Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium:

Rozwiązywanie równań różniczkowych – Metody: Eulera, Heuna i Rungego-Kutty

Przedmiot: Metody Numeryczne

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący przedmiot: dr hab. inż. Marcin Hojny

Data: 29 maj 2024

Numer lekcji: 11

Grupa laboratoryjna: 4

Wstęp teoretyczny

Metody Eulera, Heuna oraz Rungego-Kutty są podstawowymi, powszechnie używanymi metodami do rozwiązywania równań różniczkowych zwyczajnych.

Podstawy Matematyczne

Metoda służą do przybliżonego rozwiązywania równań różniczkowych zwyczajnych postaci:

$$\frac{dy}{dx} = f(x, y) \tag{1}$$

z zadanym warunkiem początkowym:

$$y(x_0) = y_0 \tag{2}$$

Metoda Eulera

Metoda Eulera jest najprostszą metodą numeryczną. Charakteryzuje się niskim kosztem obliczeniowym, ale jej główną wadą jest niska dokładność i stabilność, szczególnie przy większych krokach całkowania.

Główna idea metody Eulera polega na przybliżeniu rozwiązania w kolejnych punktach za pomocą stycznych do krzywej rozwiązania. Przybliżamy wartości funkcji krok po kroku, stosując liniową aproksymację:

$$y_{n+1} = y_n + h * f(x_n, y_n) \tag{3}$$

gdzie:

- y_n jest przybliżoną wartością funkcji w punkcie x_n,
- h jest krokiem czasowym (przyrostem x),
- $f(x_n, y_n)$to wartość pochodnej funkcji w punkcie (x_n, y_n).

Metoda Heuna

Metoda Heuna, znana również jako ulepszona metoda Eulera, rozwija podstawową koncepcję metody Eulera przez dodanie kroku korygującego. Początkowo, podobnie jak w metodzie Eulera, stosuje liniową aproksymację do przewidzenia tymczasowego wartości funkcji, a następnie używa średniej wartości stycznych na początku i na końcu przedziału krokowego do ostatecznego przybliżenia wartości funkcji.

Ta metoda dąży do zwiększenia dokładności przez zmniejszenie błędu lokalnego

$$y_{n+1} = y_n + \frac{h}{2} * [f(x_n, y_n) + f\big(x_n + h, y_n + h * f(x_n, y_n)\big)] \quad (4)$$

gdzie:

- y_n jest przybliżoną wartością funkcji w punkcie x_n,
- h jest krokiem czasowym (przyrostem x),

Metoda Rungego-Kutty

Metoda Rungego-Kutty jest bardziej zaawansowaną techniką, która polega na wykorzystaniu kilku ocen nachylenia w jednym kroku całkowania, aby uzyskać bardziej dokładne przybliżenie funkcji.

W metodzie tej, obliczane są cztery różne przybliżenia stycznych (k1, k2, k3, k4), które są następnie ważone i kombinowane, aby osiągnąć końcową wartość funkcji w kolejnym punkcie.

$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \quad (5)$$

gdzie:

- $k_1 = h * f(x_n, y_n)$
- $k_2 = h * f(x_n + 0.5 * h, y_n + 0.5 * k_1)$
- $k_3 = h * f(x_n + 0.5 * h, y_n + 0.5 * k_2)$
- $k_4 = h * f(x_n + h, y_n + k_3)$

We wszystkich powyższych metodach nowy x obliczamy korzystając z poniższego wzoru.

$$x_{n+1} = x_n + h \tag{6}$$

Kroki Algorytmu

Przebieg algorytmu będzie wyglądał dokładnie tak samo dla każdej z powyższych metod. Jedyną różnica jest użyty wzór.

1. **Inicjalizacja**: Ustalenie początkowych wartości x_0 i y_0.
2. **Iteracja**: Dla każdego kroku n:
    - Obliczanie pochodnej $f(x_n, y_n)$
    - Obliczanie nowej wartość y przy użyciu wzoru 3, 4 lub 5.
    - Przesuwanie się do nowego punktu wzorem 6.
3. **Powtarzanie**: Powtarzanie kroku iteracji, aż osiągnie się żądany punkt końcowy lub liczbę kroków.

