Modelowanie procesów fizycznych
Lab 04-05

Marcin Fabrykowski

18 kwietnia 2013

1 Metoda jawna

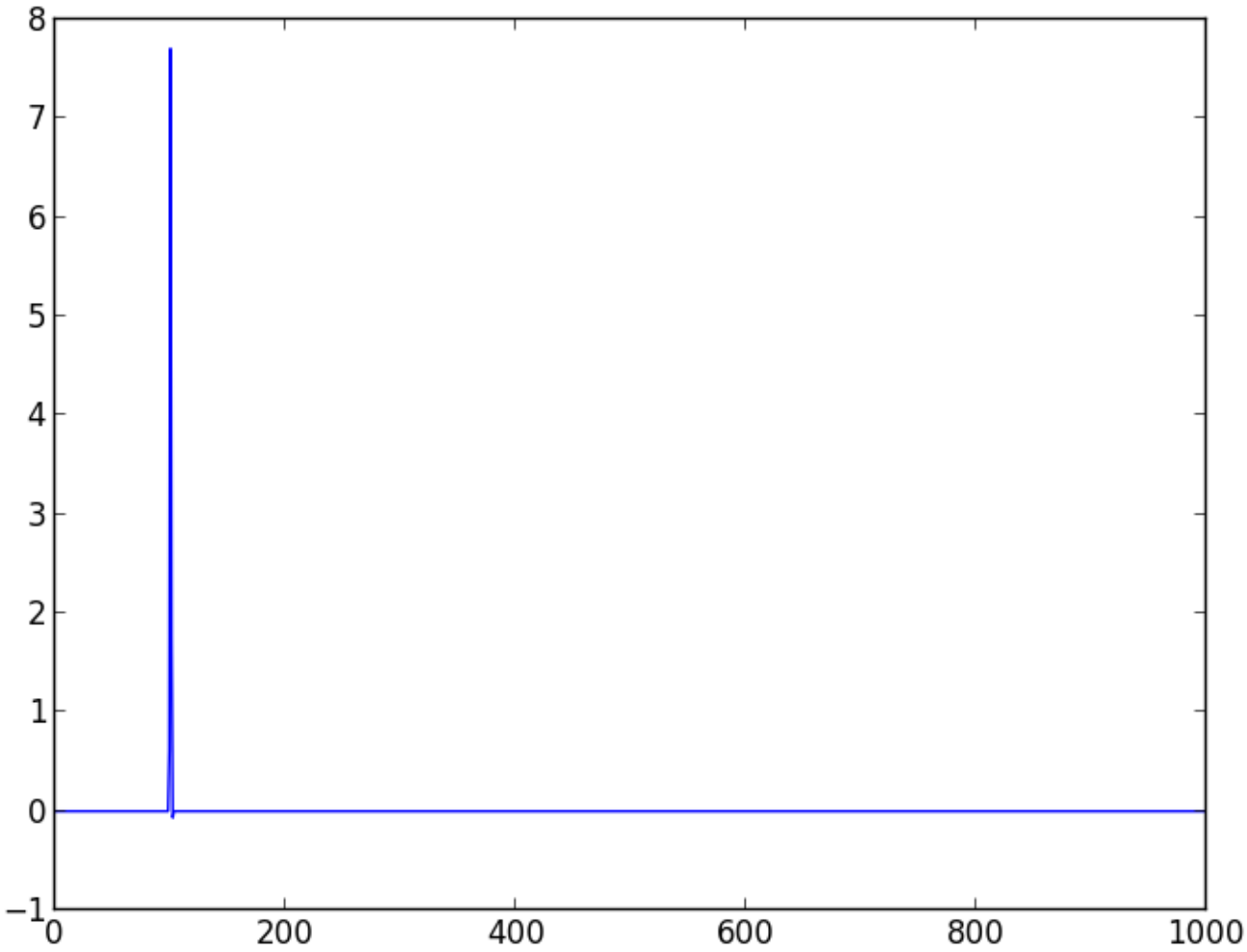
Została wykorzystana metoda QUICKEST. Charakteryzuje się ona szybkością obliczeń.
Na rysunkach 1, 2, 3 przedstawiono rozpływ cieczy fazie początkowej, środkowej oraz końcowej.
Na rysunku 4 przedstawiono stężenie cieczy w punkcie pomiarowym umieszczonym na pozycji 80

