środa, 16 listopada 2022

Ogólne uwagi

Informacje o programie.

Implementacja

Wszystkie metody implementują klasę ISolver. Gwarantuje one podstawowe metody odpowiedzialne za wyświetlanie danych i ich obliczanie. Realną różnice stanowi tak naprawdę implementacja metody ‚_method(self, i, results, linspace)' odpowiedzialnej za obliczanie x[i+1] argumentu.

Zaimplementowane zostały 4 metody:
- Metoda Eulera
- Metoda Midpoint
- Metoda Rungego-Kutta rzędu 4
- Metoda Taylora rzędu n (w przykładzie n = 3)

Kod dostępny jest także na github - link:

https://github.com/psp515/DifferentalEquations

Za optymalne rozwiązanie przyjęte zostało rozwiazanie funkcji oderint z biblioteki scipy.integrate.

Kod klasy ISolver

```python
class ISolver:
    '''
    Interface created for implementing in numeric methods that calculates 2 parameter functions.
    Class should be representing a mathematical function, that can take different parameters to get required data.
    Displays last calculated data.
    '''
    Łukasz Kolber *
    def __init__(self, functions, name, chart_color):
        '''
        Function initializes solve method and checks if parameters are valid for calculation and displaying.
        :param functions: 2 parameter function representing equations (Could be 2 parameter functions but you need to create own __init__).
        :param name: Name to be displayed.
        :param chart_color: Default color on chart.
        '''
        if not isinstance(functions, FunctionType):
            raise InvalidArgumentInstanceException("Functions should be single function.")

        inspection = getfullargspec(functions)

        if len(inspection[0]) != 2:
            raise InvalidArgumentInstanceException("Functions should be function having 2 parameters.")

        if not isinstance(chart_color, str):
            raise InvalidArgumentInstanceException("Chart color should be string.")

        self.chart_color = chart_color
        self.functions = functions
        self.name = name

        self._solution = None
        self._optimal_solution = None
```

```
#region Properites
```

Łukasz Kolber

Łukasz Kolber

```
@property
def solution(self):
    return self._solution
```

Łukasz Kolber

```
@property
def optimal_solution(self):
    return self._optimal_solution

#endregion
```

```
def solve(self, x0, a, b, n):
    '''
    Method solves equation in range a, b divided in n sectors, with starting value x0.
    :param x0: Starting value
    :param a: Compartment start.
    :param b: Compartment end.
    :param n: Number of intervals.
    :return: Solution class with results.
    '''

    if self.solution != None and self.solution.compare(x0, a, b, n, self.functions):
        return self.solution

    linspace = np.linspace(a, b, n)
    results = np.zeros(n)
    results[0] = x0

    for i in range(n-1):
        results[i+1] = self._method(i, results, linspace)

    self._solution = Solution(linspace, results, self.name, self.functions, self.chart_color)
    self._optimal_solution = self._get_optimal_solution(x0, a, b, n)

    return self.solution

def draw_plot(self, draw_optimal_solution=False):
    '''
    Method draws plot of the results.
    :param draw_optimal_solution: Indicates drawing optimal solution calculated with NumpySolver.
    '''

    if self.solution == None:
        print('Please solve solution first.')
        return

    plt.plot(self.solution.space,
             self.solution.results,
             self.sol                    lor,
             label=f'                    .name}')

    if draw_optimal_solution:
        plt.plot(self.optimal_solution.space,
                 self.optimal_solution.results,
                 self.optimal_solution.chart_color,
                 label=f'{self.optimal_solution.name}')

    plt.legend(loc='best')
    plt.xlabel('t')
    plt.grid()
    plt.show()
```

@final
class bool(int)
bool(x) -> bool

```python
def draw_global_error(self, round_digits=5):
    '''
    Function draws the biggest difference between optimal and calculated solution
    :param: Rounds number to specified number of digits
    '''

    if round_digits < 0:
        round_digits = 0

    if self.solution == None:
        print('Please solve solution first.')
        return

    print(f'Name: {self.name}')
    print(f'Global Error: {round(self._get_global_error(), round_digits)}')
```

Łukasz Kolber

```python
def draw_frame(self):
    '''
    Function draws data frame with comparatin between method solution and optimal solution
    '''
    if self.solution == None:
        print('Please solve solution first.')
        return

    array = {'t': self.solution.space,
             'Optimal': self.optimal_solution.results,
             f'{self.name}': self.solution.results}
    tab = pd.DataFrame(array)
    display(tab)
```

```python
def _method(self, i, results, linspace):
    '''
    Method for calculating i+1 index based on i-th index result.
    :param i: Index of last calculated value.
    :param results: List of results.
    :param linspace: Space of function arguments.
    :return: Result of calculations.
    '''
    raise UnimplementedException("Method must be implemented in order to use it.")
```

Łukasz Kolber

```python
def _get_global_error(self):
    """
    :return: maximal difference between optimal and calculated solution
    """
    return max([abs(self.solution.results[i]-self.optimal_solution.results[i]) for i in range(len(self.solution.results))])
```

Łukasz Kolber

```python
def _get_optimal_solution(self, x0, a, b, n):
    '''
    Returns solution of calculations with use of odeint.
    :param x0: Starting value
    :param a: Compartment start.
    :param b: Compartment end.
    :param n: Number of intervals.
    :return: Solution class with results.
    '''
    from SolvingMethods.OdeintSolver import OdeintSolver
    if type(self.functions) == list:
        return OdeintSolver(self.functions[0]).solve(x0, a, b, n)

    return OdeintSolver(self.functions).solve(x0, a, b, n)
```

Użyte funkcje, ich pochodne i parametry

Każda metoda została podzielona na 2 podpunkty **A** oraz **B** w obu została użyta 1 funkcja **F** oraz jej pochodne, podpunkty różnią się jedynie argumentami wywołania.

```
F: -2*x + exp(t)
F1: 4*x - exp(t)
F2: -8*x + 3*exp(t)
```