Implementacja

Metody zostały zaimplementowane w języku C++, w postaci funkcji `solve_differential_equation`, która wykonuje wcześniej opisane kroki algorytmu.

Funkcja przyjmuje następujące argumenty:

- Startowy x: `double x0`.
- Wartość początkowa y: `double y0`.
- Krok czasowy: `double h`.
- Czas końcowy: `double xn`.
- Wskaźnik do pochodnej: `derivative`.
- Wskaźnik do funkcji reprezentującej wzór wykorzystywany przez daną metodę `formula`.

Typem zwracanym jest `std::vector<double>`, tablica wyników, y-ków.

Funkcje pomocnicze

Wzory metod zostały zapisane jako proste funkcje. Ich definicję znajdują się poniżej. Funkcje te implementują kolejno wzory 3, 4 i 5.

```
double euler_formula(Derivative f, double h, double x, double y)
{
	return h * f(x, y);
}

double heun_formula(Derivative f, double h, double x, double y)
{

	return (h / 2) * (f(x, y) + f(x + h, y + h * f(x, y)));
}

double rungy_kutty_formula(Derivative f, double h, double x,
double y)
{
	double k1 = h * f(x, y);
	double k2 = h * f(x + 0.5 * h, y + 0.5 * k1);
	double k3 = h * f(x + 0.5 * h, y + 0.5 * k2);
	double k4 = h * f(x + h, y + k3);

	return (k1 + 2 * k2 + 2 * k3 + k4) / 6;
}
```

Dodatkowo, jako argument `derivative` podawana jest prosta funkcja reprezentująca pochodną.

```
double function(double x, double y)
{
	return y;
}
```

Pełna definicja funkcji `solve_differential_equation`

```
std::vector<double> solve_differential_equation(double x0, double y0,
double h, double xn, Derivative derivative, Formula formula)
{
	// Result array initialization
	std::vector<double> results;

	// Adding the first known result to the vector
	results.push_back(y0);

	// Assigning x0 as a start point
	double y = y0;
	double x = x0;

	while (x < xn)
	{
		// Get new x and y
		y += formula(derivative, h, x, y);
		x += h;

		// Save the result
		results.push_back(y);
	}

	return results;
}
```

Omówienie elementów funkcji `solve_differential_equation`

W pierwszym kroku funkcja inicjalizuje wektor, który będzie przechowywać wyniki.

```
std::vector<double> results;
```

W kolejnym kroku, pierwszy znany y-grek, który został przekazany funkcji w postaci argumentu zostaje dodany do listy wyników.

```
results.push_back(y0);
```

Zmienne **y** i **x** są inicjalizowane jako kolejno, znane rozwiązanie oraz wartość x, dla jakiej to rozwiązanie zostało osiągnięte.

```
double y = y0;
double x = x0;
```

Pętla `while` będzie wykonywać swoją pracę do póki nie dojdziemy do końca zakresu, w którym przeprowadzane mają być obliczenia.

```
while (x < xn)
```

Obliczenie zmiennej **y** w pętli następuje z wykorzystaniem przekazanego funkcji wzoru, tj. wzór 3, 4 lub 5. W taki sposób jesteśmy w stanie łatwo zmienić stosowaną metodę.

```
y += formula(derivative, h, x, y);
```

Zmienną **x** aktualizujemy z użyciem wzoru 6.

```
x += h;
```

Uzyskiwane wyniki są kolejno dopisywane do wektora.

```
results.push_back(y);
```

Po wyjściu z pętli zwracamy wektor.

```
return results;
```

Testy na wybranych przykładach

Zaimplementowane metody zostały przetestowane pod kątem jakości wyników oraz czasu trwania obliczeń. Wyniki zostały porównane z wynikami otrzymanymi przez Solve_IVP z biblioteki scipy.integrate języka Python.

Wyniki czasowe są średnią wynikająca z 10000 testów przeprowadzonych na każdej metodzie.