2 Metoda niejawna

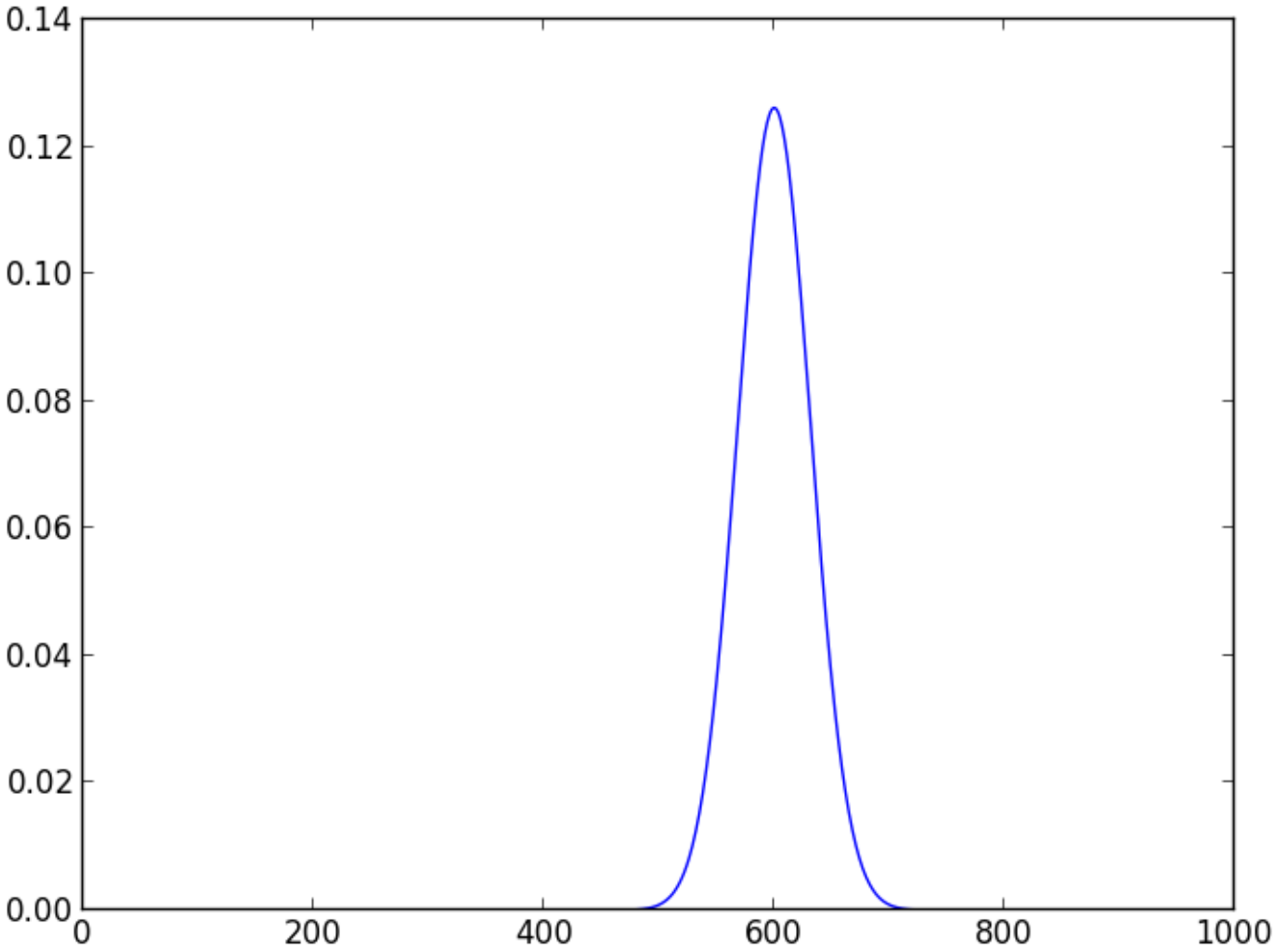
Została wykorzystana metoda Cranka-Nicolsona. Charakteryzuje się ona dokładnością obliczeń. Na rysunkach 5, 6, 7 przedstawiono rozpływ cieczy fazie początkowej, środkowej oraz końcowej.
Na rysunku 8 przedstawiono stężenie cieczy w punkcie pomiarowym umieszczonym na pozycji 80