Funkcje F1 oraz F2 zostały obliczone z pomocą następującego kodu

```python
from sympy import *

x = Symbol('x')
t = Symbol('t')

def f(t, x):
    return -2*x + exp(t)

def f_t(t,x):
    return diff(f(t,x), t)
def f_x(t,x):
    return diff(f(t,x), x)
def f_tx(t,x):
    return diff(f(t,x), t, 1, x, 1)
def f_tt(t,x):
    return diff(f(t,x), t, 2)
def f_xx(t,x):
    return diff(f(t,x), x, 2)
def f1(t,x):
    return f_t(t,x)+f(t,x)*f_x(t,x)

def f2(t, x):
    return f_tt(t,x)+2*f(t,x)*f_tx(t,x)+f_t(t,x)*f_x(t,x)+f(t,x)*(f_x(t,x))**2+(f(t,x))**2*f_xx(t,x)

if __name__ == '__main__':
    print('Program Start')
    print("F:", f(t,x))
    print("F1:", f1(t, x))
    print("F2:", f2(t, x))
```

A
We wszystkich przykładach do metody ‚solve(…)' zostały podane następujące argumenty:

x0 = 1 (wartość początkowa)

a = 0 (początek przedziału)

b = 1 (koniec przedziału)

n = 10 (liczba podziałów)

B
We wszystkich przykładach do metody ‚solve(…)' zostały podane następujące argumenty:

x0 = 1 (wartość początkowa)

a = 0 (początek przedziału)

b = 1 (koniec przedziału)

n = 40 (liczba podziałów)

Wywołanie kodu

Funkcje użyte w ćwiczeniu:

```
def f(x,t):
    return -2*x + np.exp(t)
new *
def f1(x,t):
    return 4*x - np.exp(t)
new *
def f2(x,t):
    return -8*x + 3 * np.exp(t)
```

Kod dzięki któremu uzyskano wyniki:

```python
if __name__ == '__main__':
    print('Program starts')
    print('Initializing data...')

    init, a, b, n = 1, 0, 1, 10
    solutions = []
    methods = [EulerSolver(f, chart_color='red'),
               MidpointSolver(f, chart_color='green'),
               RungeKutt4Solver(f, chart_color='yellow'),
               TaylorSolver([f, f1, f2], chart_color='blue')]

    print('Calculating data...')

    for method in methods:
        method.solve(init, a, b, n)
        method.draw_plot(True)
        method.draw_frame()
        method.draw_global_error()

    print('Initializing data...')

    init, a, b, n = 1, 0, 1, 40

    print('Calculating data...')

    for method in methods:
        method.solve(init, a, b, n)
        method.draw_plot(True)
        method.draw_frame()
        method.draw_global_error()

    print('Program Finished')
```

Metoda Eulera

Podstawowa metoda numeryczna.

Wygląd metody obliczającej x[i+1] element (wymagana do implementacji przez klasę ISolver).

```
def _method(self, i, results, linspace):
    return results[i] + (linspace[i+1] - linspace[i]) * self.functions(results[i], linspace[i])
```

A

Wykres porównujący wartość oczekiwaną i uzyskaną

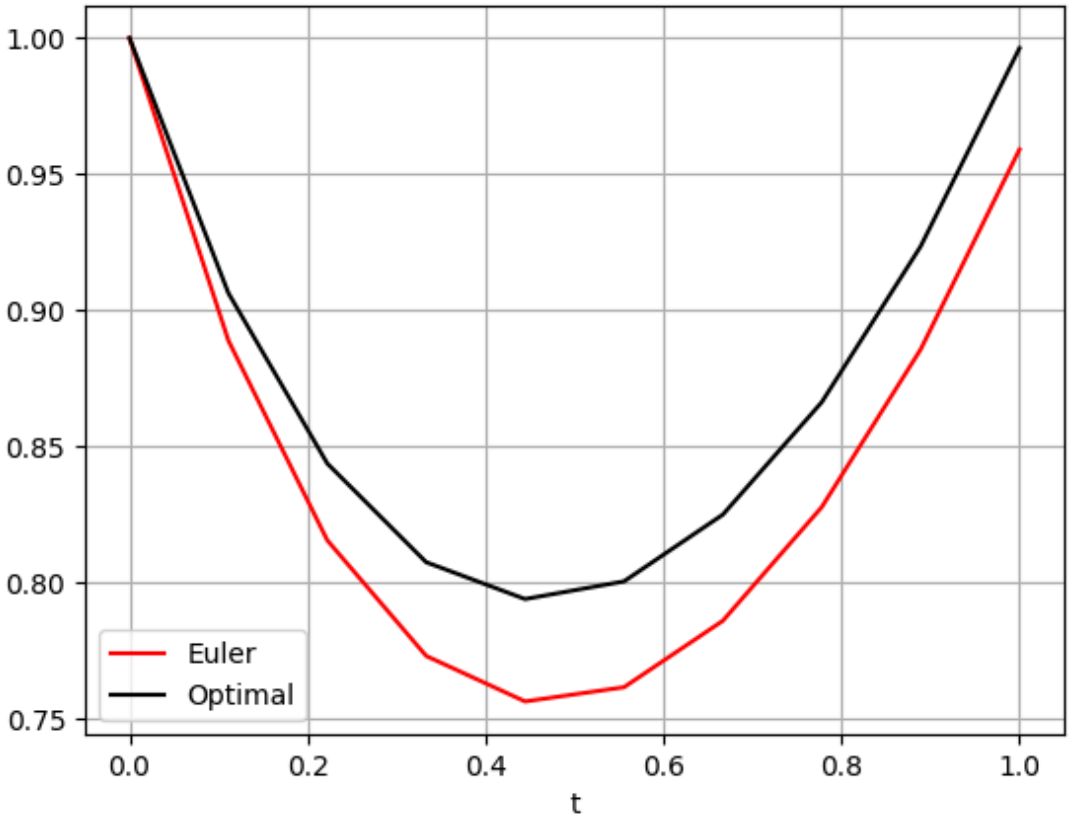


Tabela obliczonych wartości

	t	Optimal	Euler
0	0.000000	1.000000	1.000000
1	0.111111	0.906331	0.888889
2	0.222222	0.843737	0.815527
3	0.333333	0.807482	0.773060
4	0.444444	0.793949	0.756337
5	0.555556	0.800432	0.761553
6	0.666667	0.824976	0.785976
7	0.777778	0.866258	0.827729
8	0.888889	0.923484	0.885637
9	1.000000	0.996317	0.959099