Test 1: Równanie liniowe

Dane:

- Funkcja: $f(x, y) = 3\mathrm{x} + 4\mathrm{y}$
- Warunki początkowe: $x0 = 0, y0 = 1$
- Krok h: 0.1
- Zakres x: od 0 do 0.4

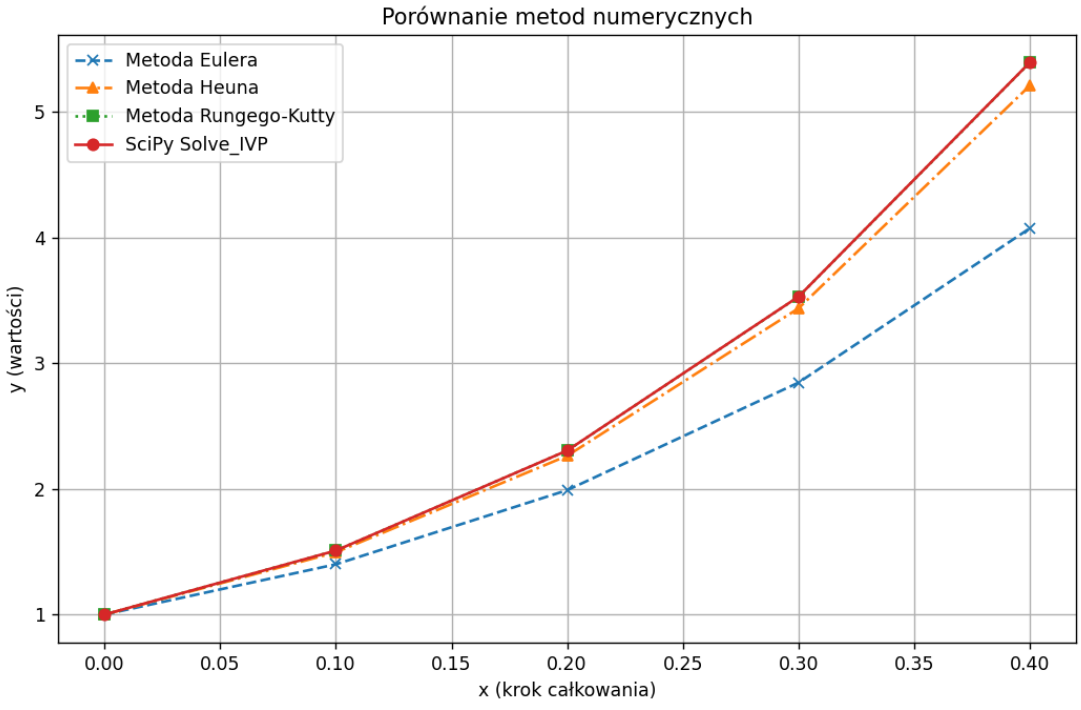
Wyniki

- Solve_IVP:
    - Rozwiązania: 1, 1.50912, 2.30517, 3.53021, 5.39434

- `euler_formula`
    - Rozwiązania: 1, 1.4, 1.99, 2.846, 4.0744
    - Czas (nanosekundy): 2833.65

- `heun_formula`
    - Rozwiązania: 1, 1.495, 2.2636, 3.43713, 5.20995
    - Czas (nanosekundy): 2849.35

- `rungy_kutty_formula`
    - Rozwiązania: 1, 1.50893, 2.30501, 3.52941, 5.39279
    - Czas(nanosekundy): 2875.52



Wykres 1: Wyniki dla równania liniowego

Test 2: Równanie oscylatora harmonicznego

Dane:

- Funkcja: $f(x, y) = -\mathrm{y}$
- Warunki początkowe: $x0 = 0$, $y0 = 1$
- Krok h: 0.05
- Zakres x: od 0 do 2.0