3 Kod programu

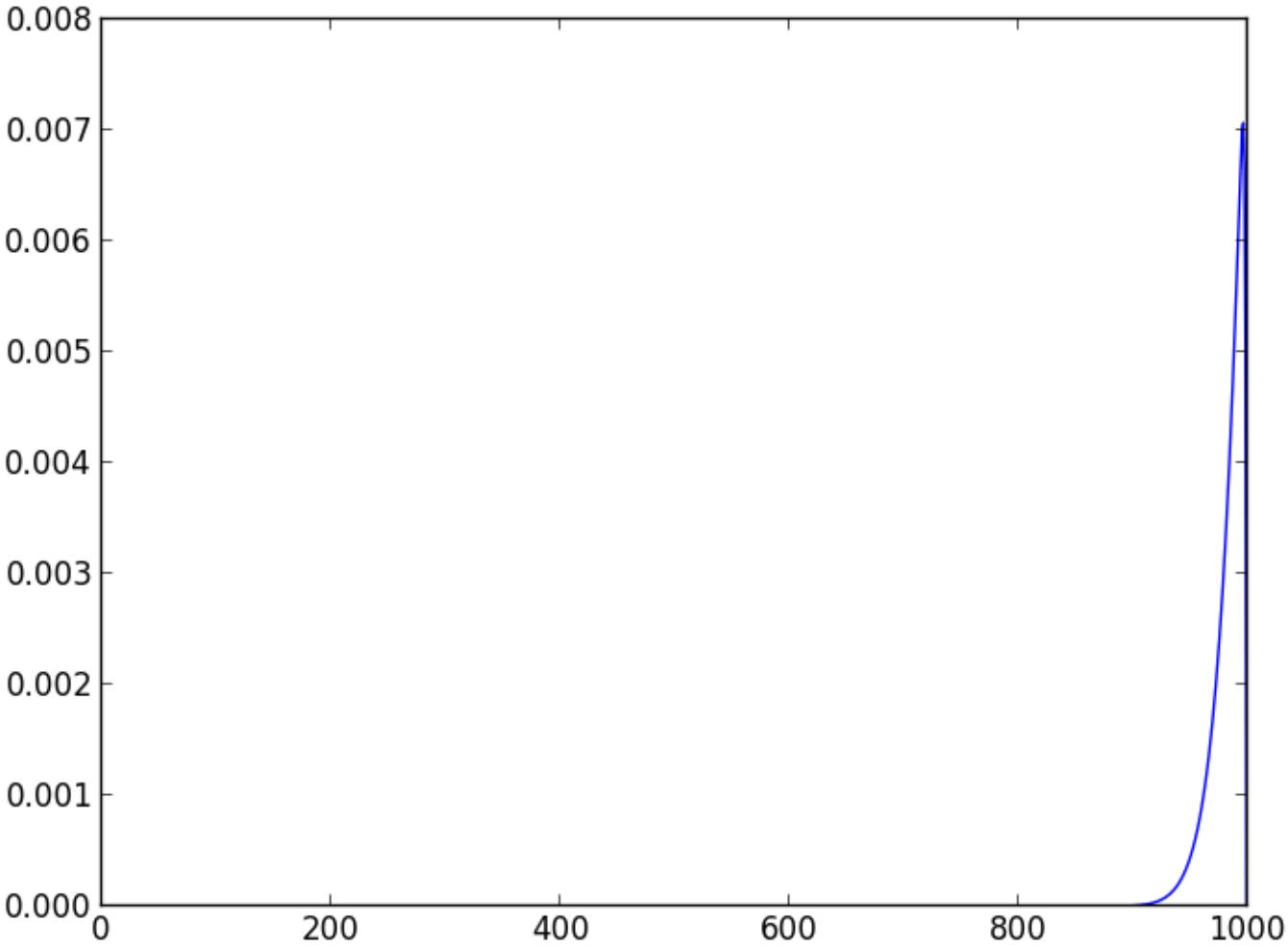
```
#!/usr/bin/env python
#*-* coding: utf-8 *-*
import numpy as np
import matplotlib.pyplot as plt
dx = 0.1
dt = 0.1
L = 100.0
d = 5.0
h = 1.0
U = 0.1
D = 0.01
x0 = 10.0 / dx
xp = 90.0 / dx
m = 1
Ca = U * dt / dx
Cd = D * dt / dx ** 2

dane1 = []
dane2 = []
```



