

Metody numeryczne: Python i C++

Większości równań ruchu nie da się rozwiązać na papierze, więc rozwiązuje je komputer, krok po kroku. Ta lekcja pokazuje cztery metody, które warto umieć napisać samemu: Eulera, Rungego–Kutty rzędu 4, symplektycznego Verleta i Newtona dla równań nieliniowych. Każdą zapisujemy obok siebie w Pythonie i w C++. Wniosek końcowy jest mierzalny: o czasie rachunku decyduje najpierw metoda i krok, a dopiero potem język.

CZAS

ok. 3 h teorii + 3 h
zadań i
programowania

ZADANIA

18 zadań w treści,
w tym projekt w
C++

ŹRÓDŁA

W. H. Press i in.,
Numerical Recipes,
wyd. 3, rozdz. 9,
17; E. Hairer, C.
Lubich, G.
Wanner,
*Geometric
Numerical
Integration*, rozdz.
I

W TEJ LEKCJI

- 0 Prolog: satelita, który uciekł
- 1 Błąd i rząd metody
- 2 Metoda Rungego–Kutty rzędu 4
- 3 Metody symplektyczne
- 4 Metoda Newtona
- 5 Python kontra C++: pomiar
- 6 Błędne intuicje
- 7 Sprawdź się
- 8 Karta wzorów i co dalej

Prolog: satelita, który uciekł

Satelita krąży po elipsie o mimośrodku 0,5 (jednostki, w których $GM = 1$, półosi $a = 1$, okres 2π). Jego energia całkowita wynosi $E = -\frac{1}{2a} = -0,5$ i powinna się nie zmieniać. Liczymy ruch metodą Eulera z lekcji 4.1 Równania różniczkowe zwyczajne, ze 100 krokami na okres. Po dziesięciu okresach energia wynosi +0,07: dodatnia, czyli satelita w symulacji uciekł z orbity. Nie zrobiła tego fizyka, tylko metoda.

Czy wystarczy mniejszy krok? Przy 10^7 krokach na okres (sto milionów kroków w sumie) Euler nadal myli się w położeniu o ok. półtorej setnej części. Metoda Rungego-Kutty z 1000 krokami na okres myli się o pół milionowej części. Ta lekcja wyjaśnia skąd ta różnica i jak ją zmierzyć.

Pytanie na start. Który program policzy tę orbitę dokładniej i szybciej: Euler w C++ czy Runge-Kutta w Pythonie? Odpowiedź w części 5.

Błąd i rząd metody

Rozwiązujemy zagadnienie początkowe $y' = f(t, y)$, $y(t_0) = y_0$ krokami długości h . Metoda Eulera $y_{n+1} = y_n + hf(t_n, y_n)$ to wielomian Taylora (lekcja 1.4 Rachunek różniczkowy w praktyce) urwany po wyrazie liniowym: $y(t+h) = y + hy' + \frac{h^2}{2}y'' + \dots$. Każdy krok wnosi więc **błąd lokalny** rzędu h^2 . Do chwili T robimy T/h kroków, a błędy się sumują, więc **błąd globalny** jest rzędu h .

🔧 RZĄD METODY

Metoda ma **rząd** p , jeśli błąd globalny w ustalonej chwili T zachowuje się jak Ch^p . Błąd lokalny jest wtedy rzędu h^{p+1} . Rząd wyznacza się doświadczalnie: dwukrotne zmniejszenie kroku zmniejsza błąd 2^p razy, więc

$$p \approx \log_2 \frac{e(h)}{e(h/2)}.$$

Na wykresie błędu w funkcji h w skali log-log rząd to nachylenie prostej.

por. Press i in. §17.1

Przykład 1.1

Jak szybko maleje błąd Eulera

Dla $y' = y$, $y(0) = 1$ policz $y(1)$ metodą Eulera z $h = 0,1$ i $h = 0,05$. Wyznacz rząd.

- 1 Dla $f = y$ jeden krok mnoży y przez $1 + h$. Z $h = 0,1$: $y(1) \approx 1,1^{10} = 2,5937$; błąd $e - 2,5937 = 0,1245$.
- 2 Z $h = 0,05$: $1,05^{20} = 2,6533$; błąd $0,0650$.
- 3 $p \approx \log_2 \frac{0,1245}{0,0650} = \log_2 1,92 \approx 0,94$.

Wynik: rząd 1: dwa razy mniejszy krok daje mniej więcej dwa razy mniejszy błąd.

Sprawdzenie: $(1 + h)^{1/h} \rightarrow e$ przy $h \rightarrow 0$ (lekcja 1.2 Ciągi, granice i ciągłość), a różnica to w przybliżeniu $\frac{e}{2}h$: dla $h = 0,1$ daje $0,136$, blisko $0,1245$ ✓.

Błąd $e - y(1)$ dla $y' = y$, $y(0) = 1$

krok h	Euler (rzęd 1)	Heun (rzęd 2)	RK4 (rzęd 4)
0,1	$1,25 \cdot 10^{-1}$	$4,20 \cdot 10^{-3}$	$2,08 \cdot 10^{-6}$
0,05	$6,50 \cdot 10^{-2}$	$1,09 \cdot 10^{-3}$	$1,36 \cdot 10^{-7}$
0,025	$3,32 \cdot 10^{-2}$	$2,78 \cdot 10^{-4}$	$8,67 \cdot 10^{-9}$

ZADANIA · CZĘŚĆ 1

4 zadania · ok. 30 min

- 1 Dla $y' = y$, $y(0) = 1$ zrób ręcznie pięć kroków Eulera z $h = 0,2$. Ile wynosi błąd $y(1)$? Ile razy jest większy niż przy $h = 0,1$?
- 2 Z rozwinięcia Taylora pokaż, że jeden krok Eulera ma błąd $\frac{h^2}{2}y''(\xi)$. Dlaczego błąd globalny jest rzędu h , a nie h^2 ?
- 3 Metoda Heuna: $k_1 = f(t_n, y_n)$, $k_2 = f(t_n + h, y_n + hk_1)$, $y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2)$. Pokaż, że dla $y' = y$ jeden krok mnoży y przez $1 + h + \frac{h^2}{2}$. Jaki stąd rząd? Sprawdź w tabeli.
- 4 Z tabeli wyznacz rząd RK4 dla obu par kroków.

