# Metody różnic skończonych
## Sprawozdanie nr 2
## Zagadnienie Dirichletta

Kacper Dąbrowski

Kwiecień 2024

### Spis treści

# 1 Ogólne zagadnienie brzegowe Dirichletta dla równania różniczkowego II rzędu

**Uwaga.** Funkcje $\alpha(x), \beta(x), \gamma(x), f(x), y(x)$ będą zapisywane bez oznaczania ich zależności od zmiennej $x$ tz. np. $\alpha(x) = \alpha$. Podobnie skrócony jest zapis funkcji w punkcie tz. funkcja $\alpha(x_1) = \alpha_1$.

Zagadnieniem Dirichletta nazywamy warunki brzegowe równania różniczkowego dane w następujący sposób:

$$\begin{cases} \alpha y'' + \beta y' + \gamma y = f \\ y|_{x=a} = y_a \\ y|_{x=b} = y_b \end{cases} \qquad x \in [a,b]$$

Oznacza to, że mamy podane jawnie punkty na brzegach funkcji. Aproksymację zadania wykonujemy przy pomocy schematów różnicowych. Po podzieleniu funkcji na przedziały $x_i$ otrzymujemy, że:

$$\text{dla } x_i : \alpha_i \tfrac{y_{i-1}-2y_i+y_{i+1}}{h^2} + \beta_i \tfrac{-y_{i-1}+y_{i+1}}{2h} + \gamma_i y_i = f_i$$

Po przekształceniach otrzymujemy taką postać macierzową:

$$A = \begin{bmatrix} -2\alpha_1 + h^2\gamma_1 & \alpha_1 + \frac{\beta_1 h}{2} & 0 & \dots & & 0 \\ \alpha_2 - \frac{\beta_2 h}{2} & -2\alpha_2 + h^2\gamma_2 & \alpha_2 + \frac{\beta_2 h}{2} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & \dots & \dots & 0 & \alpha_n - \frac{\beta_n h}{2} & -2\alpha_n + h^2\gamma_n \end{bmatrix} \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$\mathbf{f} = \begin{bmatrix} h^2 f_1 \\ h^2 f_2 \\ \vdots \\ h^2 f_n \end{bmatrix} \mathbf{g} = \begin{bmatrix} y_a(\alpha_1 - \frac{\beta_1 h}{2}) \\ 0 \\ \vdots \\ y_b(\alpha_n + \frac{\beta_n h}{2}) \end{bmatrix}$$

Teraz możemy wyliczyć wartości w punktach węzłowych poprzez rozwiązanie układu równań:

$$A\mathbf{y} = \mathbf{f} - \mathbf{g}$$

# 2 Program do rozwiązywania problemu

Program został napisany w oprogramowaniu Python. Przejście na ten język programowaia jest spowodowany chęcią stworzenia klasy, która obsługiwałaby ten problem w sposób łatwy dla użytkownika. W klasie zostały zamieszczone opisy definiujące co dokładnie użytkownik powinien podać aby otrzymać wynik. Klasa zawiera szereg metod, które są opisane poniżej.

Biblioteki potrzebne do działania programu:

- MatPlotLib - obsługa wykresów z Matlaba
- Numpy - obługa numeryki w Pythonie

## 2.1 Kontruktor klasy

Kontruktor klasy *Dirichlett* przedstawiony jest na rysunku 1. Argumentami klasy są funkcje *lambda*, które zasadą działania przypominają funkcje *handle* z Matlaba. To znaczy, że jak zapiszemy $y = lambda\ x :$ i za dwukropkiem umieścimy dowolne funkcje lub przekształcenia, to możemy potem je wywołać na tym co umieścimy w argumencie funkcji $y$. Na przykład:

$$y = lambda\ \ x : 2 * x + cos(x)$$

to dla $y(0)$ dostaniemy wynik 1.

Bardzo wygodne jest to, że możemy takie funkcje umieszczać jako argumenty innych funkcji lub klas. Python dobrze interpretuje dane wejściowe i wie, że dany argument funkcji może zostać wywołany jako inna funkcja.

Argumentami wejściowymi konstruktowa klasy są funkcje będące współczynnikami równania różniczkowego II rzędu oraz funkcja źródłowa $f$. Zaimplementowany jest również argument opcjonalny *solveType*, który domyślnie jest ustawiony na wartość logiczną *False* i odpowiada za to, czy chcemy rozwiązać układ równań metodą Thomasa czy nie. Przy inicjacji klasy tworzone są inne zmienne globalne, które będą przydatne w dalszej części.

```python
class Dirichlett:
    '''
    Solves second order differencial equations with Dirichlett boundary conditions.
    '''
    def __init__(self, alpha, beta, gamma, f, solveType : bool=False) -> None:
        '''
        Initiation of problem given by equation: alpha(x)*y''+beta(x)*y'+gamma(x)*y = f(x).

        Args:
            alpha (lambda function): alpha coefficient
            beta (lambda function): beta coefficient
            gamma (lambda function): gamma coefficient
            f (lambda function): function f
            solveType (bool, optional): if True, solves the problem using ThomasSolve. Defaults to False.
'''     self.alpha = alpha
        self.beta = beta
        self.gamma = gamma
        self.f = f
        self.solveType : bool = solveType

        self.solution = None
        self.label : str = ""

        self.sol_exist : bool = False
        self.set_n : bool = False
        self.set_ab : bool = False

        self.n : int = 1
        self.a : float = 0
        self.b : float = 2
        self.val_a : float = 0
        self.val_b : float = 0
        self.h : float = 0

        self.x : np.ndarray
        self.unknown_x : np.ndarray
```

**Rysunek 1:** Kod konstruktora klasy

## 2.2 Metody konfigurujące klasę

Aby klasa była w stanie rozwiązać równanie różniczkowe II rzędu, musimy jej podać wszystkie parametry zadania. W konstruktorze zawarte są tylko i wyłącznie informacje o samym równaniu. Resztę parametrów należy podać poprzez różne metody.