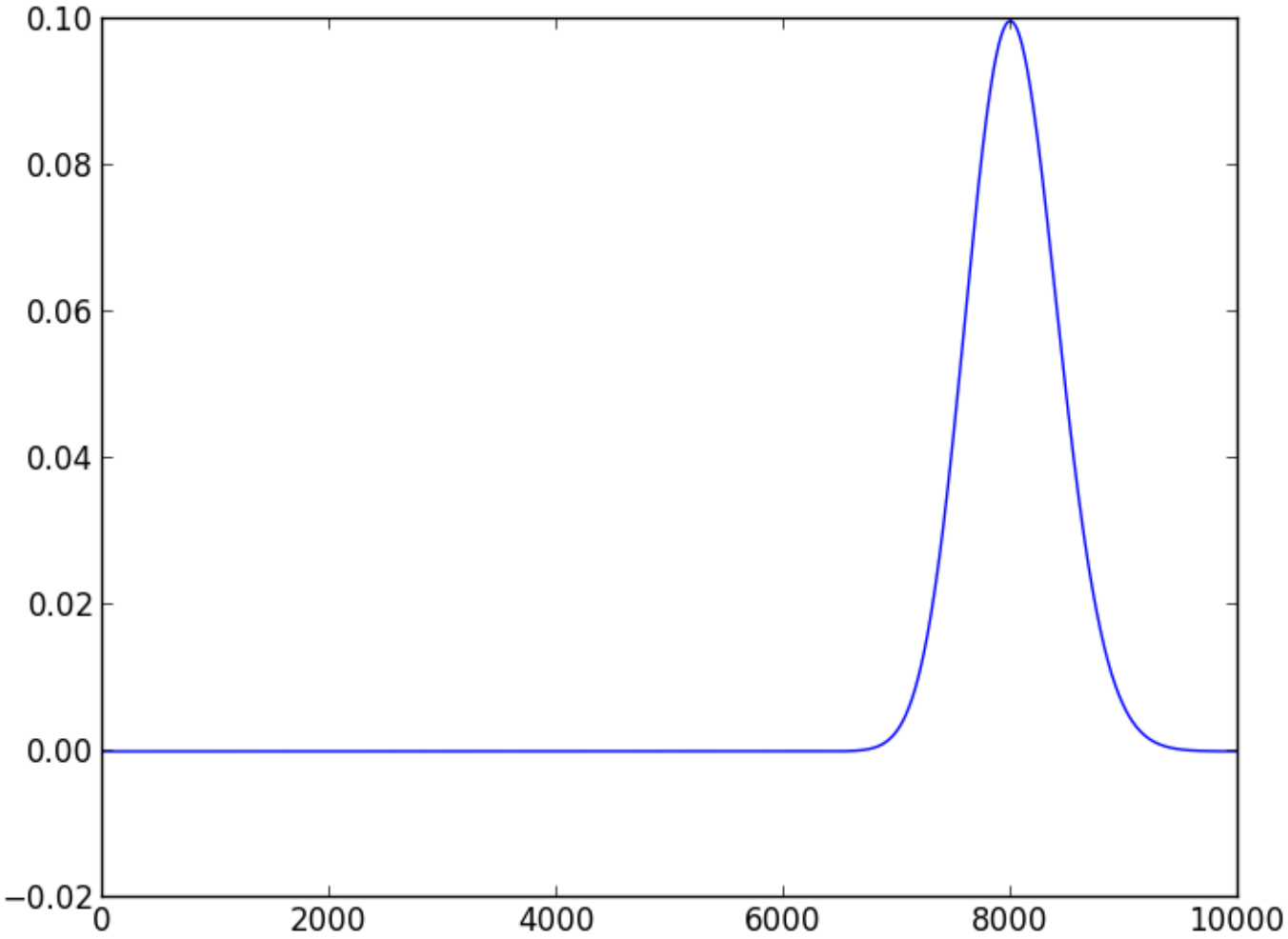
Rysunek 1: Metoda jawna - faza początkowwa