2 Metoda Rungego–Kutty rzędu 4

Zamiast liczyć wyższe pochodne, metody Rungego–Kutty kilka razy obliczają f w punktach wewnątrz kroku i biorą średnią ważoną nachyleń. Wagi dobiera się tak, żeby wynik zgadzał się z szeregiem Taylora do możliwie wysokiej potęgi h .

⚡ RK4

$$\begin{aligned} k_1 &= f(t_n, y_n), & k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), & k_4 &= f(t_n + h, y_n + hk_3), \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

Cztery obliczenia f na krok, błąd globalny rzędu h^4 . Te same wzory działają, gdy y jest wektorem: równanie drugiego rzędu $x'' = a(x, x')$ zapisujemy jako układ dla $y = (x, v)$: $x' = v, v' = a(x, v)$. por. Press i in. §17.1

Dla $f = y$ wagi $1 : 2 : 2 : 1$ dają $1 + h + \frac{h^2}{2} + \frac{h^3}{6} + \frac{h^4}{24}$, czyli e^h do wyrazu h^4 (zadanie 2.1). Ta sama recepta to reguła Simpsona (lekcja 1.5 Całki), gdy f zależy tylko od t .

Poniżej obie metody w Pythonie i RK4 w C++. Wersja w Pythonie działa bez zmian dla liczb i dla tablic `numpy`. Wersja w C++ jest szablonem dla wektora stanu o stałej długości `std::array<double, N>`.

Python · rk.py

```
import numpy as np

def euler(f, t, y, h, n):
    for _ in range(n):
        y = y + h * f(t, y)
        t = t + h
    return y

def rk4(f, t, y, h, n):
    for _ in range(n):
        k1 = f(t, y)
        k2 = f(t + h/2, y + h/2 * k1)
        k3 = f(t + h/2, y + h/2 * k2)
        k4 = f(t + h, y + h * k3)
        y = y + h/6 * (k1 + 2*k2 + 2*k3 + k4)
        t = t + h
    return y

if __name__ == "__main__":
    f = lambda t, y: y                # y' = y, y(0) = 1, y(1) = e
    for h in (0.1, 0.05, 0.025):
        n = round(1 / h)
        print(h, np.e - euler(f, 0.0, 1.0, h, n), np.e - rk4(f, 0.0, 1.0, h, n))
```

C++ · rk4.cpp (g++ -O2 -std=c++17 rk4.cpp -o rk4)

```
#include <array>
#include <cmath>
#include <cstdio>

template <std::size_t N>
using Vec = std::array<double, N>;

template <std::size_t N>
Vec<N> axpy(const Vec<N>& y, double a, const Vec<N>& k) { // y + a·k
    Vec<N> r;
    for (std::size_t i = 0; i < N; ++i) r[i] = y[i] + a * k[i];
    return r;
}

template <std::size_t N, class F>
Vec<N> rk4(F f, double t, Vec<N> y, double h, long n) {
    for (long s = 0; s < n; ++s) {
        Vec<N> k1 = f(t, y);
        Vec<N> k2 = f(t + h / 2, axpy(y, h / 2, k1));
        Vec<N> k3 = f(t + h / 2, axpy(y, h / 2, k2));
        Vec<N> k4 = f(t + h, axpy(y, h, k3));
        for (std::size_t i = 0; i < N; ++i)
            y[i] += h / 6 * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]);
        t += h;
    }
    return y;
}

int main() {
    auto f = [](double, const Vec<1>& y) { return Vec<1>{y[0]}; };
    for (double h : {0.1, 0.05, 0.025}) {
        long n = std::lround(1 / h);
        Vec<1> y = rk4(f, 0.0, Vec<1>{1.0}, h, n);
        std::printf("h = %.3f  błąd = %.3e\n", h, std::exp(1.0) - y[0]);
    }
}
```

Oba programy wypisują te same błędy Eulera i RK4 co tabela w części 1 (`rk4.cpp` tylko RK4).
Analizę i wykresy wygodniej robić w Pythonie: `numpy` dopasuje prostą do błędów w skali log-log,
`matplotlib` narysuje wykres, a `scipy.integrate.solve_ivp` to gotowy integrator z automatyczną kontrolą kroku.

```

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from rk import euler, rk4

f = lambda t, y: y
hs = np.array([0.2, 0.1, 0.05, 0.025, 0.0125])
err_e = [abs(np.e - euler(f, 0.0, 1.0, h, round(1/h))) for h in hs]
err_r = [abs(np.e - rk4(f, 0.0, 1.0, h, round(1/h))) for h in hs]
p_e = np.polyfit(np.log(hs), np.log(err_e), 1)[0] # nachylenie = rząd metody
p_r = np.polyfit(np.log(hs), np.log(err_r), 1)[0]
print(f"rząd Eulera ≈ {p_e:.2f}, rząd RK4 ≈ {p_r:.2f}")

plt.loglog(hs, err_e, "o-", label="Euler")
plt.loglog(hs, err_r, "s-", label="RK4")
plt.xlabel("krok h"); plt.ylabel("błąd w t = 1"); plt.legend()
plt.savefig("rzad.png", dpi=120)

sol = solve_ivp(f, (0, 1), [1.0], method="RK45", rtol=1e-10, atol=1e-12)
print("solve_ivp:", sol.y[0, -1], " błąd:", np.e - sol.y[0, -1], " wywołań f:", sol.nfev)

```

Wynik: „rząd Eulera $\approx 0,95$, rząd RK4 $\approx 3,95$ ”, a `solve_ivp` osiąga błąd $5 \cdot 10^{-11}$ po 170 obliczeniach f , bo sam dobiera krok, porównując wyniki dwóch metod różnych rzędów.

Przykład 2.1

Ten sam koszt, inna dokładność

Dla $y' = y$ na $[0; 1]$ porównaj RK4 z $h = 0,1$ i Eulera z tą samą liczbą obliczeń f .

- 1 RK4: 10 kroków po 4 obliczenia, razem 40; błąd $2,1 \cdot 10^{-6}$.
- 2 Euler z 40 obliczeniami to $h = 0,025$; błąd $3,3 \cdot 10^{-2}$.
- 3 Żeby Euler osiągnął $2,1 \cdot 10^{-6}$, potrzebuje $h \approx \frac{2,1 \cdot 10^{-6}}{e/2} \approx 1,5 \cdot 10^{-6}$, czyli ok. 650 000 kroków.

