Tak prezentuje się wykres energii w tymże algorytmie:
Przybliżając wykres energii możemy zauważyć, że energia całkowita , mimo że nie jest stała, to nie rośnie wraz z postępem symulacji.

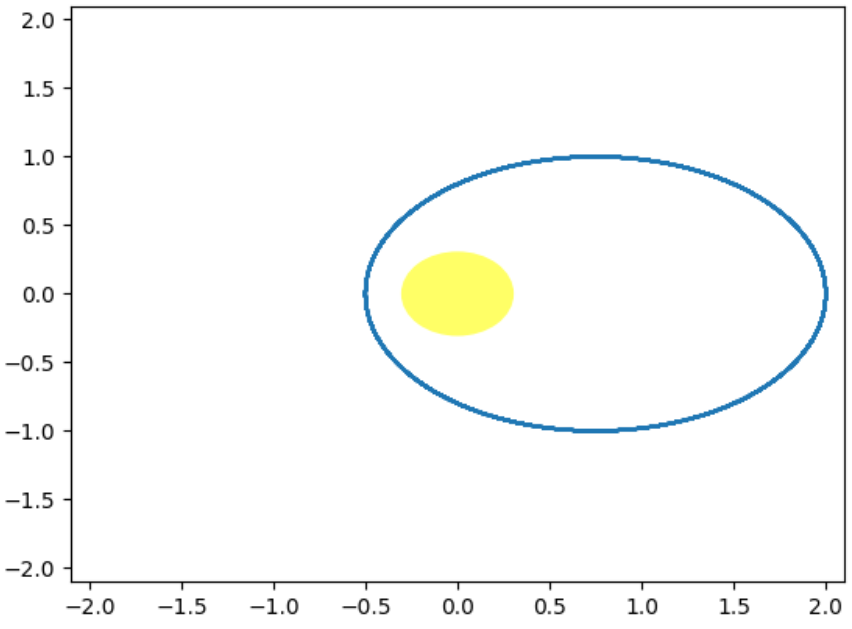


Rys. 2: Bilans energii w symulacji metoda Eulera.

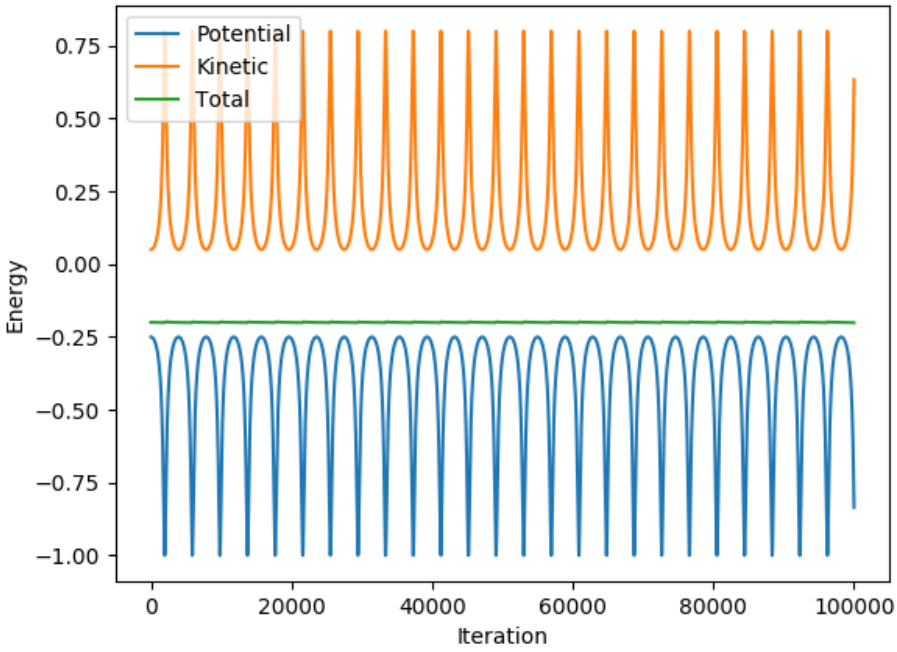


Rys. 3: Przybliżenie wykresu energii - widać wzrosty energii całkowitej.