Pierwszą taką metodą jest funkcja *setN*, jak nazwa wskazuje służy ona do ustawienia parametru $n$ naszego zadania. Ta metoda przedstawiona jest na rysunku 2. Dodatkowo ustawiony jest parametr *set_n* na wartość *True*. Funkcjonalność tego parametru będzie opisana w dalszej części. Następnie wywoływana jest funkcja wewnętrzna *setH*, której zadaniem jest wyliczenie parametru $h$ oraz wyznaczenie punktów węzłowych. Funkcja ta znajduje się na rysunku 3.

```python
def setN(self, n : int) -> None:
        '''
        Sets the number of points.

        Args:
            n (int): number of points'''
        self.n = n
        self.set_n = True
        self.setH()
```

**Rysunek 2:** Kod metody ustalającej $n$

```python
def setH(self) -> None:
        '''Sets distance between points'''
        self.h = (self.b - self.a)/(self.n+1)
        self.x = np.linspace(self.a, self.b, self.n+2)
        self.unknown_x = self.x[1:self.n+1]
```

**Rysunek 3:** Kod metody ustalającej $h$

Metoda *setAB* służy do ustawienia warunków brzegowych Dirichletta. Funkcjonalność tej funkcji zawarta jest na rysunku 4. Za argument przyjmuje dwie listy $a$ oraz $b$. Każda z nich zawiera dwa parametry odpowiadające ich brzegowi. Pierwszy parametr listy $a$ odpowiada argumentowi, w którym znany jest lewy brzeg, a drugi odpowiada jego wartości (jak chcemy np. wybrać punkt 3 a w nim wartość wynosi 8 to wprowadzamy listę $a = [3,8]$). Odpowiednio dla brzegu $b$. Podobnie jak przy *setN* zmieniana jest zmienna losowa *set_ab* na wartość *True*. Następnie wywoływana jest funkcja *setH* pokazana na rysunku 3.

```
def setAB(self, a : list, b : list) -> None:
        '''
        Sets boundary conditions.

        Args:
            a (list): Two element list containing
            value of left boundary argument and value.
            b (list): Two element list containing
            value of right boundary argument and value.'''
        self.a = a[0]
        self.b = b[0]
        self.val_a = a[1]
        self.val_b = b[1]
        self.set_ab = True
        self.setH()
```

**Rysunek 4:** Kod metody ustalającej parametry brzegowe

Na rysunku 5 przedstawiona jest funkcja *setLabel*. Odpowiedzialna jest ona za konfigurację tytułu wykresu, na którym prezentowane są wyniki. Co ważne obsługuje ona składnię LaTeX'a, przez co można tam na przykład wpisać jakie równanie różniczkowe rozwiązujemy.

```
def setLabel(self, label : str) -> None:
        '''
        Sets given equation in LaTeX to write it on plot.

        Args:
            label (str): label of graph'''
        self.label = label
```

**Rysunek 5:** Kod metody ustalającej tytuł wykresu

Ostatnia metoda do konfiguracji zadania jest opcjonalna. Służy ona do dodania rozwiązania analitycznego. Funkcja *addSolution* zmienia także zmienną globalną *sol_exist* po to, aby program wiedział, że rozwiązanie zostało dodane. Funkcja ta przedstawiona jest na rysunku 6.

```
def addSolution(self, solution) -> None:
        '''Add solution to the problem'''
        self.solution = solution
        self.sol_exist = True
```