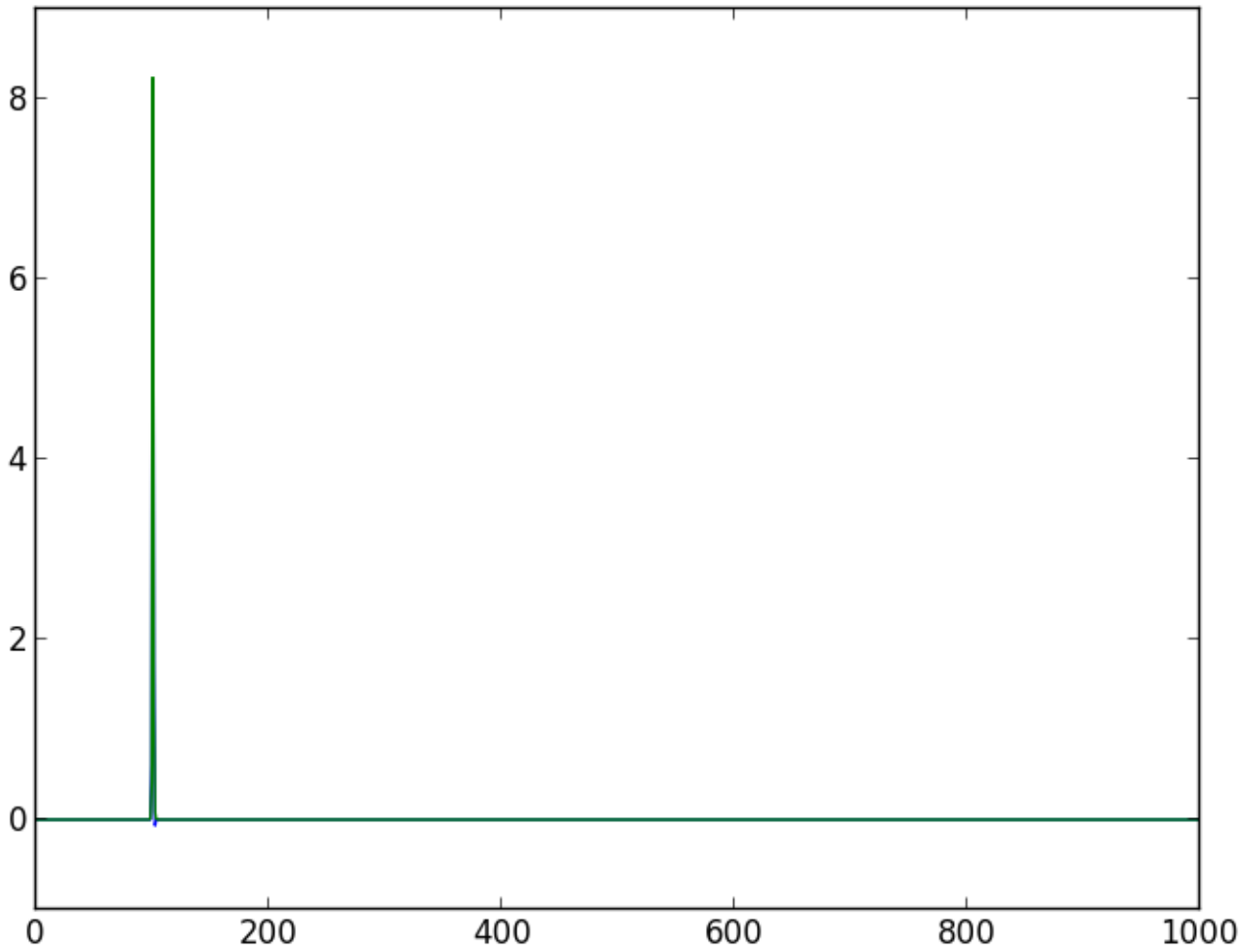
Rysunek 2: Metoda jawna - faza środkowa



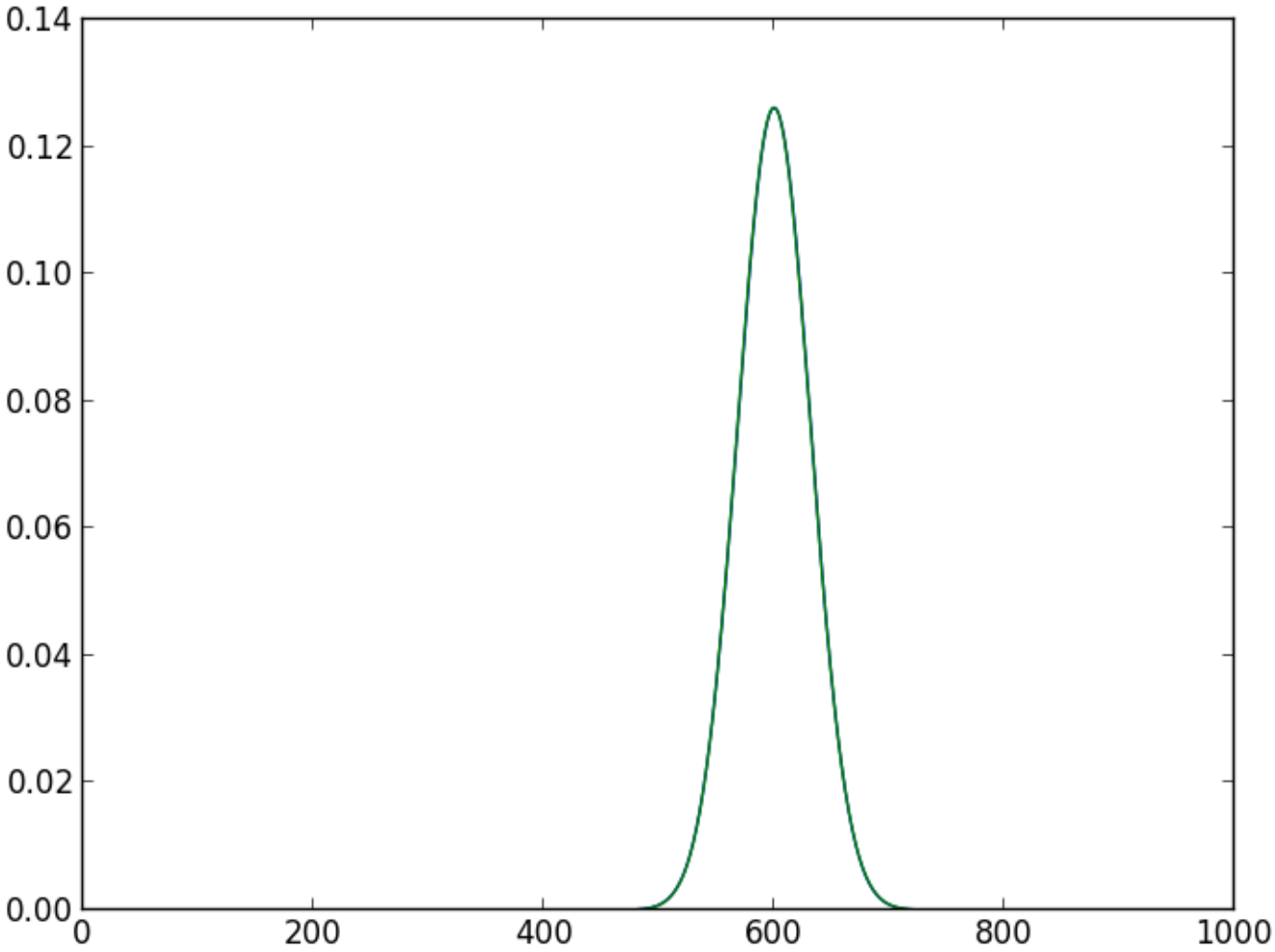