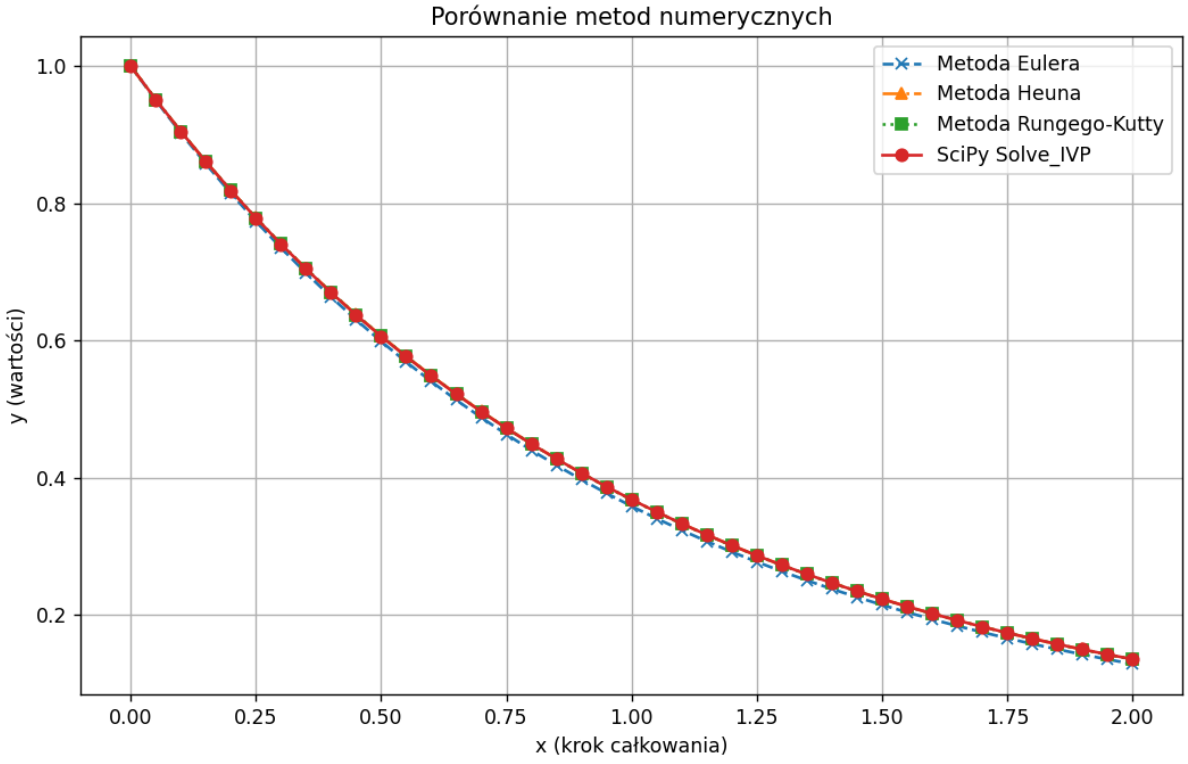
Wyniki (pierwsze 4)

- Solve_IVP:
  - Rozwiązania: 1, 1.50912, 2.30517, 3.53021, 5.39434

- `euler_formula`
  - Rozwiązania: 1, 0.95, 0.9025, 0.857375, 0.814506
  - Czas (nanosekundy): 7818.02

- `heun_formula`
  - Rozwiązania: 1, 0.95125, 0.904877, 0.860764, 0.818802
  - Czas (nanosekundy): 8106.23

- `rungy_kutty_formula`
  - Rozwiązania: 1, 0.951229, 0.904837, 0.860708, 0.818731
  - Czas(nanosekundy): 8674.28



Wykres 2: Wyniki dla oscylatora harmonicznego

Test 3: Równanie eksponencjalne

Dane:

- Funkcja: $f(x, y) = \mathrm{y} * \log(\mathrm{y})$
- Warunki początkowe: $x0 = 1, y0 = e$
- Krok h: 0.1
- Zakres x: od 1 do 9