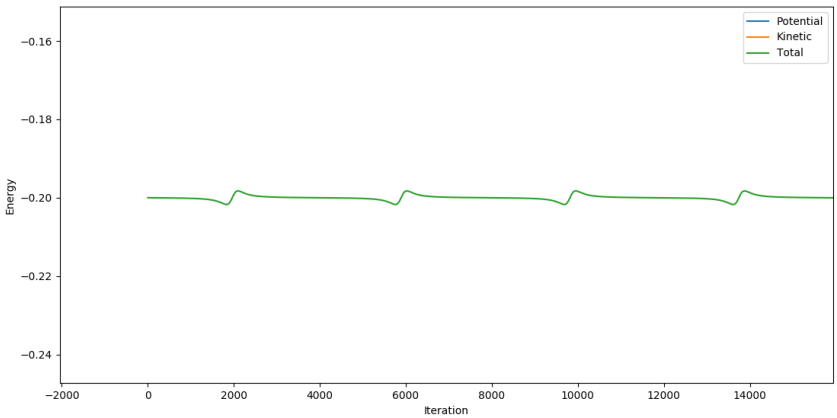
```
def evolve_verlet(self):
        for p in self.planets:
            f = self.calculate_force(p)

            if len(p.pos_history) == 0:
                pos_prev = p.pos - p.mom * self.dt/p.m
                p.pos_history.append(pos_prev)

            p.pos_history.append(p.pos)

            kin, pot = self.calculate_energy(p)
            p.kin.append(kin)
            p.pot.append(pot)

            p.pos = 2*p.pos_history[-1] - p.pos_history[-2] + f*(self.dt)**2/p.m
            p.mom = p.m/(2*self.dt) * (p.pos - p.pos_history[-2])
```



Rys. 4: Orbita planety używajac algorytmu Verleta.



Rys. 5: Bilans energii w symulacji wykorzystując algorytm Verleta.



Rys. 6: Przybliżenie wykresu energii w metodzie wykorzystującej algorytm Verleta

Algorytm skokowy (leapfrog)

W algorytmie skokowym liczymy prędkość w kroku $n + 1/2$, aby uzyskać położenie. Wzory potrzebne do przeprowadzenia symulacji:

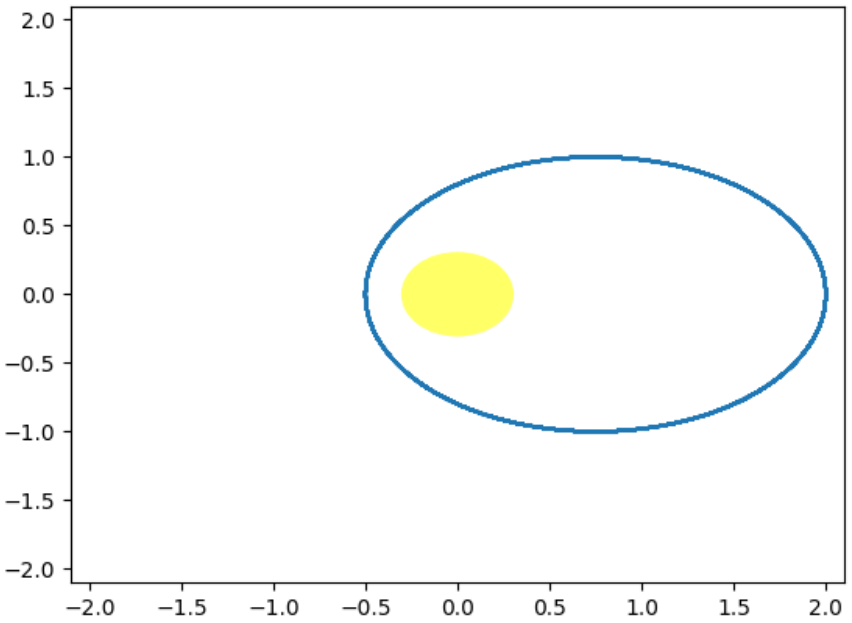
$$v_{n+1/2} = v_{n-1/2} + (\frac{F}{m})\delta t$$