**Rysunek 6:** Kod metody dodającej rozwiązanie analityczne

## 2.3 Skrypt rozwiązujący problem

Za rozwiązanie równania różniczkowego II rzędu z zagadnieniem Dirichletta odpowiada funkcja *solve*, przedstawiona na rysunku 7. Na samym początku sprawdzane jest, czy wszystkie parametry, konieczne do rozwiązania zadania, zostały spełnione. Właśnie w tym momencie wykorzystujemy zmienne, które zmienialiśmy przy konfiguracji $n$'a oraz warunków brzegowych. Jeśli nie zostały wcześniej wywołane tamte funkcje, program zwróci błąd mówiący, że nie można rozwiązać zadania. Następnie generowane są wszystkie macierze potrzebne do rozwiązania problemu. Na samym końcu, w zależności od wyboru użytkownika, program rozwiązuje układ równań odpowiednią metodą. Do rozwiązywania układów równań w Pythonie wykorzystana jest biblioteka *numpy* i jej moduł *linalg* odpowiedzialny za liniową algebrę liniową. Funkcja *ThomasSolve* będzie opisana w dalszej części. Przy okazji obliczany jest czas każdej z metod. Funkcja ta zwraca rozwiązanie jakie wyliczyła.

```python
def solve(self) -> float:
        '''
        Solves the problem.

        Returns:
            float: solution depending on solve type variable.'''
        if not all([self.set_ab, self.set_n]):
            raise ValueError("Cannot solve equation without setting parameters")
        v1 : np.ndarray = np.array([])
        v2 : np.ndarray = np.array([])
        v3 : np.ndarray = np.array([])

        for i in range(len(self.unknown_x)):
            v1 = np.append(v1, -2*self.alpha(self.unknown_x[i]) + self.gamma(self.unknown_x[i])*self.h**2)
            v2 = np.append(v2, self.alpha(self.unknown_x[i]) + self.beta(self.unknown_x[i])*self.h/2)
            v3 = np.append(v3, self.alpha(self.unknown_x[i]) - self.beta(self.unknown_x[i])*self.h/2)


        v2 = v2[0:len(v2)-1]
        v3 = v3[1:len(v3)]

        F = list(map(lambda x: x*self.h**2, list(map(self.f, self.unknown_x))))
        G = np.zeros(len(self.unknown_x))

        G[0] = self.val_a*(self.alpha(self.unknown_x[0])-(1/2)*self.beta(self.unknown_x[0])*self.h)
        G[-1] = self.val_b*(self.alpha(self.unknown_x[-1])+(1/2)*self.beta(self.unknown_x[-1])*self.h)
        if self.solveType:
            st = time.time()
            sol = ThomasSolve(v3,v1,v2,F-G)
            et = time.time()
        else:
            A = np.diag(v1) + np.diag(v2,1) + np.diag(v3,-1)
            st = time.time()
            sol =  LA.solve(A,F-G)
            et = time.time()
        print(f"Estymowany czas działania funkcji: {et - st} sekund")
        return sol
```

**Rysunek 7:** Kod metody rozwiązującej równanie

## 2.4 Rysowanie wykresów

Pierwszą metodą odpowiadającą za obsługę wykresów jest funkcja *plt_config* jej zadaniem jest ustawienie parametrów wykresu tak, aby obługiwała LaTeX'a. Co ważne trzeba mieć na komputerze zainstalowane oprogramowanie MiKTeX, aby poprawnie działały wykresy. Jest to najpewniej spowodowane faktem, że Python kompiluje w locie tekst jaki mu się wprowadzi. Przewagą tego rozwiązania nad Matlabem jest to, że obsługiwany jest tutaj package z językiem polskim. Metoda ta przedstawiona jest na rysunku 8.

```python
def plt_config(self) -> None:
        '''
        Configures plot.
        '''
        A = 6
        plt.rc('figure', figsize=[46.82 * .5**(.5 * A), 33.11 * .5**(.5 * A)])
        plt.rc('text', usetex=True)
        plt.rc('text.latex', preamble=r'\usepackage{polski}')
        plt.rc('font', family='serif')
```

**Rysunek 8:** Kod metody konfigurującej wykres

Główną metodą programu jest funkcja *show*. Na początku wykonywana jest konfiguracja wykresu. Następnie skrypt rozwiązuje równanie i rozwiązanie łączy z warunkami brzegowymi. Następnie w zależności od tego czy podane zostało rozwiązanie analityczne, obliczany jest błąd i rysowane jest rozwiązanie. Jeśli rozwiązanie istnieje podawany jest także błąd pomiędzy rozwiązaniem analitycznym a numerycznym. Metoda ta ukazana jest na rysunku 9.

```python
def show(self) -> str:
        '''
        Plots the solution.

        Returns:
            str: Error norm if can.'''
        self.plt_config()
        unknown_sol = self.solve()
        approx_sol = np.insert(unknown_sol, 0, self.val_a)
        approx_sol = np.insert(approx_sol, len(approx_sol), self.val_b)
        if self.sol_exist:
            real_sol = list(map(self.solution, self.x))
            error_sol = np.abs(approx_sol-real_sol)
            error_norm = LA.norm(error_sol, np.inf)
            plt.subplot(211)
            plt.grid(True)
            domain = np.linspace(self.a, self.b, 1000)
            plt.plot(domain ,list(map(self.solution, domain)), color='b', label=r"Rzeczywista funkcja")
            if self.n < 30:
                plt.scatter(self.x ,approx_sol, color='g', label=r"Aproksymacja")
            else:
                plt.plot(self.x ,approx_sol, color='g', label=r"Aproksymacja")
            plt.title(self.label)
            plt.ylabel(r"Wartości funkcji $f(x)$")
            plt.legend()
            plt.subplot(212)
            plt.grid(True)
            plt.plot(self.x ,error_sol, color='r', label=r"Błąd")
            plt.title(r"Wykres błędu")
            plt.xlabel(r"Wartości $x$")
            plt.ylabel(r"Wartość błędu")
            plt.subplots_adjust(hspace=0.4)
            plt.show()
            self.result = approx_sol
            return error_norm
        plt.grid(True)
        plt.plot(self.x ,approx_sol, color='g', label=r"Aproksymacja")
        plt.title(self.label)
        plt.xlabel(r"Wartości $x$")
        plt.ylabel(r"Wartości funkcji $f(x)$")
        plt.subplots_adjust(hspace=0.4)
        plt.show()
        self.result = approx_sol
        return ""
```

**Rysunek 9:** Kod metody rysującej wykresy

Ostatnią metodą zawartą w klasie *Dirichlett* jest metoda *__str__*. Jest to specjalna metoda służąca do tekstowej reprezentacji klasy. Głównym jej zadaniem jest, aby po wywołaniu *print(klasa)*, program wykonywał metody, które wyświetlą użytkownikowi wynik. Metoda ta przedstawiona jest na rysunku 10.

```python
def __str__(self) -> str:
        '''
        String representation of result. Shows result in plot with simple print function.
        '''
        solution = self.show()
        if self.sol_exist:
            return f"Rozwiazanie przyblizone:\n {self.result} \n Blad: {solution}"
        return f"Rozwiazanie przyblizone:\n {self.result}"
```