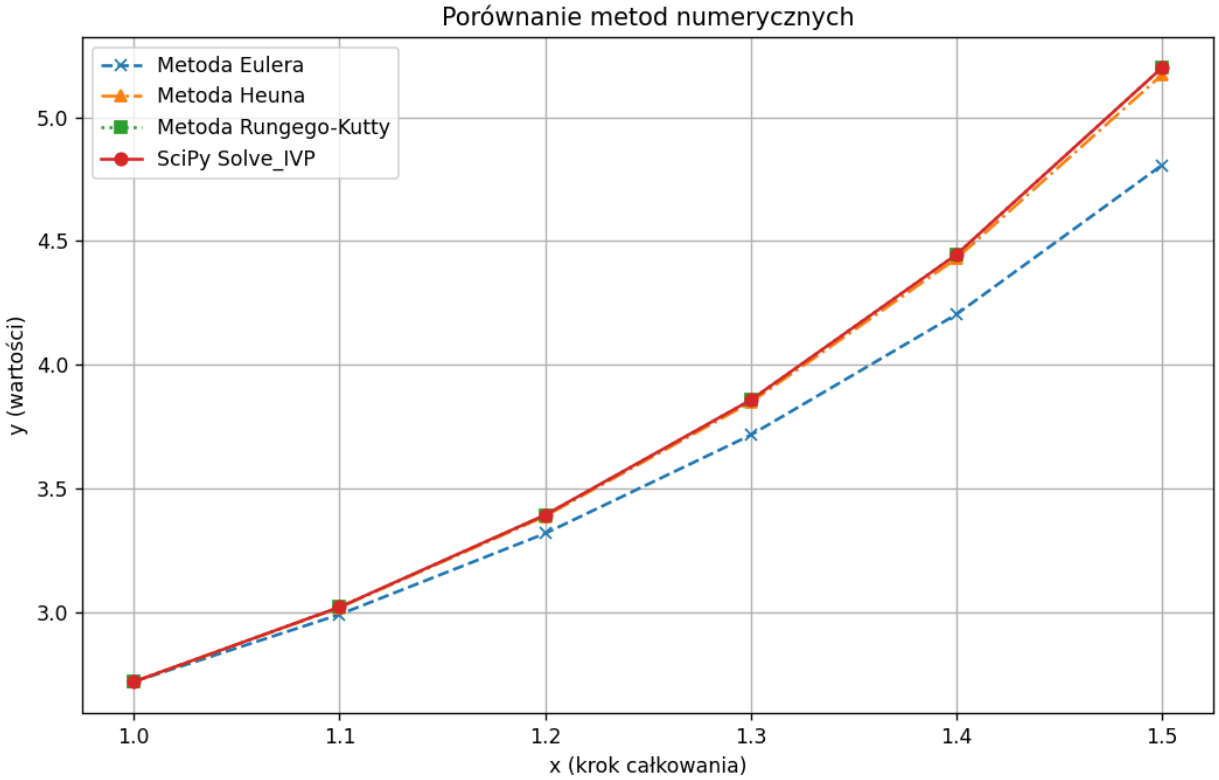
Wyniki (pierwsze 6)

- Solve_IVP:
  - Rozwiązania: 2.71828, 3.01976, 3.39246, 3.85698, 4.44481, 5.20033

- `euler_formula`
  - Rozwiązania: 2.71828, 2.99011, 3.31762, 3.71548, 4.20315, 4.80665
  - Czas (nanosekundy): 10858.4

- `heun_formula`
  - Rozwiązania: 2.71828, 3.01795, 3.38727, 3.84761, 4.42859, 5.17192
  - Czas (nanosekundy): 12319.5

- `rungy_kutty_formula`
  - Rozwiązania: 2.71828, 3.01974, 3.39193, 3.85686, 4.44517, 5.20027
  - Czas(nanosekundy): 15199.6



Wykres 3: Wyniki dla równania eksponencjalnego (6 pierwszych wynikow)

Test 4: Równanie sinusoidalne

Dane:

- Funkcja: $f(x, y) = \sin(\mathrm{x}) - \cos(\mathrm{y})$
- Warunki początkowe: $x0 = 0, y0 = 0$
- Krok h: 0.2
- Zakres x: od 0 do 25.6