Rysunek 3: Metoda jawna - faza końcowa



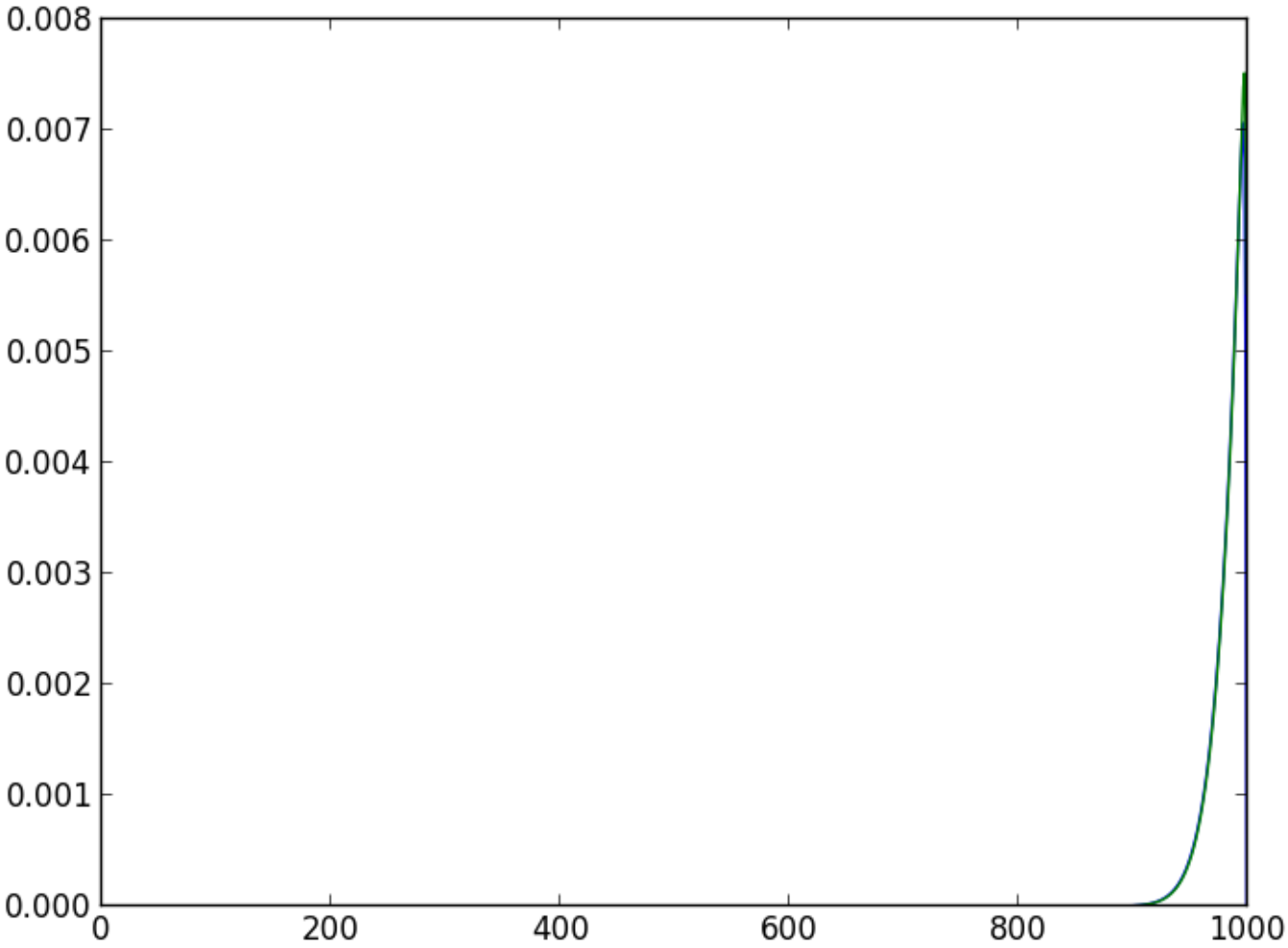
Rysunek 4: Metoda