**Rysunek 10:** Kod metody obsługującej reprezentację tekstową

## 3 Metoda Thomasa

Metoda Thomasa dla układów trójprzekątniowych jest metodą rozwiązywania układów równań, w których macierz główna $A$ jest macierzą trójprzekątniową. To znaczy:

$$A = \begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

Po sporej liczbie przekształceń jesteśmy w stanie otrzymać algorytm, który bardzo niskim kosztem rozwiązuje taki typ układu równań. Kod implementacji tego algorytmu jest na rysunku 10.

```python
def ThomasSolve(a, b, c, d) -> float:
    """
    Solves a tridiagonal system of linear equations using Thomas algorithm.

    Args:
        a (numpy.ndarray): Bottom diagonal
        b (numpy.ndarray): Main diagonal
        c (numpy.ndarray): Upper diagonal
        d (numpy.ndarray): Constant terms of linear equations
    Returns:
        numpy.ndarray: Solution of linear equations
    """
    n = len(b)
    beta = np.zeros(n)
    gamma = np.zeros(n)
    c = np.insert(c, len(c), 0)
    a = np.insert(a, 0, 0)
    beta[0] = -c[0]/b[0]
    gamma[0] = d[0]/b[0]
    for i in range(1, n):
        beta[i] = -c[i]/(b[i] + a[i]*beta[i-1])
        gamma[i] = (d[i] - a[i]*gamma[i-1])/(b[i] + a[i]*beta[i-1])
    x = np.zeros(n)
    x[-1] = gamma[-1]
    for i in range(n-2, -1, -1):
        x[i] = gamma[i] + beta[i]*x[i+1]
    return x
```

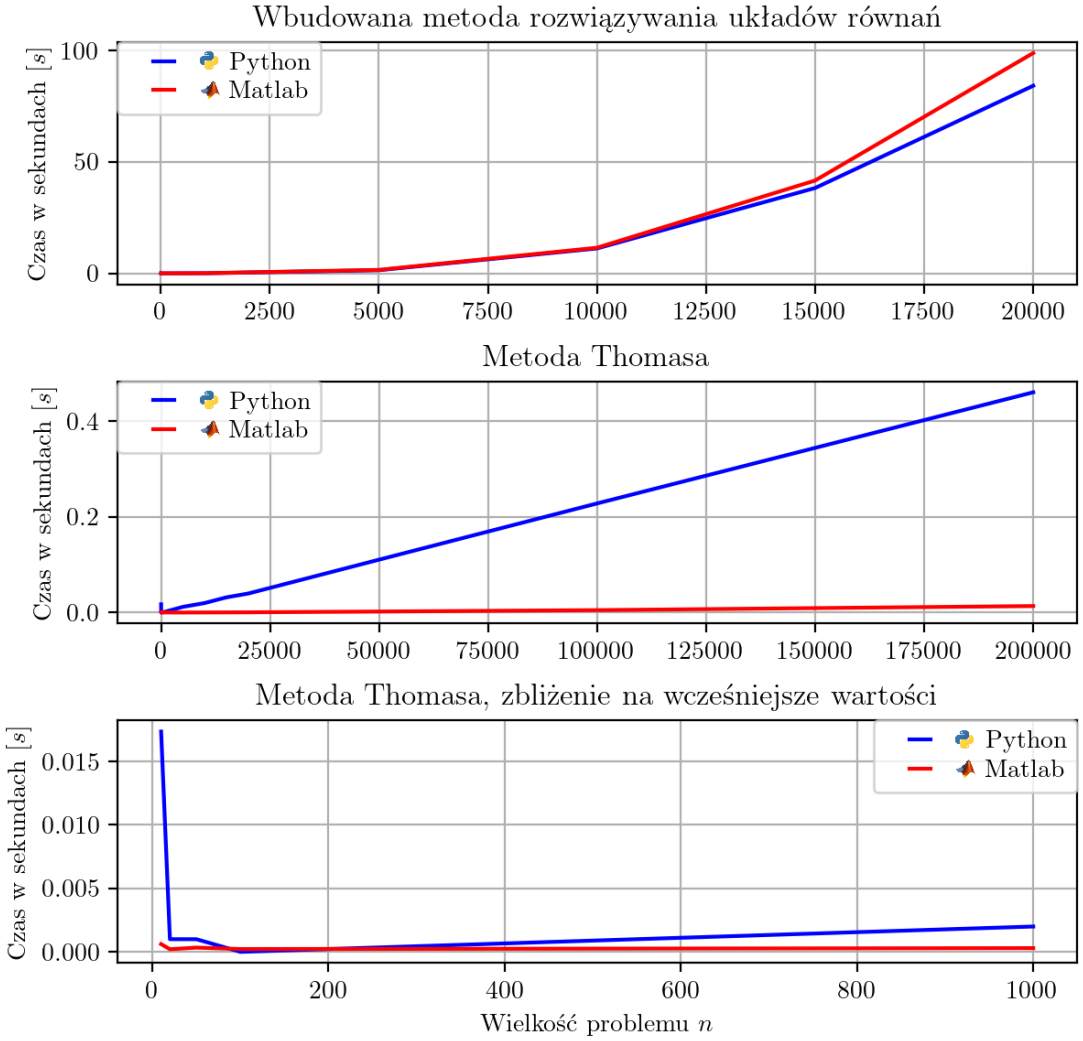
**Rysunek 11:** Kod implementujący metodę Thomasa