Wynik: przy tym samym koszcie RK4 jest 16 000 razy dokładniejszy, a przy tej samej dokładności ok. 16 000 razy tańszy.

Sprawdzenie: bezpośredni rachunek z $N = 650\,000$ krokami Eulera daje błąd $2,09 \cdot 10^{-6}$ ✓.

ZADANIA · CZĘŚĆ 2

4 zadania · ok. 45 min

- 1 Zrób ręcznie jeden krok RK4 z $h = 0,1$ dla $y' = y$, $y(0) = 1$. Porównaj z $e^{0,1}$.

2 Użyj programu `rk.py` (albo `rk4.cpp`), tam tylko RK4) dla $y' = -2ty$, $y(0) = 1$. Policz $y(1)$ metodą Eulera i RK4 z $h = 0,2$ i $h = 0,1$. Porównaj z rozwiązaniem dokładnym.

3 Zapisz oscylator tłumiony $x'' + 0,1x' + x = 0$ jako układ pierwszego rzędu i podaj funkcję `f` do `rk4` w Pythonie i w C++.

4 Metoda rzędu p z s obliczeniami f na krok ma błąd $\approx C(s/N)^p$, gdzie N to liczba obliczeń f . Ile razy trzeba zwiększyć N , żeby błąd zmalał 10^4 razy, dla Eulera i dla RK4?

▀▀▀ podstawowe ▀▀▀ średnie ▀▀▀ feynmanowskie

3 Metody symplektyczne

Mały błąd po jednym okresie to nie wszystko. W mechanice nieba i dynamice molekularnej liczy się tysiące okresów, a wtedy ważne jest, czy błąd energii *narasta*. Najprościej widać to na oscylatorze $x'' = -x$ (energia $E = \frac{1}{2}(x^2 + v^2)$).

Euler: $x_{n+1} = x_n + hv_n$, $v_{n+1} = v_n - hx_n$. Stąd $x_{n+1}^2 + v_{n+1}^2 = (1 + h^2)(x_n^2 + v_n^2)$: energia rośnie geometrycznie, a tor jest spiralą na zewnątrz. Wystarczy jednak jedna zmiana kolejności, żeby to naprawić.

🔧 EULER PÓŁJAWNY I VERLET PRĘDKOŚCIOWY

- **Euler półjawnny (symplektyczny):** $v_{n+1} = v_n + ha(x_n)$, potem $x_{n+1} = x_n + hv_{n+1}$ (nowa prędkość!). Rząd 1.
- **Verlet prędkościowy:** $v_{n+1/2} = v_n + \frac{h}{2}a(x_n)$, $x_{n+1} = x_n + hv_{n+1/2}$, $v_{n+1} = v_{n+1/2} + \frac{h}{2}a(x_{n+1})$. Rząd 2, jedno nowe obliczenie siły na krok (przyspieszenie $a(x_{n+1})$ przechodzi do następnego kroku).

Obie metody są **symplektyczne**: krok zachowuje pole (objętość) w przestrzeni fazowej (x, v) , tak jak dokładny ruch hamiltonowski. Dzięki temu prawie dokładnie zachowują nieco zmienioną energię (dla oscylatora dokładnie), a błąd prawdziwej energii pozostaje ograniczony przez wykładniczo długi czas, w praktyce przez każdy czas, jaki liczymy. (por. Hairer, Lubich, Wanner rozdz. I.1, I.3)

Python · verlet.py

```
def verlet(x, v, a, h, n):
    """Prędkościowy Verlet dla x'' = a(x)."""
    acc = a(x)
    for _ in range(n):
        v = v + h/2 * acc
        x = x + h * v
        acc = a(x)          # jedno nowe obliczenie siły na krok
        v = v + h/2 * acc
    return x, v

if __name__ == "__main__":
    x, v = verlet(1.0, 0.0, lambda x: -x, 0.1, 1000)  # oscylator, t = 100
    print(x, v, (x*x + v*v) / 2)
```

C++ · verlet.cpp

```
#include <cstdio>

template <class A>
void verlet(double& x, double& v, A a, double h, long n) {
    double acc = a(x);
    for (long i = 0; i < n; ++i) {
        v += h / 2 * acc;
        x += h * v;
        acc = a(x);          // jedno nowe obliczenie siły na krok
        v += h / 2 * acc;
    }
}

int main() {
    double x = 1.0, v = 0.0;
    verlet(x, v, [](double q) { return -q; }, 0.1, 1000); // oscylator, t = 100
    std::printf("%.6f %.6f %.8f\n", x, v, (x * x + v * v) / 2);
}
```

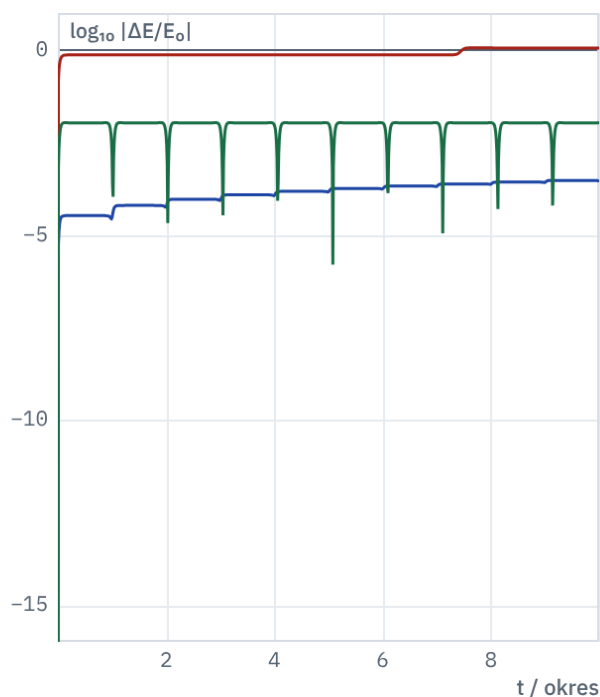
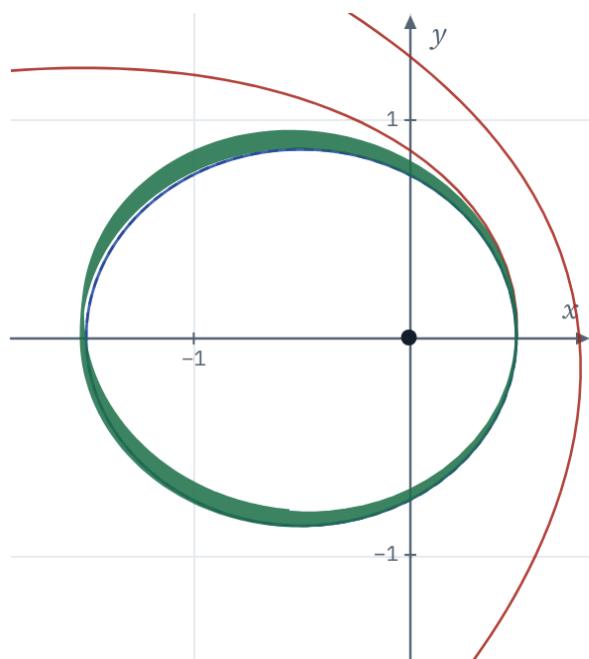
Oscylator $x'' = -x$, $h = 0,1$: stosunek energii E/E_0