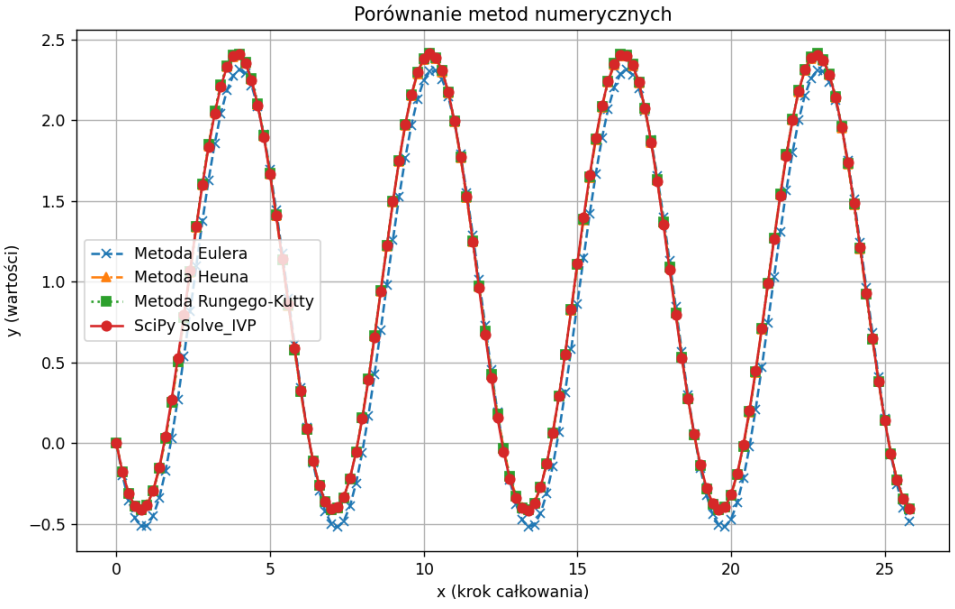
Wyniki (Pierwsze 8**)**

- Solve_IVP:
  - Rozwiązania: 0.00000, -0.17874, -0.31051, -0.38999, -0.41406, -0.38177, -0.29353, -0.14860

- `euler_formula`
  - Rozwiązania: 0, -0.2, -0.356279, -0.462608, -0.514747, -0.510617, -0.450383, -0.336447
  - Czas (nanosekundy): 15396.2

- `heun_formula`
  - Rozwiązania: 0, -0.17814, -0.309444, -0.388677, -0.412682, -0.3805, -0.293415, -0.154898
  - Czas (nanosekundy): 19338.2

- `rungy_kutty_formula`
  - Rozwiązania: 0, -0.178736, -0.31048, -0.389978, -0.414063, -0.381774, -0.294397, -0.155417
  - Czas(nanosekundy): 21785.8



Wykres 4: Wyniki dla równania sinusoidalnego

Opracowanie wyników

Analiza Dokładności

Równanie Liniowe ($f(x, y) = 3x + 4y$):

- Metoda Rungego-Kutty wykazała się porównywalną dokładnością do Solve_IVP.
- Metody Heuna i Eulera (zwłaszcza Eulera) były mniej dokładne, z widocznymi odchyleniami w obliczeniach.

Równanie Oscylatora Harmonicznego ($f(x, y) = -y$):

- Wszystkie metody numeryczne dobrze odwzorowały kształt oscylacji, jednak metoda Rungego-Kutty była najbliższa wynikom Solve_IVP.
- Metoda Eulera i Heuna pokazały większe, aczkolwiek nie tak znaczne rozbieżności, zwłaszcza w dłuższym przedziale czasowym.

Równanie Eksponencjalne ($f(x, y) = y * log(y)$):

- Wyniki Solve_IVP i metody Rungego-Kutty były bardzo bliskie, co wskazuje na wysoką dokładność tej metody.
- Metoda Heuna była względnie blisko poprawnych rozwiązań.
- Metody Eulera ponownie wykazał większe błędy w obliczeniach, szczególnie dla większych wartości x.