Błąd Globalny

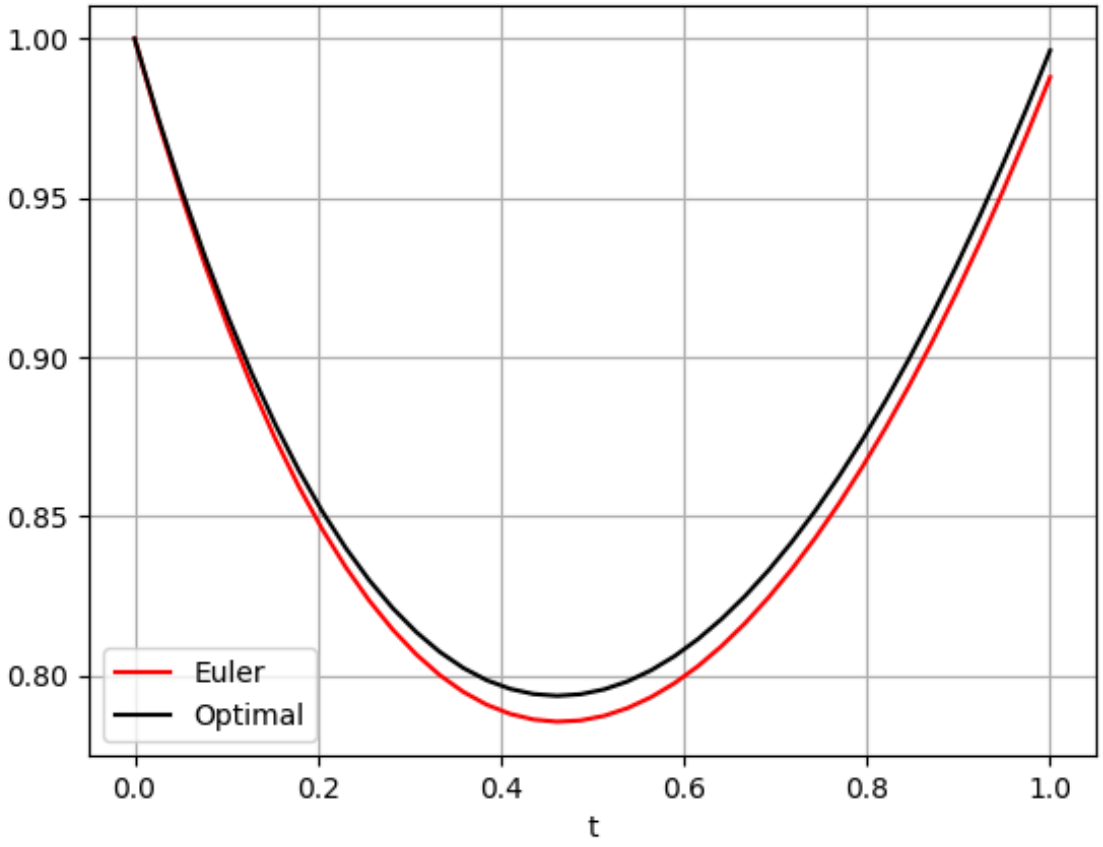
```
Name: Euler
Global Error: 0.039
```

Wnioski

Wywołanie metody Eulera dla małego n dało słabe rezultaty, błąd globalny jest dość duży. Zwiększenie n powinno dać znacznie lepsze rezultaty.

B

Wykres porównujący wartość oczekiwaną i uzyskaną



Błąd Globalny

```
Name: Euler
Global Error: 0.00854
```

Tabela obliczonych wartości

	t	Optimal	Euler
0	0.000000	1.000000	1.000000
1	0.025641	0.975331	0.974359
2	0.051282	0.952553	0.950699
3	0.076923	0.931589	0.928935
4	0.102564	0.912365	0.908989
5	0.128205	0.894812	0.890785
6	0.153846	0.878865	0.874252
7	0.179487	0.864463	0.859324
8	0.205128	0.851548	0.845938
9	0.230769	0.840065	0.834035
10	0.256410	0.829964	0.823561
11	0.282051	0.821196	0.814463
12	0.307692	0.813716	0.806691
13	0.333333	0.807482	0.800202
14	0.358974	0.802455	0.794951
15	0.384615	0.798596	0.790898
16	0.410256	0.795872	0.788007
17	0.435897	0.794250	0.786243
18	0.461538	0.793701	0.785573
19	0.487179	0.794196	0.785967
20	0.512821	0.795709	0.787397
21	0.538462	0.798218	0.789838
22	0.564103	0.801699	0.793266
23	0.589744	0.806133	0.797659
24	0.615385	0.811501	0.802998
25	0.641026	0.817787	0.809264
26	0.666667	0.824976	0.816441
27	0.692308	0.833054	0.824514
28	0.717949	0.842009	0.833470
29	0.743590	0.851830	0.843298
30	0.769231	0.862509	0.853987
31	0.794872	0.874038	0.865529
32	0.820513	0.886409	0.877916
33	0.846154	0.899618	0.891143
34	0.871795	0.913661	0.905204
35	0.897436	0.928534	0.920096
36	0.923077	0.944236	0.935817
37	0.948718	0.960767	0.952365
38	0.974359	0.978127	0.969741
39	1.000000	0.996317	0.987946

Wnioski

Zwiększenie n, poprawiło dokładność kilkukrotnie, wnioskiem można powiedzieć jest że używanie metody Eulera dla bardzo dużego n da nam realne rezultaty.