metoda	$t = 10$	$t = 100$	$t = 1000$
Euler	2,70	20 959	$1,6 \cdot 10^{40}$
Euler półniejawny	0,956	1,043	0,987
Verlet	0,99925	0,99945	0,99758
RK4	0,9999986	0,999986	0,99986

Przy tym kroku RK4 jest najdokładniejszy, ale jego energia maleje *systematycznie*, liniowo z czasem. Energia Verleta tylko oscyluje w paśmie o względnej szerokości ok. $h^2/4$ (tu 0,25%). Na orbicie z prologu (mimośród 0,5), przy tym samym koszcie (200 kroków RK4 albo 800 kroków Verleta na okres), błąd energii RK4 po 10, 100, 1000 i 10 000 okresach wynosi $4,6 \cdot 10^{-6}$, $4,6 \cdot 10^{-5}$, $4,6 \cdot 10^{-4}$, $4,65 \cdot 10^{-3}$, a Verleta nie przekracza ok. 10^{-4} .

SYMULACJA 1

Orbita Keplera liczona trzema metodami: Eulera, RK4 i Verleta. Lewy (na telefonie: górny) wykres pokazuje tory, prawy logarytm błędu energii w czasie. Zmieniaj liczbę kroków na okres, liczbę okresów i mimośród. Odczyt podaje błąd położenia po ostatnim okresie, największy względny błąd energii i liczbę obliczeń siły.



— Euler — RK4 — Verlet — — dokładna orbita

Euler: błąd położenia 11,6

RK4: błąd położenia $2,71 \cdot 10^{-2}$

Verlet: błąd położenia 1,32

Euler: max $|\Delta E/E_0|$ 1,14

RK4: max $|\Delta E/E_0|$ $2,99 \cdot 10^{-4}$

Verlet: max $|\Delta E/E_0|$ $1,08 \cdot 10^{-2}$

obliczenia siły (E / RK4 / V) 1 000 / 4 000 / 1 000

Euler daleko od orbity: energia rośnie z każdym okresem, tor rozwija się w spiralę. RK4 trzyma orbitę, Verlet już nie.

Stan na rysunku: 100 kroków na okres, 10 okresów, mimośród 0,5; błąd położenia: Euler 11,6, RK4 $2,71 \cdot 10^{-2}$, Verlet 1,32.

ZADANIA · CZĘŚĆ 3

5 zadań · ok. 45 min

- 1 Pokaż, że Euler dla $x'' = -x$ mnoży $x^2 + v^2$ przez $1 + h^2$ w każdym kroku. O ile wzrośnie energia do $t = 10$ i $t = 100$ przy $h = 0,1$?

- 2 Zapisz krok Eulera i Eulera półniejawnego dla oscylatora jako mnożenie wektora (x, v) przez macierz. Oblicz wyznaczniki (lekcja 2.1 Macierze, wyznaczniki i układy równań). Co oznaczają?
- 3 Sprawdź, że Euler półniejawny dla oscylatora dokładnie zachowuje $\tilde{E} = x^2 + v^2 - h x v$, a Verlet zachowuje $(1 - \frac{h^2}{4})x^2 + v^2$. Co z tego wynika dla prawdziwej energii?
- 4 Dlaczego Verlet potrzebuje jednego obliczenia siły na krok, choć we wzorach występują $a(x_n)$ i $a(x_{n+1})$? Ile kroków Verleta kosztuje tyle co jeden krok RK4?
- 5 Po 10 000 okresach orbity z tekstu obie metody mają błąd *położenia* rzędu promienia orbity, choć Verlet trzyma energię. Jak to możliwe? Którą metodę wybrać do symulacji Układu Słonecznego na miliony lat?

▣▣ podstawowe ▣▣▣ średnie ▣▣▣▣ feynmanowskie

4 Metoda Newtona

Równania nieliniowe $f(x) = 0$ pojawiają się wszędzie: w niejawnych metodach całkowania, w równaniu Keplera $E - e \sin E = M$ (położenie planety w czasie), w punktach równowagi. Metoda Newtona zastępuje f styczną (lekcja 1.4 Rachunek różniczkowy w praktyce) i bierze jej miejsce zerowe.

🔧 METODA NEWTONA

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Jeśli $f(x^*) = 0$, $f'(x^*) \neq 0$ i start jest dość blisko, błąd $e_n = x_n - x^*$ spełnia $e_{n+1} \approx \frac{f''(x^*)}{2f'(x^*)} e_n^2$: **zbieżność kwadratowa**, liczba poprawnych cyfr mniej więcej podwaja się w każdym kroku. Daleko od pierwiastka metoda może się rozbiec albo wpaść w cykl.

por. Press i in. §9.4

Skąd kwadrat: z Taylora $0 = f(x^*) = f(x_n) - f'(x_n)e_n + \frac{1}{2}f''(\xi)e_n^2$. Dzieląc przez $f'(x_n)$: $e_n - \frac{f(x_n)}{f'(x_n)} = \frac{f''(\xi)}{2f'(x_n)}e_n^2$, a lewa strona to właśnie e_{n+1} .