$$r_{n+1} = r_n + v_{n+1/2}\delta t$$

Kod przedstawiający implementacje algorytmu skokowego:

```
def evolve_leapfrog(self):
        for p in self.planets:
            f = self.calculate_force(p)

            if len(p.vel_history) == 0:
                p.vel_history.append(p.mom/p.m - f*self.dt/(2*p.m))

            p.pos_history.append(p.pos)
            p.vel_history.append(p.vel_history[-1] + f*self.dt/p.m)

            kin, pot = self.calculate_energy(p)
            p.kin.append(kin)
            p.pot.append(pot)

            p.mom = p.m*(p.vel_history[-1] + p.vel_history[-2])/2
            p.pos = p.pos + p.vel_history[-1]*self.dt
```

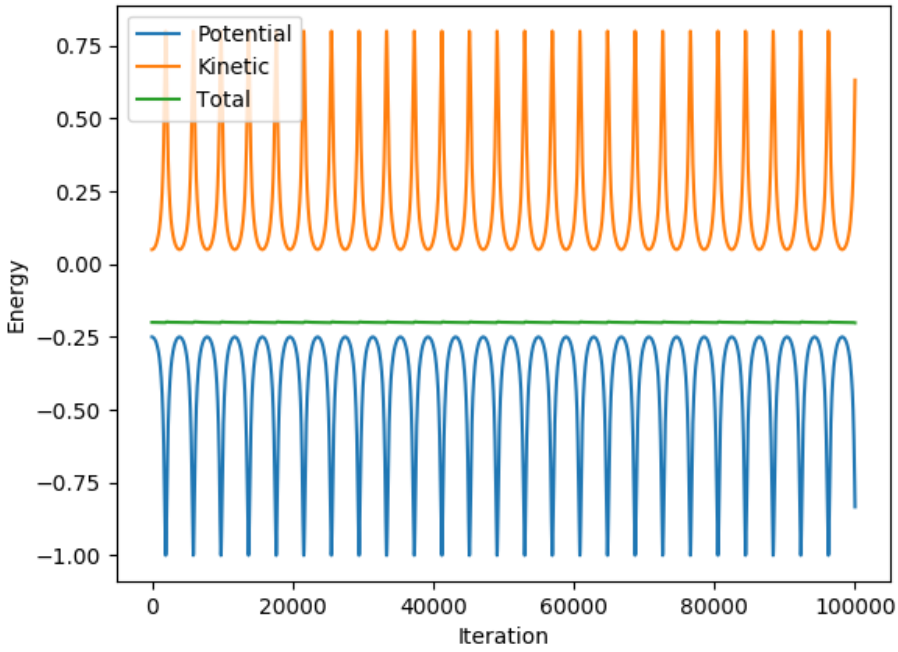
Tak prezentuje się orbita planety w symulacji wykorzystującej algorytm skokowy:



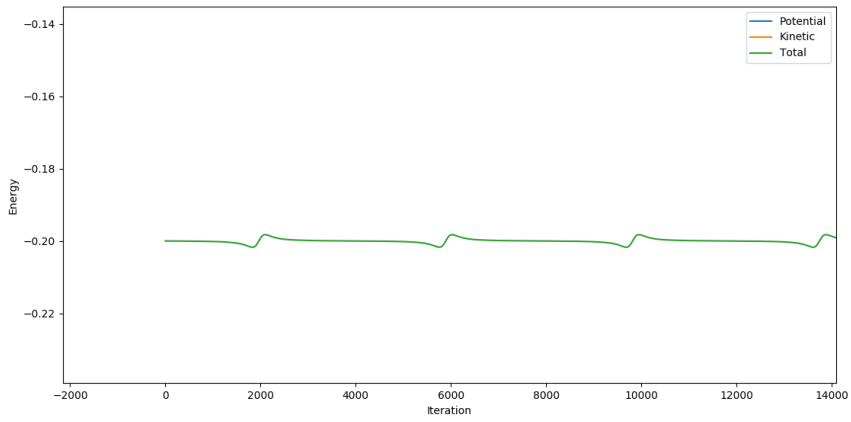
Rys. 7: Orbita planety używajac algorytmu leapfrog.