Metoda Midpoint

Modyfikacja metody Eulera.

Wygląd metody obliczającej x[i+1] element (wymagana do implementacji przez klasę ISolver).

```
def _method(self, i, results, linspace):
    diff = linspace[i + 1] - linspace[i]
    first = self.functions(results[i], linspace[i])
    second = self.functions(results[i] + first * diff / 2., linspace[i] + diff / 2.)
    return results[i] + second * diff
```

A

Wykres porównujący wartość oczekiwaną i uzyskaną

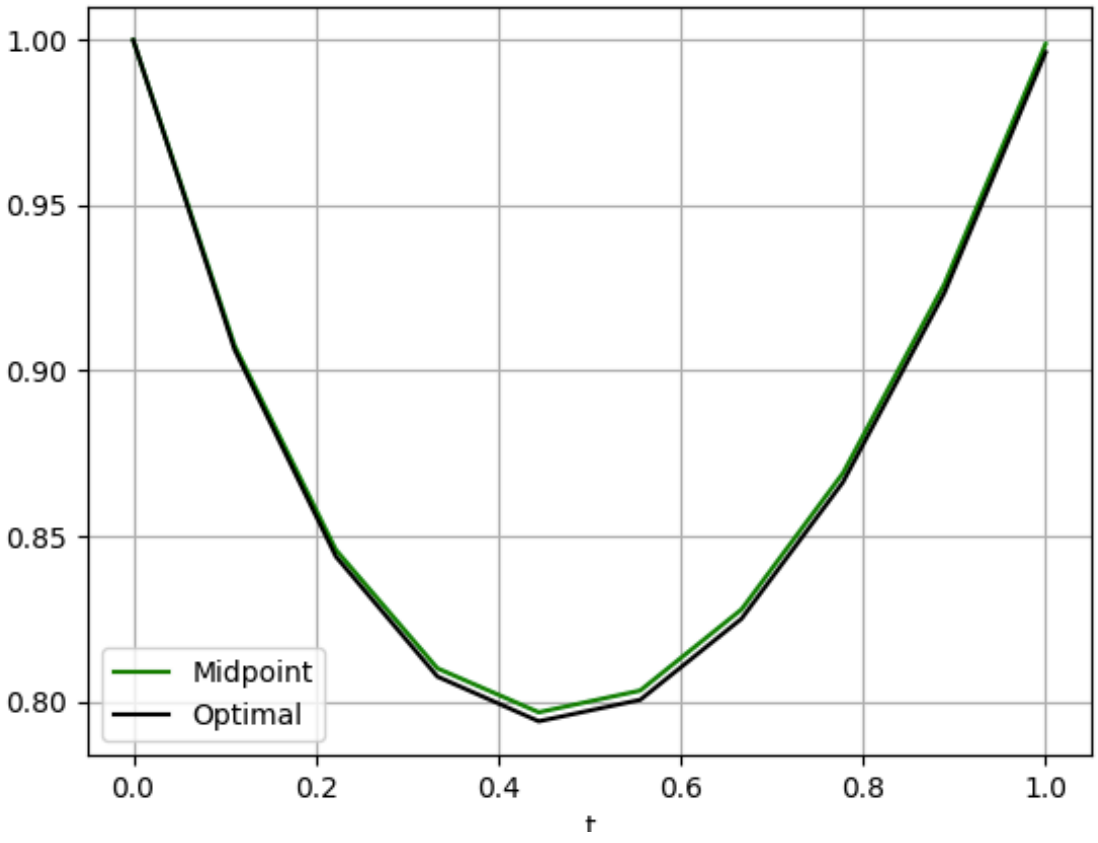


Tabela obliczonych wartości

	t	Optimal	Midpoint
0	0.000000	1.000000	1.000000
1	0.111111	0.906331	0.907582
2	0.222222	0.843737	0.845772
3	0.333333	0.807482	0.809976
4	0.444444	0.793949	0.796678
5	0.555556	0.800432	0.803246
6	0.666667	0.824976	0.827783
7	0.777778	0.866258	0.869002
8	0.888889	0.923484	0.926139
9	1.000000	0.996317	0.998878

Błąd Globalny

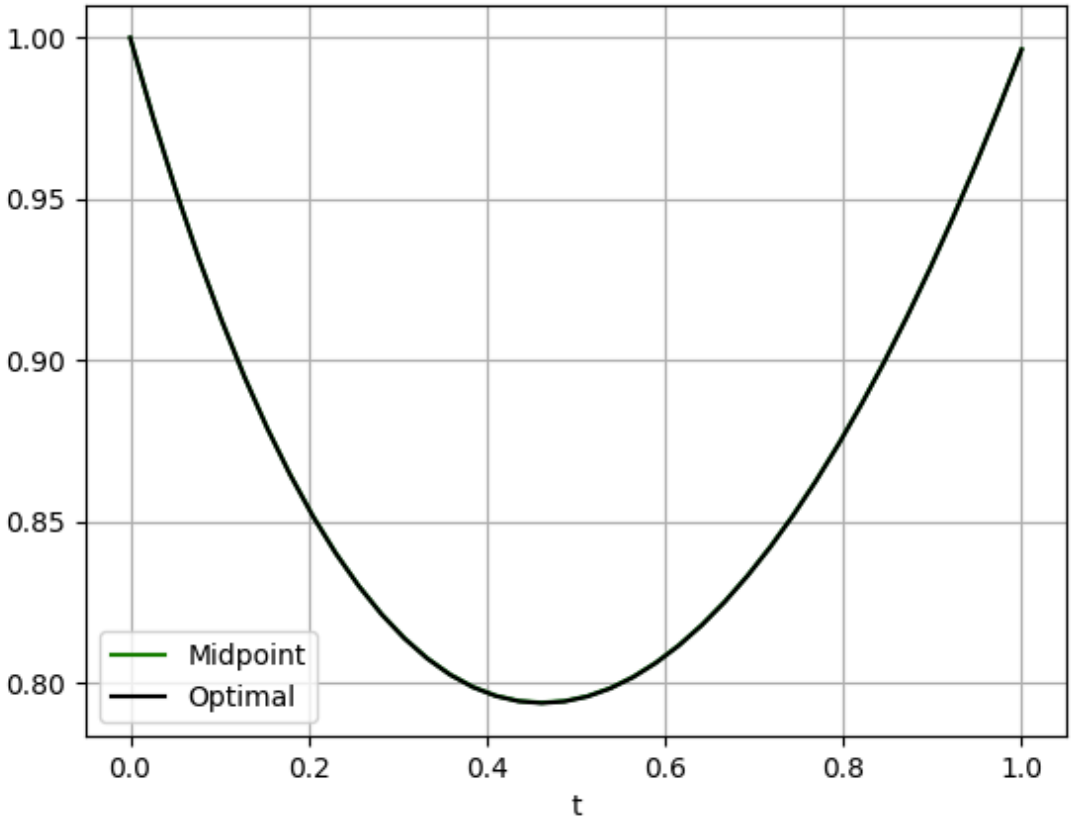
```
Name: Midpoint
Global Error: 0.00281
```