Przykład 4.1

$\sqrt{2}$

jak w Babilonie

Rozwiąż $x^2 - 2 = 0$ metodą Newtona z $x_0 = 1$.

- 1 $x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{1}{2} \left(x_n + \frac{2}{x_n} \right)$: średnia z x_n i $2/x_n$.
- 2 $x_1 = 1,5$; $x_2 = 1,41667$; $x_3 = 1,4142157$; $x_4 = 1,41421356237469$.
- 3 Błędy: $8,6 \cdot 10^{-2}$; $2,5 \cdot 10^{-3}$; $2,1 \cdot 10^{-6}$; $1,6 \cdot 10^{-12}$.

Wynik: po czterech krokach 12 poprawnych cyfr.

Sprawdzenie: $\frac{f''}{2f'} = \frac{1}{2\sqrt{2}} \approx 0,354$; $0,354 \cdot (2,1 \cdot 10^{-6})^2 = 1,6 \cdot 10^{-12} \checkmark$.

Python · newton.py

```
import math

def newton(f, df, x, tol=1e-12, maxit=50):
    for i in range(maxit):
        d = df(x)
        if d == 0:
            raise ZeroDivisionError("f'(x) = 0: styczna pozioma")
        dx = f(x) / d
        x -= dx
        if abs(dx) < tol * max(1.0, abs(x)):
            return x, i + 1
    raise RuntimeError("brak zbieżności")

if __name__ == "__main__":
    M, e = 2.0, 0.9 # równanie Keplera E - e sin E = M
    E, it = newton(lambda E: E - e * math.sin(E) - M,
                    lambda E: 1 - e * math.cos(E), M)
    print(f"E = {E:.15f} po {it} krokach")
```

```

#include <cmath>
#include <cstdio>
#include <stdexcept>

template <class F, class D>
double newton(F f, D df, double x, int& it, double tol = 1e-12, int maxit = 50) {
    for (it = 1; it <= maxit; ++it) {
        double d = df(x);
        if (d == 0) throw std::runtime_error("f'(x) = 0: styczna pozioma");
        double dx = f(x) / d;
        x -= dx;
        if (std::fabs(dx) < tol * std::fmax(1.0, std::fabs(x))) return x;
    }
    throw std::runtime_error("brak zbieżności");
}

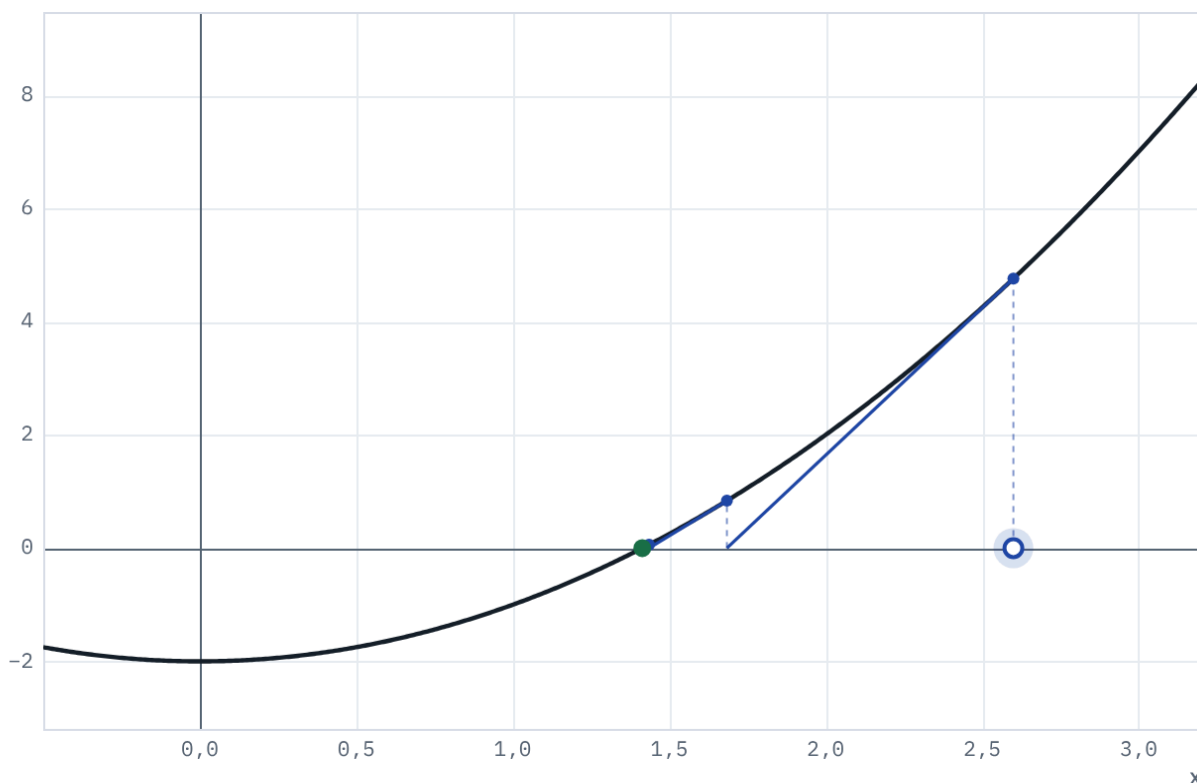
int main() {
    double M = 2.0, e = 0.9; // równanie Keplera E - e sin E = M
    int it = 0;
    double E = newton([&](double z) { return z - e * std::sin(z) - M; },
        [&](double z) { return 1 - e * std::cos(z); }, M, it);
    std::printf("E = %.15f po %d krokach\n", E, it);
}

```

Kryterium stopu porównuje poprawkę $|\Delta x|$ z tolerancją względną (dla $|x| < 1$ bezwzględną). Gdy $f'(x_n) = 0$, programy zgłaszają błąd zamiast dzielić przez zero. Samo $|f(x)| < \varepsilon$ bywa zwodnicze: dla bardzo płaskiej funkcji f jest małe daleko od pierwiastka.

SYMULACJA 2

Metoda Newtona krok po kroku: styczne, kolejne przybliżenia i ich odległość od pierwiastka. Wybierz funkcję i przeciągnij punkt startowy x_0 . Dla $\arctg x$ znajdź granicę między zbieżnością a rozbieżnością; dla $x^3 - 2x + 2$ start z 0 daje cykl.