## 3.1 Porównanie czasów działania algorytmów

Algorytm Thomasa powinien w teorii mocno skrócić czasy rozwiązywania dużych układów równań. Do testów wykorzystane będą dwa takie same algorytmy Thomasa. Jeden napisany w Pythonie, a drugi w Matlabie. Zostaną one porównane z metodami wbudowanymi w Matlaba oraz w bibliotekę *numpy*. Aby wyniki był miarodajne wszystkie algorytmy rozwiązują ten sam układ, a wielkość każdego problemu jest równa $n$. Wyniki testu pokazane są w tabeli 1 oraz na rysunku 12.

**Tabela 1:** Zestawienie wyników porównania

| $n$ | Thomas (Python) | LinAlg.solve (Python) | Thomas (Matlab) | Linsolve (Matlab) |
|---|---|---|---|---|
| 10 | 0,017346 | 0,002030 | 0,000611 | 0,118818 |
| 20 | 0,000998 | 0,001963 | 0,000207 | 0,015077 |
| 50 | 0,000997 | 0,005985 | 0,000335 | 0,005280 |
| 100 | 0,000000 | 0,001953 | 0,000222 | 0,001497 |
| 1000 | 0,001995 | 0,033910 | 0,000295 | 0,068558 |
| 5000 | 0,011967 | 1,404332 | 0,000143 | 1,545856 |
| 10000 | 0,019978 | 11,176047 | 0,000295 | 11,497836 |
| 15000 | 0,031913 | 38,310257 | 0,000406 | 41,670646 |
| 20000 | 0,039895 | 84,224024 | 0,000603 | 98,914380 |
| 50000 | 0,110737 | | 0,002234 | |
| 100000 | 0,228359 | | 0,005139 | |
| 200000 | 0,460769 | | 0,013664 | |




**Rysunek 12:** Porównanie metod

### 3.1.1 Wnioski

Metoda Thomasa jest znacznie szybsza od metod wbudowanych. Mimo tej samej impementacji kodu różne języki programowania osiągały różne wyniki. Możemy się spodziewać, że kod w Pythonie będzie wolniejszy niż ten sam napisany w Matlabie. Głównym problemem jest to, że Python ogólnie jest jednym z najwolniejszych języków programowania. Największą jego wadą są wolne pętle, przez które widać różnice w czasach. Matlab był pisany po części w języku C przez co wykonuje pętle i obliczenia znacznie szybciej.

Zastkakujące się wydaje, że funkcja *solve* z Pythona jest trochę szybsza od Matlabowego *linsolve*'a. Dzieje się tak ponieważ biblioteka *numpy* jest prawie całkowicie napisana w języku C przez co jej prędkość może być szybsza niż Matlabowa funkcja *linsolve*.

Przez ograniczenia nie dało się wygenerować macierzy o rozmiarach większych niż $20000 \times 20000$ zarówno w Pythonie jak i Matlabie. Metoda Thomasa jest w stanie rozwiązywać dużo większe układy równań.

Mimo faktu, że zaimplementowana metoda Thomasa w Pythonie była wolniejsza, te różnice nie mają większego znaczenia. Jest tak ponieważ nawet dla gigantycznego układu $200000 \times 200000$ skrypt poradził sobie w niecałe pół sekundy.

# 4 Rozwiązywanie podstawowego równania różniczkowego II rzędu

W tej części będziemy rozwiązywać równania różniczkowe II rzędu z zagadnieniem Dirichletta postaci:

$$\begin{cases} y'' = f \\ y|_{x=a} = y_a \\ y|_{x=b} = y_b \end{cases} \qquad x \in [a,b]$$

## 4.1 Zadanie 1

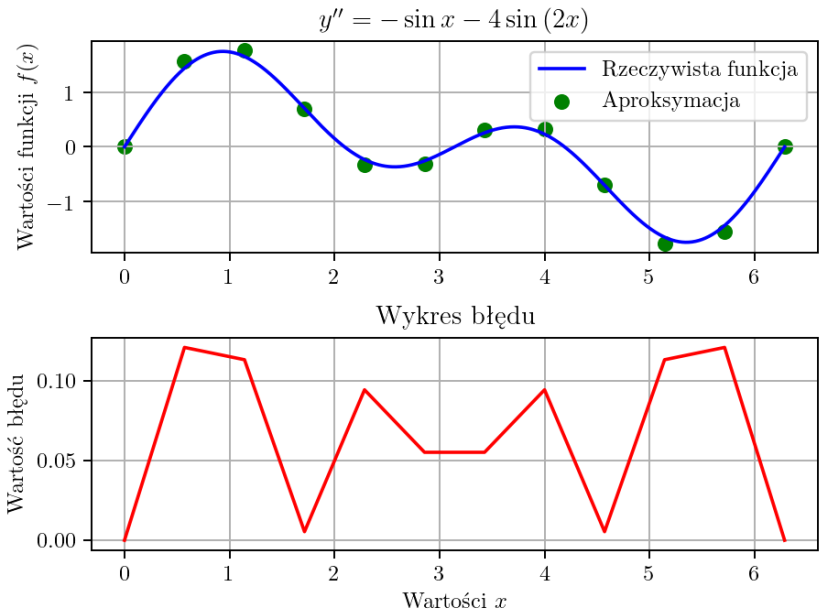
Mamy równanie różniczkowe II rzędu postaci:

$$\begin{cases} y'' = -\sin x - 4\sin(2x) \\ y|_{x=0} = 0 \\ y|_{x=2\pi} = 0 \end{cases} \qquad x \in [0,2\pi]$$

Rozwiązanie analityczne:

$$y = \sin x + \sin(2x)$$

Rozwiązanie numeryczne dla $n = 10$ przedstawione jest na rysunku 13. Rysunek 14 przedstawia kod źródłowy.