Wnioski

Lekka modyfikacja metody Eulera na start powoduje już znaczy wzrost dokładności obliczeń.

B

Wykres porównujący wartość oczekiwaną i uzyskaną



Błąd Globalny

```
Name: Midpoint
Global Error: 0.00281
```

Tabela obliczonych wartości

	t	Optimal	Midpoint
0	0.000000	1.000000	1.000000
1	0.025641	0.975331	0.975347
2	0.051282	0.952553	0.952584
3	0.076923	0.931589	0.931632
4	0.102564	0.912365	0.912420
5	0.128205	0.894812	0.894878
6	0.153846	0.878865	0.878940
7	0.179487	0.864463	0.864547
8	0.205128	0.851548	0.851639
9	0.230769	0.840065	0.840163
10	0.256410	0.829964	0.830068
11	0.282051	0.821196	0.821305
12	0.307692	0.813716	0.813830
13	0.333333	0.807482	0.807600
14	0.358974	0.802455	0.802575
15	0.384615	0.798596	0.798720
16	0.410256	0.795872	0.795998
17	0.435897	0.794250	0.794378
18	0.461538	0.793701	0.793831
19	0.487179	0.794196	0.794327
20	0.512821	0.795709	0.795841
21	0.538462	0.798218	0.798350
22	0.564103	0.801699	0.801832
23	0.589744	0.806133	0.806266
24	0.615385	0.811501	0.811635
25	0.641026	0.817787	0.817920
26	0.666667	0.824976	0.825109
27	0.692308	0.833054	0.833186
28	0.717949	0.842009	0.842141
29	0.743590	0.851830	0.851962
30	0.769231	0.862509	0.862640
31	0.794872	0.874038	0.874167
32	0.820513	0.886409	0.886538
33	0.846154	0.899618	0.899746
34	0.871795	0.913661	0.913787
35	0.897436	0.928534	0.928660
36	0.923077	0.944236	0.944361
37	0.948718	0.960767	0.960891
38	0.974359	0.978127	0.978250
39	1.000000	0.996317	0.996440

Wnioski

Tak samo jak w metodzie Eulera zwiększenie liczby podziałów zwiększyło dokładność, ale także trzeba zaznaczyć, że nakład pracy na stworzenie metody midpoint nie jest tak duży, a efekty jakie można uzyskać są dobre.

Metoda Rungego-Kuttego

Implementacja na bazie metody rzędu 4.

Wygląd metody obliczającej x[i+1] element (wymagana do implementacji przez klasę ISolver).

```
def _method(self, i, results, linspace):
    diff = linspace[i + 1] - linspace[i]

    first = self.functions(results[i], linspace[i])
    second = self.functions(results[i] + first * diff / 2., linspace[i] + diff / 2.)
    third = self.functions(results[i] + second * diff / 2., linspace[i] + diff / 2.)
    fourth = self.functions(results[i] + third * diff, linspace[i] + diff)

    return results[i] + (diff / 6.) * (first + 2*second + 2*third + fourth)
```