jawna - punk pomiarowy



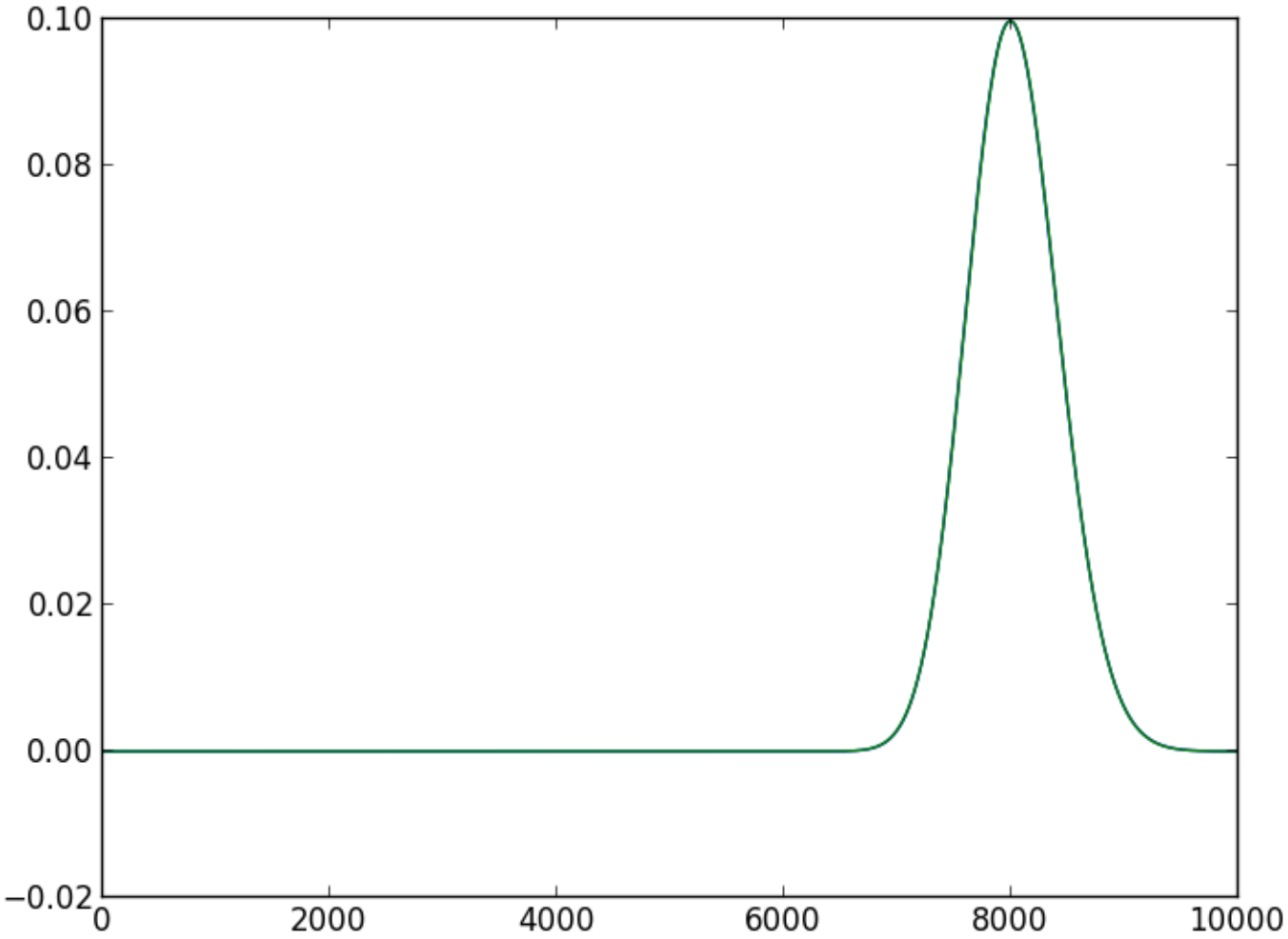
Rysunek 5: Metoda jawna - faza początkowwa