— $f(x)$ — styczne i kolejne przybliżenia — pierwiastek

x_0 2,6000000000

x_1 1,6846153846

x_2 1,4359149982

x_3 1,4143775527

x_4 1,4142135719

x_5 1,4142135624

$|x_5 - \text{pierwiastek}|$ $2,22 \cdot 10^{-16}$

Zbieżność kwadratowa: liczba poprawnych cyfr rośnie mniej więcej dwukrotnie w każdym kroku (0 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 8 $\rightarrow$ 15).

Stan na rysunku: $f(x) = x^2 - 2$, $x_0 = 2,60$; kolejne przybliżenia 2,600000; 1,684615; 1,435915; 1,414378; 1,414214; 1,414214.

ZADANIA · CZĘŚĆ 4

4 zadania · ok. 35 min

- Metodą Newtona oblicz (a) $\sqrt[3]{10}$ z $x_0 = 2$; (b) rozwiązanie $x = \cos x$ z $x_0 = 1$. Podaj trzy pierwsze przybliżenia.

2 Równanie Keplera $E - e \sin E = M$ wiąże czas (anomalja średnia M) z położeniem na orbicie (anomalja mimośrodowa E). Dla $e = 0,9$, $M = 2$ rozwiąż je metodą Newtona z $E_0 = M$.

3 Zastosuj metodę Newtona do $x^3 - 2x + 2 = 0$ z $x_0 = 0$. Co się dzieje? Gdzie jest pierwiastek?

4 Dla $f = \arctg x$ pokaż, że start z $\pm x^*$, gdzie $\arctg x^* (1 + x^{*2}) = 2x^*$, daje cykl $x^* \rightarrow -x^* \rightarrow x^*$. Znajdź x^* numerycznie. Co się dzieje dla $|x_0| < x^*$ i $|x_0| > x^*$?

▢▢▢ podstawowe ▢▢▢ średnie ▢▢▢ feynmanowskie

5 Python kontra C++: pomiar

Zmierzyliśmy czas liczenia orbity z prologu (10 okresów) programami w czystym Pythonie i w C++ (kompilacja `g++ -O2`). Bezwzględne czasy zależą od komputera; liczą się stosunki.

Orbita o mimośrodku 0,5, 10 okresów

program	kroków	błąd położenia	czas
Euler, C++	10^8	$1,5 \cdot 10^{-2}$	1,1 s
RK4, Python	10^4	$5,3 \cdot 10^{-7}$	0,047 s
RK4, C++	10^4	$5,3 \cdot 10^{-7}$	0,0005 s
RK4, Python	10^5	$2,9 \cdot 10^{-11}$	0,46 s
RK4, C++	10^5	$2,9 \cdot 10^{-11}$	0,005 s

🔨 MORAŁ

Ta sama metoda w C++ jest kilkadziesiąt do stu razy szybsza niż w czystym Pythonie (tu ok. 90 razy). Lepsza metoda daje jednak czynniki rzędu tysięcy i milionów: RK4 w Pythonie jest ok. 20 razy szybszy i 30 000 razy dokładniejszy niż Euler w C++. Kolejność decyzji: najpierw metoda i krok, potem biblioteki (`numpy`, `scipy`), a dopiero na końcu język kompilowany dla pętli, które naprawdę dominują czas.

- 1 Projekt. Napisz w C++ program `orbita.cpp`, który liczy orbitę Keplera ($GM = 1$, start w peryhelium: $x = 1 - e$, $y = 0$, $v_x = 0$, $v_y = \sqrt{(1+e)/(1-e)}$, okres 2π) metodą podaną w wierszu poleceń (Euler, RK4, Verlet), zadaną liczbą kroków na okres i okresów. Na końcu ma wypisać błąd położenia (odległość od punktu startu), zmianę energii $E = \frac{1}{2}v^2 - 1/r$ i czas rachunku zmierzony `std::chrono::steady_clock`. Odtwórz tabelę powyżej i sprawdź rząd każdej metody, zwiększając liczbę kroków 10 razy.

▣▣ podstawowe ▣▣▣ średnie ▣▣▣▣ feynmanowskie

6 Błędne intuicje

× „Wystarczy zmniejszyć krok.”

Dla metody rzędu 1 dokładność 10^{-6} oznacza miliony kroków, a przy bardzo małym h zaczyna przeszkadzać zaokrąglanie liczb zmiennoprzecinkowych. Zmiana metody daje więcej niż zmiana kroku.

× „C++ jest szybszy, więc wynik będzie lepszy.”

Język zmienia czas o stały czynnik, a nie dokładność. Euler w C++ przegrywa z RK4 w Pythonie w obu kategoriach (część 5).

× „Metoda wyższego rzędu jest zawsze lepsza.”

W bardzo długich symulacjach układów zachowawczych symplektyczny Verlet rzędu 2 zachowuje energię lepiej niż RK4, który dryfuje.

× „Metoda Newtona zawsze znajdzie pierwiastek.”

Tylko startując dość blisko. Może się rozbiec ($\arctg x$), wpaść w cykl ($x^3 - 2x + 2$) albo zatrzymać się, gdy $f'(x_n) = 0$.

7 Sprawdź się

1 Co znaczy, że metoda ma rząd p ?

Błąd globalny maleje jak h^p ; połowa kroku zmniejsza błąd 2^p razy.

2 Ile razy RK4 oblicza f w jednym kroku?

Cztery razy.

3 Jak zamienić równanie drugiego rzędu na układ pierwszego rzędu?

Wprowadzić prędkość: $x' = v, v' = a(x, v)$.

4 Co się dzieje z energią oscylatora w metodzie Eulera?

Rośnie o czynnik $1 + h^2$ na krok.

5 Czym różni się Euler półniejawny od zwykłego?

Położenie aktualizuje nową prędkością; zachowuje pole w przestrzeni fazowej.

6 Zapisz krok metody Newtona.

$$x_{n+1} = x_n - f(x_n)/f'(x_n).$$

7 Co decyduje o czasie rachunku najbardziej?

Metoda i krok; język daje stały czynnik.