Równanie Sinusoidalne ($f(x, y) = sin(x) - cos(y)$):

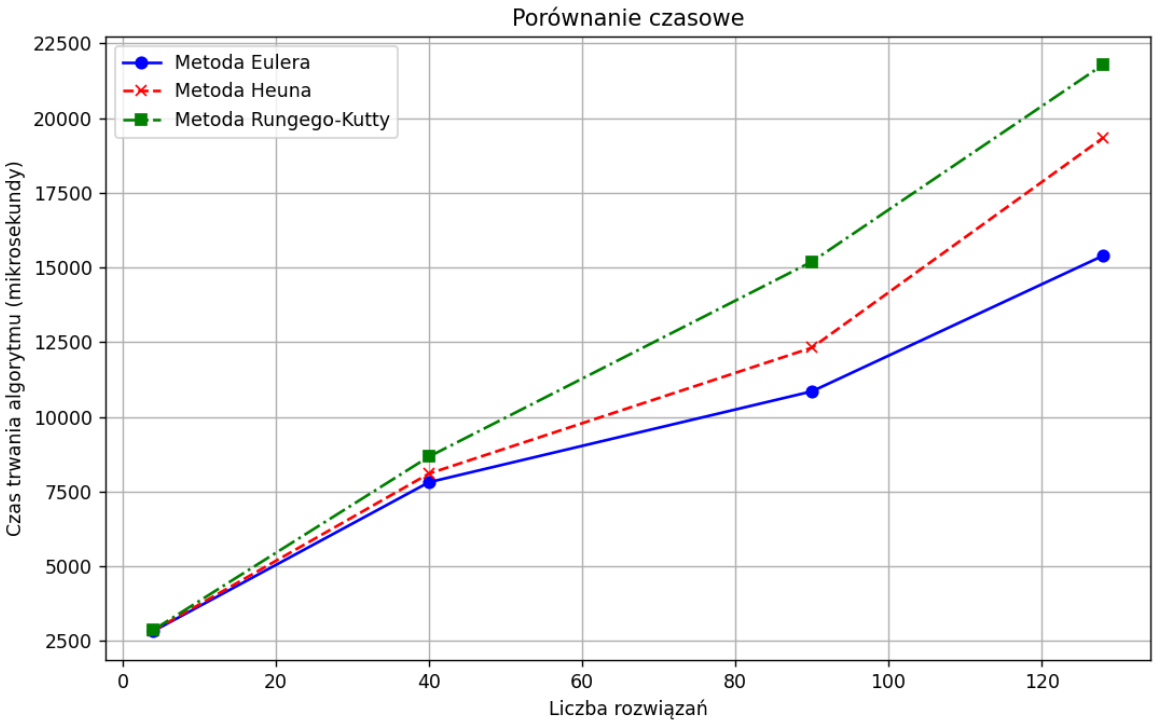
- Metoda Rungego-Kutty była najdokładniejsze, z niewielkimi różnicami między wynikami z Solve_IVP.
- Metoda Heuna była bliska, lecz z nieco większymi odchyleniami, szczególnie w skrajnych punktach testu.
- Metoda Eulera wykazała największe różnice w stosunku do oczekiwanych wyników.

Analiza Czasu Obliczeń

Czas trwania obliczeń został zmierzony w nanosekundach, a wyniki są średnimi z 10 000 testów dla każdej metody.

Czas pracy				
Test 1	Test 2	Test 3	Test 4	Metoda
2833.65	7818.02	10858.4	15396.2	Eulera
2849.35	8106.23	12319.5	19338.2	Heuna
2875.52	8674.28	15199.6	21785.8	Rungego-Kutty

Tabela 1: Czas pracy metod



Wykres 5: Czas pracy metod a liczba rozwiązań

Metoda Rungego-Kutty, choć najdokładniejsza, okazała się być również najbardziej czasochłonna w każdym z testów. Metoda Eulera była najmniej dokładna, ale również najszybsza, co może być korzystne w aplikacjach wymagających szybkiego czasu odpowiedzi kosztem precyzji.

Wnioski

Testy wyraźnie pokazują, że metoda Rungego-Kutty jest najlepsza pod względem dokładności wyników, lecz wymaga również najwięcej czasu na obliczenia. Metoda Eulera, mimo że najszybsza, nie oferuje porównywalnej dokładności i jest zalecana tylko w sytuacjach, gdzie czas jest bardziej krytyczny niż precyzja. Metoda Heuna znajduje się gdzieś pomiędzy tymi dwoma pod względem szybkości obliczeń, jednak pod względem dokładności nie odbiega aż tak bardzo od metody Rungego-Kutty, sugerując, że spośród tych trzech metod mogłaby ona być tą najbardziej uniwersalną.