Rysunek 6: Metoda jawna - faza środkowa



Rysunek 7: Metoda jawna - faza końcowa



Rysunek 8: Metoda jawna - punk pomiarowy

```
def f1():
    return 10


def method1():
    num = 0
    C1 = Cd * (1.0 - Ca) - Ca / 6.0 * (Ca ** 2 - 3.0 * Ca + 2.0)
    C2 = Cd * (2.0 - 3.0 * Ca) - Ca / 2.0 * (Ca ** 2 - 2.0 * Ca - 1.0)
    C3 = Cd * (1.0 - 3.0 * Ca) - Ca / 2.0 * (Ca ** 2 - Ca - 2.0)
    C4 = Cd * Ca + Ca / 6.0 * (Ca ** 2 - 1.0)
    vec = np.zeros(L / dx)
    vec[x0] = f1()
    hist = []
    for x in xrange(10000):
#     print vec
        vec[2:-1] = vec[2:-1] + C1 * vec[3:] - C2 * vec[2:-1] + \
            C3 * vec[1:-2] + C4 * vec[:-3]
        hist.append(vec[xp])
        if not x % 50:
            print x, vec.sum()
            plt.figure(x)

            plt.plot(xrange(len(vec)), vec)
            plt.savefig("plots1/%05d.png" % num)
            num += 1
            print "fig"

    plt.figure(x + 1)
    plt.plot(xrange(len(hist)), hist)
    plt.savefig("plots1/hist.png")


def method2():
    num = 0
    vec = np.zeros(L / dx)
    vec[x0] = f1()
    hist = []
    s = int(L / dx)
    AA = np.diag([1.0 + Cd] * s)
    AA[0:-1, 1:] += np.diag([-Cd / 2.0 + Ca / 4.0] * (s - 1))
    AA[1:, 0:-1] += np.diag([-Cd / 2.0 - Ca / 4.0] * (s - 1))
```

```
    AA = np.linalg.inv(AA)

    BB = np.diag([1.0 - Cd] * s)
    BB[0:-1, 1:] += np.diag([Cd / 2.0 - Ca / 4.0] * (s - 1))
    BB[1:, 0:-1] += np.diag([Cd / 2.0 + Ca / 4.0] * (s - 1))

    CC = np.dot(AA, BB)
    vec[x0] = f1()
    vec = vec.transpose()
    for x in xrange(10000):
        vec = np.dot(CC, vec)
        hist.append(vec[xp])
        if not x % 50:
            print x, vec.sum()
            plt.figure(x)
            plt.plot(xrange(len(vec)), vec)
            plt.savefig("plots2/%05d.png" % num)
            num += 1
            print "fig"

    plt.figure(x + 1)
    plt.plot(xrange(len(hist)), hist)
    plt.savefig("plots2/hist.png")

method1()
method2()
```

4 Wnioski

Zauważamy, że obie metody dały bardzo porównywalne wyniki.
Oba algorytmy spełniają zasadę zachowania masy.