**Rysunek 13:** Wykres z rozwiązaniem dla $n = 10$

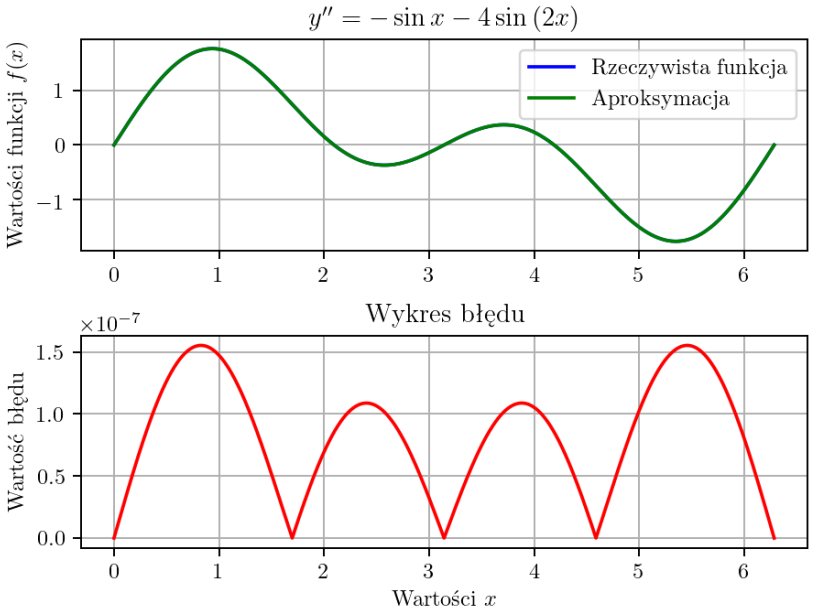
```python
from Dirichlett import Dirichlett
from numpy import cos, sin, pi
def main():
    label    = r"$y'' = -\sin{x}-4\sin{(2x)}$"
    alpha    = lambda x: 1                          # Współczynnik przy y''
    beta     = lambda x: 0                          # Współczynnik przy y'
    gamma    = lambda x: 0                          # Współczynnik przy y
    f        = lambda x: -sin(x)-4*sin(2*x)         # Wyraz wolny
    solution = lambda x: sin(x)+sin(2*x)            # Rozwiązanie analityczne
    n = 20                                          # Liczba węzłów
    a = [0, 0]                                      # Argument funkcji, wartość w punkcie
    b = [2*pi,0]                                    # -''- dla prawego punktu
    dirichlet = Dirichlett(alpha, beta, gamma, f, True)
    dirichlet.setN(n)
    dirichlet.setAB(a, b)
    dirichlet.setLabel(label)
    dirichlet.addSolution(solution)
    print(dirichlet)

if __name__ == '__main__':
    main()
```

**Rysunek 14:** Kod źródłowy

Kody źródłowe następnych zadań będą takie same tylko ze zmienionymi danymi. Zatem nie ma potrzeby powielania tego samego kodu w dalszych częściach.

Ciekawe jest to, że dla większej liczby punktów węzłowych kształt błędu jest zachowany. Jedyne co się zmienia to rząd wielkości. Dobrze to obrazuje rysunek 15.




**Rysunek 15:** Wykres z rozwiązaniem dla $n = 10000$

Zbiorowe wyniki błędów dla różnych $n$ znajdują się w tabeli 2.

## 4.2 Zadanie 2

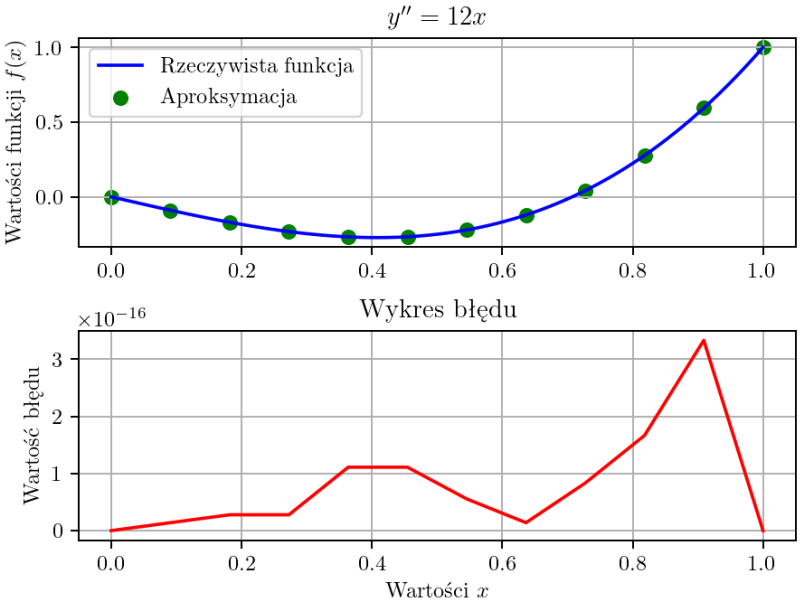
Mamy równanie różniczkowe II rzędu postaci:

$$\begin{cases} y'' = 12x \\ y|_{x=0} = 0 \\ y|_{x=1} = 1 \end{cases} \quad x \in [0,1]$$