Wykres energii w algorytmie leapfrog:

Przybliżając wykres energii możemy zauważyć, że tutaj, podobnie jak w przypadku algorytmu Verleta, energia nie jest zachowana, lecz nie rośnie nieograniczenie.



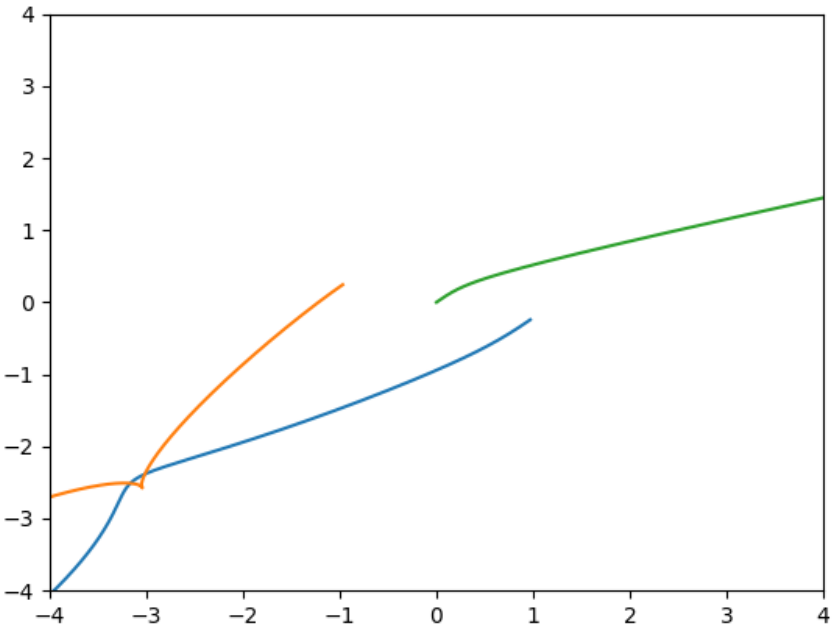
Rys. 8: Bilans energii w symulacji wykorzystując algorytm leapfrog.



Rys. 9: Przybliżenie wykresu energii w metodzie wykorzystującej algorytm leapfrog

Próby uzyskania lemniskaty Chencinera

Niestety, nie udało się uzyskać orbity w kształcie lemniskaty, mimo zastosowania podanych w pracy [1] warunków początkowych. Poniżej zostały przedstawione otrzymane rezultaty:



Rys. 10: Symulacja przeprowadzona dla warunków początkowych przedstawionych w pracy Chencinera.

Inne metody / fragmenty kodu użyte w symulacji

Klasa World - piaskownica symulacji:

```
class World:
        def __init__(self, G, planets, static_objects, dt):
                self.G = G
                self.planets = planets
                self.statics = static_objects
                self.dt = dt
                self.size = 2.1
```

Metoda licząca siłę, działającą na cząstkę:

```
def calculate_force(self, planet):
        f = np.array([0., 0.])

        for other in self.planets + self.statics:
            if planet is not other:
                d = World.d(other.pos, planet.pos)
                f +=  ( self.G * planet.m * other.m / d**3 )\
                      * (other.pos - planet.pos)

        return f
```

Metoda licząca energię kinetyczną i potencjalną cząstki:

```
def calculate_energy(self, p):
        pot = 0
        for o in self.planets + self.statics:
            if p is not o:
                pot += (-1) * self.G * o.m * p.m / World.d(o.pos, p.pos)
        kin = 0.5 * np.linalg.norm(p.mom) ** 2 / p.m
        return (kin, pot)
```

Metoda licząca odległość między dwiema cząsteczkami:

```
@staticmethod
def d(r1, r2):
        dist = ( (r1[0] - r2[0])**2 + (r1[1] - r2[1])**2  )**0.5
        return dist
```

References

[1] Oryginalna praca Chencinera http://emis.matem.unam.mx/journals/Annals/152_3/chencine.pdf