RÓWNANIA RÓŻNICZKOWE

Euler: $y_{n+1} = y_n + hf(t_n, y_n)$; rząd 1

RK4: $y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$;
rząd 4, 4 obliczenia f

Verlet: $v += \frac{h}{2}a$; $x += hv$; $a = a(x)$; $v += \frac{h}{2}a$; rząd 2, symplektyczny

rząd z pomiaru: $p \approx \log_2 \frac{e(h)}{e(h/2)}$

RÓWNANIA NIELINIOWE I PRAKTYKA

Newton: $x_{n+1} = x_n - f(x_n)/f'(x_n)$;

$$e_{n+1} \approx \frac{f''}{2f'} e_n^2$$

stop: $|\Delta x| < \text{tol} \cdot \max(1, |x|)$

Python: `scipy.integrate.solve_ivp`,
`scipy.optimize.brentq`, `newton`

C++: `g++ -O2 -std=c++17`; czas:

`std::chrono::steady_clock`

Do przeczytania

- W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, wyd. 3: rozdz. 9 (pierwiastki równań, w tym Newton), rozdz. 17 (równania różniczkowe zwyczajne, Runge–Kutta, kontrola kroku). Kod w C++.
- E. Hairer, C. Lubich, G. Wanner, *Geometric Numerical Integration*, wyd. 2, rozdz. I: przykłady (oscylator, Kepler, Układ Słoneczny) pokazujące, dlaczego metody symplektyczne zachowują energię.

Te metody wrócą w automatyce przy symulacji układów sterowania i w filtrze Kalmana (4.4 Filtr Kalmana), a w astronautyce przy orbitach i równaniu Keplera. Równania ruchu z zasady działania (lekcja 6.1 Rachunek wariacyjny) to dokładnie te, które całkuje Verlet.



Odpowiedzi

Zadania • część 1

- 1,2; 1,44; 1,728; 2,0736; 2,48832. Błąd $e - 2,48832 \approx 0,2300$, czyli 1,85 razy więcej niż 0,1245. Blisko 2, jak dla rzędu 1.
- $y(t+h) = y(t) + hy'(t) + \frac{h^2}{2}y''(\xi)$ (reszta Lagrange'a), a Euler bierze dwa pierwsze wyrazy. Do chwili T jest T/h kroków; jeśli błędy się sumują (i nie są wzmacniane bardziej niż o stały czynnik), razem dają $\frac{T}{h} \cdot O(h^2) = O(h)$.
- $k_1 = y, k_2 = y(1+h), y_{n+1} = y(1 + \frac{h}{2}(2+h)) = y(1+h + \frac{h^2}{2})$: Taylor e^h do h^2 , błąd lokalny h^3 , globalny h^2 . Tabela: $\frac{4,20}{1,09} \approx 3,85$ i $\frac{1,09}{0,278} \approx 3,93$, czyli prawie $2^2 = 4$.
- $\log_2 \frac{2,084 \cdot 10^{-6}}{1,358 \cdot 10^{-7}} = \log_2 15,35 \approx 3,94$; $\log_2 \frac{1,358 \cdot 10^{-7}}{8,666 \cdot 10^{-9}} = \log_2 15,67 \approx 3,97$. Rząd 4.

Zadania • część 2

- $k_1 = 1, k_2 = 1,05, k_3 = 1,0525, k_4 = 1,10525$; $y_1 = 1 + \frac{0,1}{6}(1 + 2,1 + 2,105 + 1,10525) = 1,1051708$. $e^{0,1} = 1,1051709$; różnica $8 \cdot 10^{-8} \approx \frac{h^5}{120}$.
- Dokładnie $e^{-1} = 0,36787944$. Euler: 0,399383 ($h = 0,2$), 0,381707 ($h = 0,1$). RK4: 0,36790367 i 0,36788107; błędy $2,4 \cdot 10^{-5}$ i $1,6 \cdot 10^{-6}$, stosunek ok. 15.
- $y = (x, v), f(t, y) = (v, -x - 0,1v)$. Python: `f = lambda t, y: np.array([y[1], -y[0] - 0.1*y[1]])`. C++: `auto f = [](double, const Vec<2>& y) { return Vec<2>{y[1], -y[0] - 0.1 * y[1]}; }`
- Błąd $\propto N^{-p}$, więc N rośnie $10^{4/p}$ razy: Euler 10 000 razy, RK4 tylko 10 razy. Dlatego metoda wyższego rzędu zwykle wygrywa z przyspieszeniem sprzętu czy języka.

Zadania • część 3

- $(x + hv)^2 + (v - hx)^2 = (1 + h^2)(x^2 + v^2)$, bo wyrazy mieszane się znoszą. $1,01^{100} \approx 2,70$ i $1,01^{1000} \approx 20\,959$, jak w tabeli.
- Euler: $\begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix}$, wyznacznik $1 + h^2$. Późniejawny (v najpierw): $\begin{pmatrix} 1 - h^2 & h \\ -h & 1 \end{pmatrix}$, wyznacznik 1. Wyznacznik to współczynnik zmiany pola: Euler powiększa pole każdego obszaru przestrzeni fazowej, metoda późniejsza zachowuje pole, tak jak ruch dokładny.
- Podstawienie $v_{n+1} = v_n - hx_n, x_{n+1} = x_n + hv_{n+1}$ i wymnożenie daje $\tilde{E}_{n+1} = \tilde{E}_n$ (podobnie dla Verleta). Wielkość zachowana różni się od $x^2 + v^2$ o wyraz rzędu h (albo h^2), więc prawdziwa energia może się wahać tylko w paśmie tej szerokości: dla Verleta z $h = 0,1$ to ok. 0,25%, zgodnie z tabelą. Nie może dryfować.
- $a(x_{n+1})$ obliczone na końcu kroku to $a(x_n)$ następnego kroku: w programie trzymamy je w zmiennej `acc`. Siłę liczy się więc raz na krok, a RK4 cztery razy; stąd porównanie 800 kroków Verleta z 200 krokami RK4.
- W Verlecie energia, a więc i okres, nie dryfuje; zostaje stały błąd częstości, więc błąd fazy rośnie liniowo. W RK4 energia, a z nią okres, dryfuje, więc błąd fazy rośnie szybciej, mniej więcej kwadratowo (po 10 i 100 okresach: $9 \cdot 10^{-4}$ i $8 \cdot 10^{-2}$). W obu przypadkach satelita jest na prawie

właściwej orbicie, ale w złym miejscu na niej. Do długich symulacji wybiera się metody symplektyczne: kształt i energia orbity nie dryfują, a położenie na orbicie i tak nie jest przewidywalne na miliony lat. Do krótkich, bardzo dokładnych rachunków lepszy jest RK4 lub metody adaptacyjne.