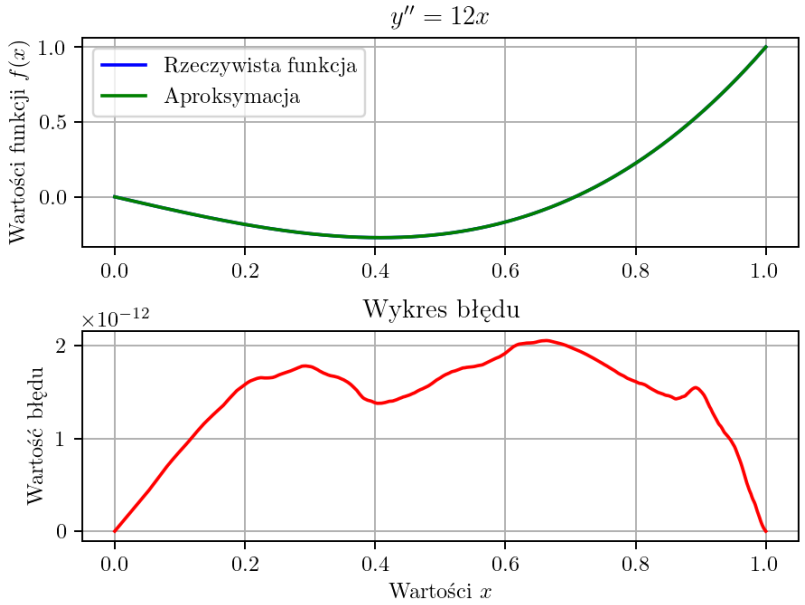
Rozwiązanie analityczne:

$$y = 2x^3 - x$$

Rozwiązanie numeryczne dla $n = 10$ przedstawione jest na rysunku 16. Rysunek 17 zawiera rozwiązanie dla $n = 10000$.




**Rysunek 16:** Wykres z rozwiązaniem dla $n = 10$




**Rysunek 17:** Wykres z rozwiązaniem dla $n = 10000$

W tym przypadku błąd nie utrzymał kształtu i dodatkowo był większy dla większego $n$'a.

# 5 Rozwiązanie ogólnego równania różniczkowego II rzędu z zagadnieniem Dirichletta

W tej części będziemy rozwiązywać równania różniczkowe II rzędu postaci:

$$\begin{cases} \alpha y'' + \beta y' + \gamma y = f \\ y|_{x=a} = y_a \\ y|_{x=b} = y_b \end{cases} \qquad x \in [a,b]$$

## 5.1 Zadanie 1

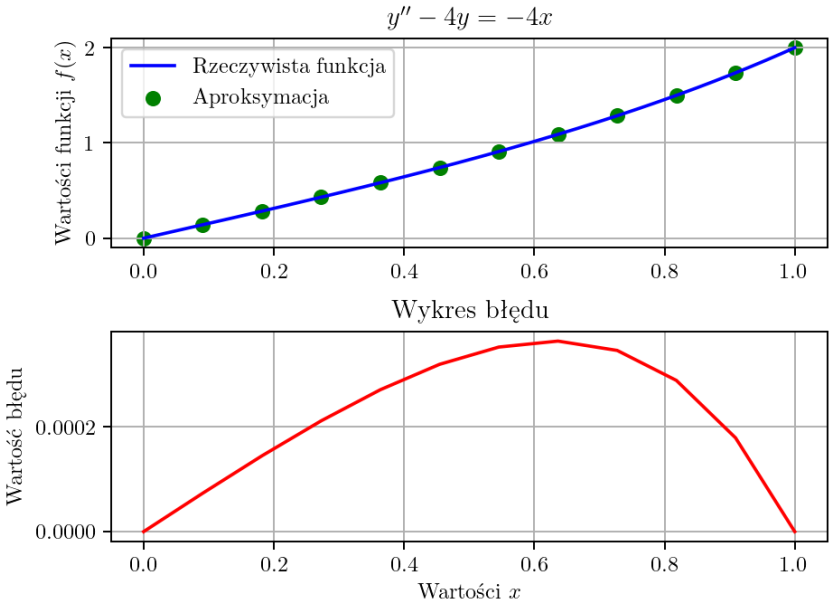
Mamy równanie różniczkowe II rzędu postaci:

$$\begin{cases} y'' - 4y = -4x \\ y|_{x=0} = 0 \\ y|_{x=1} = 2 \end{cases} \qquad x \in [0,1]$$