A

Wykres porównujący wartość oczekiwaną i uzyskaną

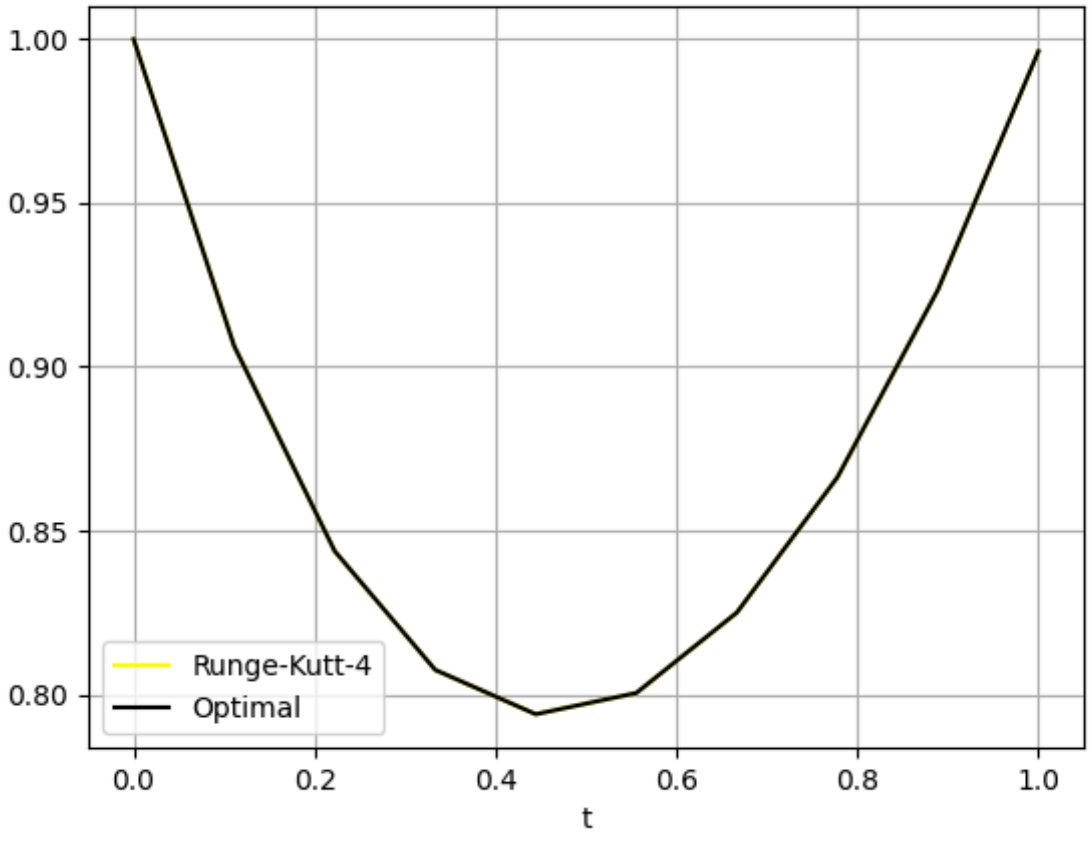


Tabela obliczonych wartości

	t	Optimal	Runge-Kutt-4
0	0.000000	1.000000	1.000000
1	0.111111	0.906331	0.906335
2	0.222222	0.843737	0.843742
3	0.333333	0.807482	0.807489
4	0.444444	0.793949	0.793957
5	0.555556	0.800432	0.800440
6	0.666667	0.824976	0.824985
7	0.777778	0.866258	0.866267
8	0.888889	0.923484	0.923493
9	1.000000	0.996317	0.996327

Błąd Globalny

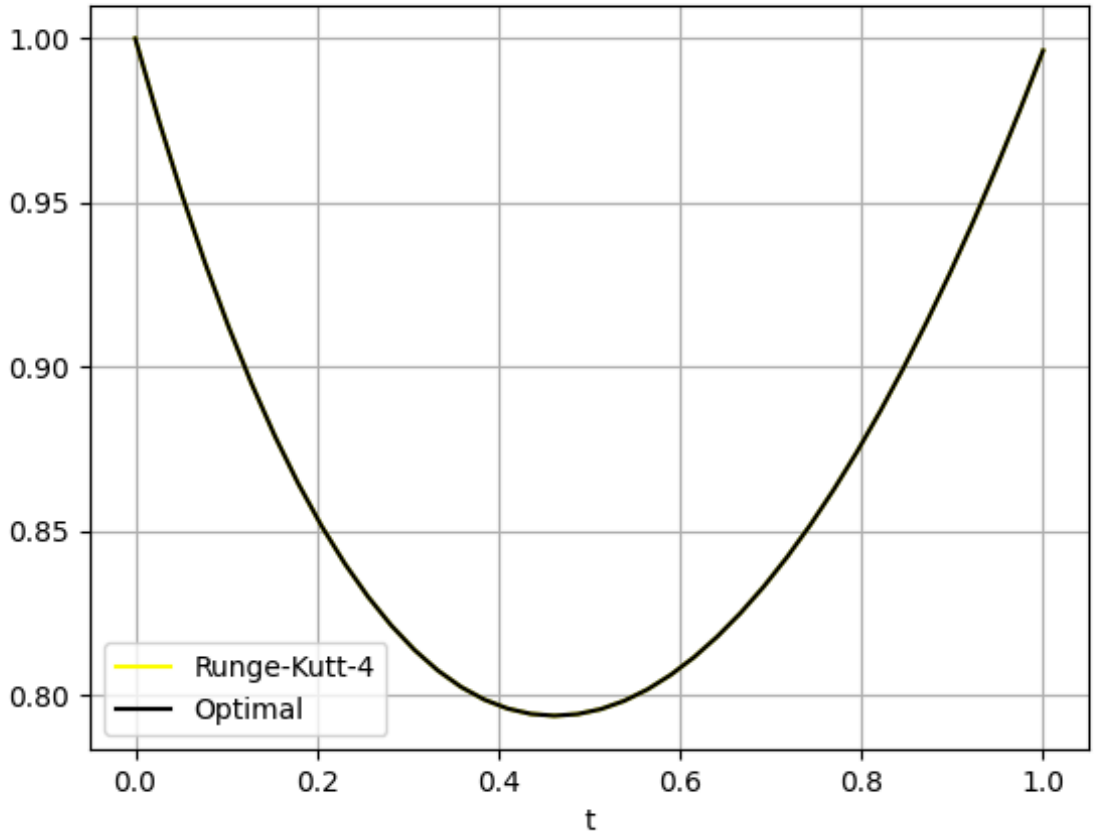
```
Name: Runge-Kutt-4
Global Error: 1e-05
```

Wnioski

Metoda Rungego-Kutta4 daje bardzo dobre wyniki już dla niewielkich n, więc na pewno warto jej używać do tego typu obliczeń.