Zadania · część 4

1. (a) $f = x^3 - 10$: 2,16667; 2,15450; 2,1544347 (dokładnie 2,15443469...). (b) $f = x - \cos x$, $f' = 1 + \sin x$: 0,750364; 0,739113; 0,7390851.
2. $E_{n+1} = E_n - \frac{E_n - 0,9 \sin E_n - 2}{1 - 0,9 \cos E_n}$: 2,59538; 2,52310; 2,522366; 2,5223654340. Program `newton.py` kończy po 5 krokach.
3. $x_1 = 0 - \frac{2}{-2} = 1$, $x_2 = 1 - \frac{1}{1} = 0$: cykl $0 \rightarrow 1 \rightarrow 0$ bez końca. Jedyny pierwiastek rzeczywisty to $x \approx -1,769$; start np. z $x_0 = -2$ szybko do niego zbiega.
4. Krok: $x_{n+1} = x_n - (1 + x_n^2) \arctg x_n$; warunek $x_{n+1} = -x_n$ daje podane równanie, $x^* \approx 1,3917$. Dla $|x_0| < x^*$ przybliżenia skaczą na przemian po obu stronach zera z malejącą amplitudą i w końcu zbiegają (z 1,39 powoli, z 1 już po 4 krokach). Dla $|x_0| > x^*$ amplituda rośnie: z 1,5 po pięciu krokach -1575 .

Zadania · część 5

1. Przykładowe rozwiązanie poniżej. Przy 10 okresach i 10^3 , 10^4 , 10^5 krokach na okres błąd położenia maleje: Verlet $1,77 \cdot 10^{-2} \rightarrow 1,77 \cdot 10^{-4} \rightarrow 1,77 \cdot 10^{-6}$ (rzęd 2); RK4 $5,3 \cdot 10^{-7} \rightarrow 2,9 \cdot 10^{-11}$ (rzęd ok. 4, dalej przeszkadza zaokrąglanie). Euler wchodzi w zakres rzędu 1 dopiero od ok. 10^6 kroków na okres: błąd $1,5 \cdot 10^{-1}$ przy 10^6 i $1,5 \cdot 10^{-2}$ przy 10^7 (wcześniej, przy 10^4 i 10^5 , błąd jest rzędu 1 i rzędu metody nie widać).

```

// Orbita Keplera (GM = 1): Euler, RK4 i Verlet w C++, z pomiarem czasu.
// Kompilacja: g++ -O2 -std=c++17 orbita.cpp -o orbita
// Uruchomienie: ./orbita [metoda] [kroki_na_okres] [okresy] [mimosrod]
#include <array>
#include <chrono>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <string>

using State = std::array<double, 4>; // x, y, vx, vy

State deriv(const State& s) {
    double r2 = s[0] * s[0] + s[1] * s[1], r3 = r2 * std::sqrt(r2);
    return {s[2], s[3], -s[0] / r3, -s[1] / r3};
}

State euler(const State& s, double h) {
    State d = deriv(s), out;
    for (int i = 0; i < 4; ++i) out[i] = s[i] + h * d[i];
    return out;
}

State rk4(const State& s, double h) {
    auto add = [](const State& a, const State& b, double c) {
        State r;
        for (int i = 0; i < 4; ++i) r[i] = a[i] + c * b[i];
        return r;
    };
    State k1 = deriv(s);
    State k2 = deriv(add(s, k1, h / 2));
    State k3 = deriv(add(s, k2, h / 2));
    State k4 = deriv(add(s, k3, h));
    State out;
    for (int i = 0; i < 4; ++i) out[i] = s[i] + h / 6 * (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]);
    return out;
}

double energy(const State& s) {
    return 0.5 * (s[2] * s[2] + s[3] * s[3]) - 1 / std::hypot(s[0], s[1]);
}

int main(int argc, char** argv) {
    std::string name = argc > 1 ? argv[1] : "rk4";
    long n_per = argc > 2 ? std::atol(argv[2]) : 1000;
    long periods = argc > 3 ? std::atol(argv[3]) : 10;
    double e = argc > 4 ? std::atof(argv[4]) : 0.5;
    if (name != "euler" && name != "rk4" && name != "verlet") {
        std::fprintf(stderr, "nieznana metoda: %s (euler, rk4, verlet)\n", name.c_str());
        return 1;
    }
    // start w peryhelium orbity o mimośrodzie e i półosi a = 1 (okres 2π)
    State s = {1 - e, 0.0, 0.0, std::sqrt((1 + e) / (1 - e))};
    const double pi = std::acos(-1.0);
    double h = 2 * pi / n_per, E0 = energy(s);
    auto t0 = std::chrono::steady_clock::now();
    long n = n_per * periods;
    if (name == "verlet") {
        State d = deriv(s);
        for (long i = 0; i < n; ++i) {
            s[2] += h / 2 * d[2]; s[3] += h / 2 * d[3];

```

```

        s[0] += h * s[2];    s[1] += h * s[3];
        d = deriv(s);        // jedno nowe obliczenie siły na krok
        s[2] += h / 2 * d[2]; s[3] += h / 2 * d[3];
    }
} else if (name == "euler") {
    for (long i = 0; i < n; ++i) s = euler(s, h);
} else {
    for (long i = 0; i < n; ++i) s = rk4(s, h);
}
double t = std::chrono::duration<double>(std::chrono::steady_clock::now() - t0).count();
std::printf("%s: %ld kroków, błąd położenia %.3e, zmiana energii %.3e, czas %.3f s\n",
            name.c_str(), n, std::hypot(s[0] - (1 - e), s[1]), energy(s) - E0, t);
}

```

Matematyka · Metody numeryczne: Python i C++. Źródła: W. H. Press i in., *Numerical Recipes*, wyd. 3 (Cambridge University Press 2007), rozdz. 9, 17; E. Hairer, C. Lubich, G. Wanner, *Geometric Numerical Integration*, wyd. 2 (Springer 2006), rozdz. I. Wszystkie programy uruchomione, a liczby policzone na nowo; czasy zmierzone na jednym komputerze.