B

Wykres porównujący wartość oczekiwaną i uzyskaną



Błąd Globalny

```
Name: Runge-Kutt-4
Global Error: 0.0
```

Tabela obliczonych wartości

	t	Optimal	Runge-Kutt-4
0	0.000000	1.000000	1.000000
1	0.025641	0.975331	0.975331
2	0.051282	0.952553	0.952553
3	0.076923	0.931589	0.931589
4	0.102564	0.912365	0.912365
5	0.128205	0.894812	0.894812
6	0.153846	0.878865	0.878865
7	0.179487	0.864463	0.864463
8	0.205128	0.851548	0.851548
9	0.230769	0.840065	0.840065
10	0.256410	0.829964	0.829964
11	0.282051	0.821196	0.821196
12	0.307692	0.813716	0.813716
13	0.333333	0.807482	0.807482
14	0.358974	0.802455	0.802455
15	0.384615	0.798596	0.798596
16	0.410256	0.795872	0.795872
17	0.435897	0.794250	0.794250
18	0.461538	0.793701	0.793701
19	0.487179	0.794196	0.794196
20	0.512821	0.795709	0.795709
21	0.538462	0.798218	0.798218
22	0.564103	0.801699	0.801699
23	0.589744	0.806133	0.806133
24	0.615385	0.811501	0.811501
25	0.641026	0.817787	0.817787
26	0.666667	0.824976	0.824976
27	0.692308	0.833054	0.833054
28	0.717949	0.842009	0.842009
29	0.743590	0.851830	0.851830
30	0.769231	0.862509	0.862509
31	0.794872	0.874038	0.874038
32	0.820513	0.886409	0.886409
33	0.846154	0.899618	0.899618
34	0.871795	0.913661	0.913661
35	0.897436	0.928534	0.928534
36	0.923077	0.944236	0.944236
37	0.948718	0.960767	0.960767
38	0.974359	0.978127	0.978127
39	1.000000	0.996317	0.996317

Wnioski

Przy zwiększonym n ta metoda jest praktycznie nie odróżnialna od rozwiązania przyjętego przez nas jako wzorcowe. Potwierdza to że ta metoda jest niebywale dobra do obliczeń tego typu.

Metoda Taylora

Implementacja dozwalająca na wprowadzenie n pochodnych.

Wygląd metody obliczającej x[i+1] element (wymagana do implementacji przez klasę ISolver).

```
def _method(self, i, results, linspace):
    diff = linspace[i+1] - linspace[i]
    f_count = len(self.functions)
    result = results[i]
    x, t = results[i], linspace[i]
    sil = 1
    for j in range(f_count):
        sil = sil / (j + 1)
        k = self.functions[j](x, t)
        result += sil * k * (diff**(j+1))

    return result
```

A

Wykres porównujący wartość oczekiwaną i uzyskaną

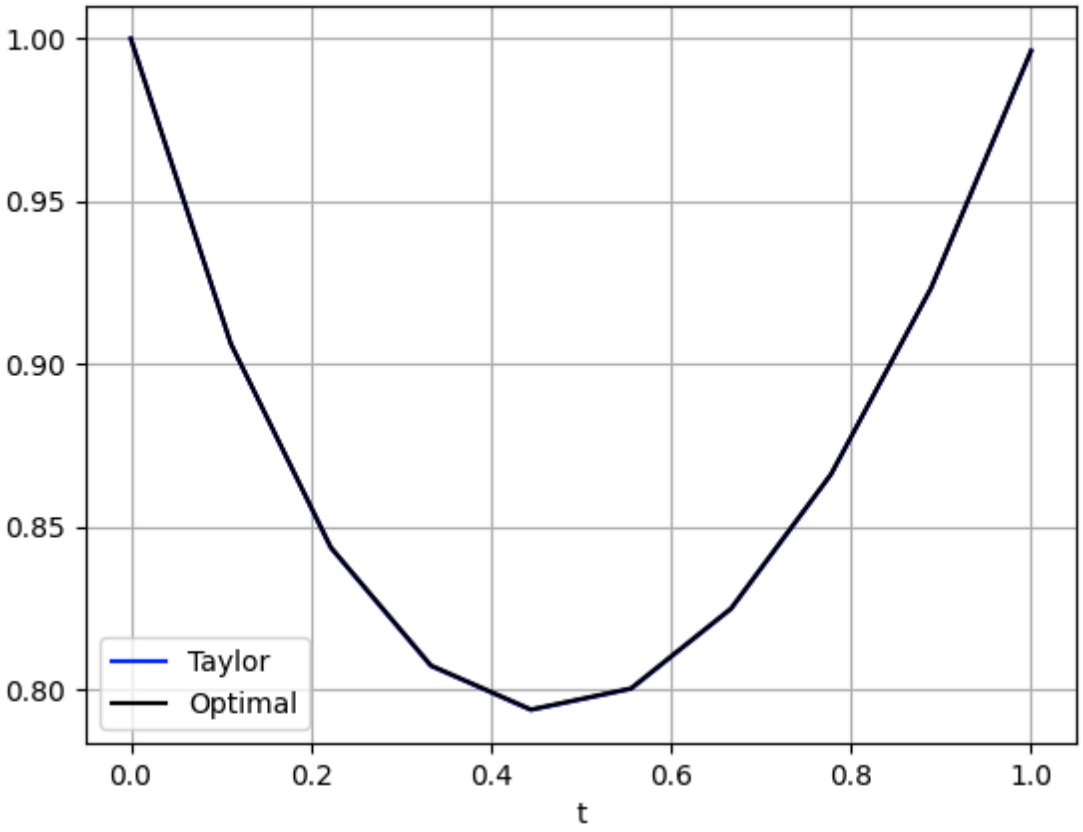


Tabela obliczonych wartości

	t	Optimal	Taylor
0	0.000000	1.000000	1.000000
1	0.111111	0.906331	0.906264
2	0.222222	0.843737	0.843629
3	0.333333	0.807482	0.807351
4	0.444444	0.793949	0.793808
5	0.555556	0.800432	0.800289
6	0.666667	0.824976	0.824837
7	0.777778	0.866258	0.866125
8	0.888889	0.923484	0.923359
9	1.000000	0.996317	0.996201

Błąd Globalny

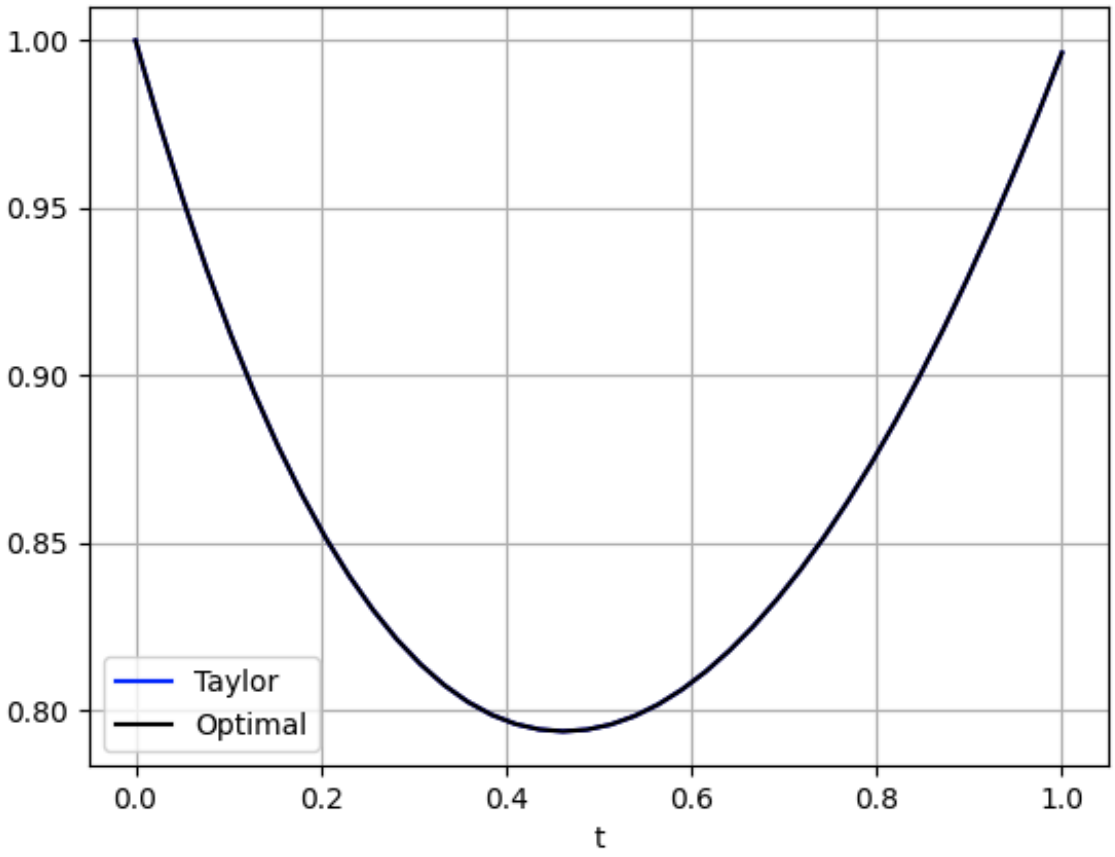
```
Name: Taylor
Global Error: 0.00014
```

Wnioski

Metoda Taylora rzędu 3 również ma bardzo dobra dokładność jest niewiele gorsza od rozwiązania wcześniejszego jednakże potrzebny nadała pracy jest znacznie większy, trzeba wyznaczyć pochodne funkcji.

B

Wykres porównujący wartość oczekiwaną i uzyskaną



Błąd Globalny

```
Name: Taylor
Global Error: 0.0
```

Tabela obliczonych wartości

	t	Optimal	Taylor
0	0.000000	1.000000	1.000000
1	0.025641	0.975331	0.975331
2	0.051282	0.952553	0.952553
3	0.076923	0.931589	0.931588
4	0.102564	0.912365	0.912364
5	0.128205	0.894812	0.894811
6	0.153846	0.878865	0.878864
7	0.179487	0.864463	0.864462
8	0.205128	0.851548	0.851546
9	0.230769	0.840065	0.840064
10	0.256410	0.829964	0.829963
11	0.282051	0.821196	0.821195
12	0.307692	0.813716	0.813715
13	0.333333	0.807482	0.807481
14	0.358974	0.802455	0.802453
15	0.384615	0.798596	0.798595
16	0.410256	0.795872	0.795870
17	0.435897	0.794250	0.794249
18	0.461538	0.793701	0.793699
19	0.487179	0.794196	0.794194
20	0.512821	0.795709	0.795708
21	0.538462	0.798218	0.798216
22	0.564103	0.801699	0.801697
23	0.589744	0.806133	0.806131
24	0.615385	0.811501	0.811500
25	0.641026	0.817787	0.817786
26	0.666667	0.824976	0.824975
27	0.692308	0.833054	0.833052
28	0.717949	0.842009	0.842007
29	0.743590	0.851830	0.851829
30	0.769231	0.862509	0.862508
31	0.794872	0.874038	0.874036
32	0.820513	0.886409	0.886408
33	0.846154	0.899618	0.899617
34	0.871795	0.913661	0.913659
35	0.897436	0.928534	0.928533
36	0.923077	0.944236	0.944235
37	0.948718	0.960767	0.960766
38	0.974359	0.978127	0.978126
39	1.000000	0.996317	0.996316

Wnioski

Przy zwiększonym n metoda Taylora jest tak samo dokładna jak metoda Rungego-Kutta,

Ogólnie wnioski

Na powyższych wykresach widzimy, że istnieje kilka czynników prowadzących do poprawy wyników obliczeń.

Głównym według mnie jest liczba przedziałów, widzimy znaczy wzrost dokładności przy zwiększeniu liczby podziałów (aczkolwiek faktem jest ze wyniki dalej odstają).

Kolejnym istotnym czynnikiem jest metoda jak widzimy niektóre metody już dla małych n mogą dawać dość dokładne wyniki, kolejnym faktem przemawiającym za stosowaniem odpowiedniej metody jest ‚energooszczędność' np porównując metodę Eulera B do Rungego-Kutta4 A nawet przy małej liczbie n uzyskujemy znacznie lepsze wyniki.