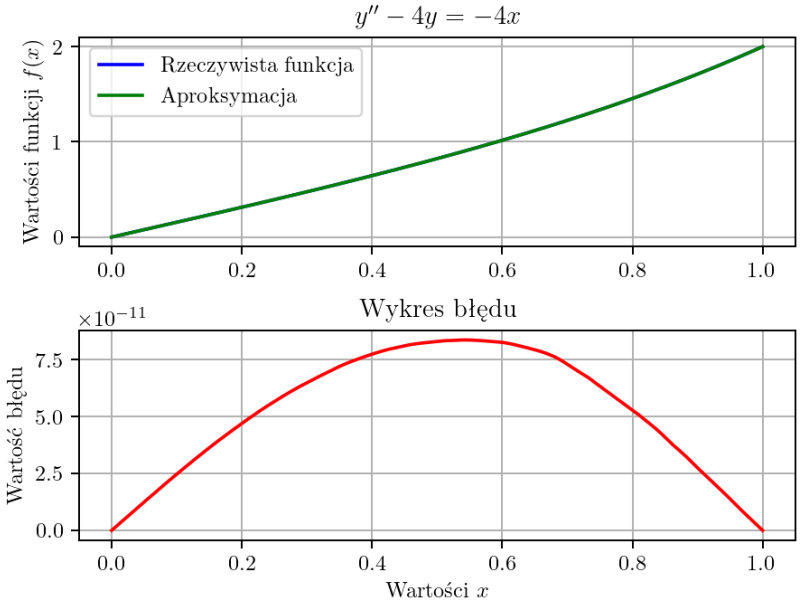
Rozwiązanie analityczne:

$$y = e^2(e^4 - 1)^{-1}(e^{2x} - e^{-2x}) + x$$

Rozwiązanie numeryczne dla n = 10 przedstawione jest na rysunku 18. Rysunek 19 zawiera rozwiązanie dla n = 10000.




**Rysunek 18:** Wykres z rozwiązaniem dla $n = 10$




**Rysunek 19:** Wykres z rozwiązaniem dla $n = 10000$

## 5.2 Zadanie 2

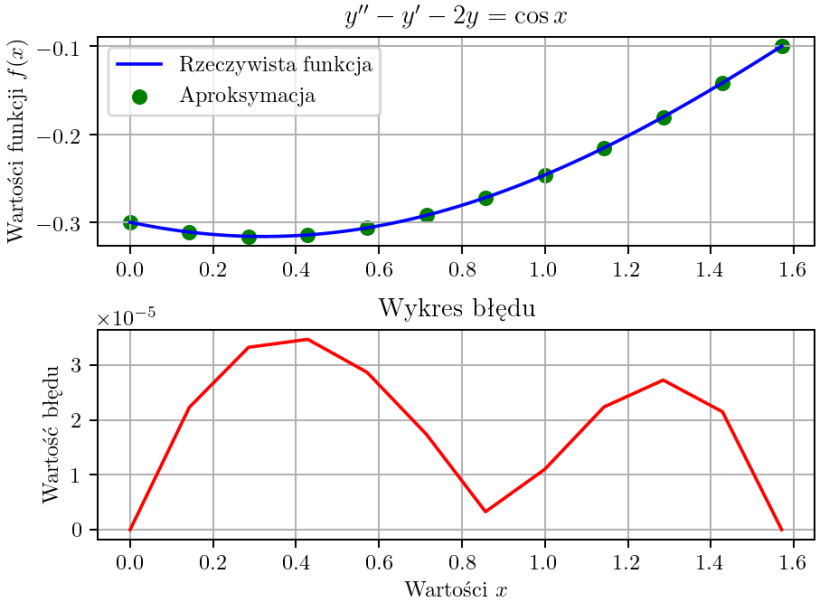
Mamy równanie różniczkowe II rzędu postaci:

$$\begin{cases} y'' - y' - 2y = \cos x \\ y|_{x=0} = -\frac{3}{10} \\ y|_{x=\frac{\pi}{2}} = -\frac{1}{10} \end{cases} \qquad x \in [0, \frac{\pi}{2}]$$

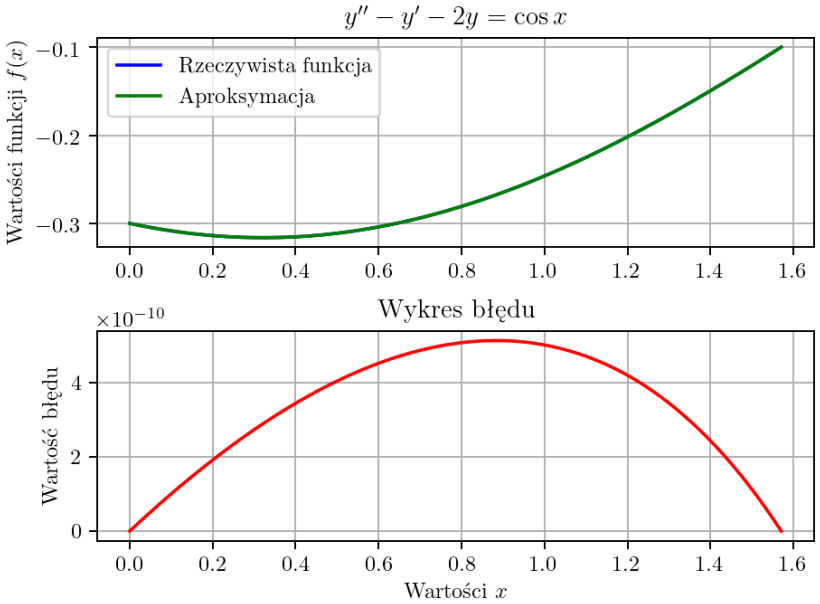
Rozwiązanie analityczne:

$$y = -\frac{\sin x + 3\cos x}{10}$$

Rozwiązanie numeryczne dla n = 10 przedstawione jest na rysunku 20. Rysunek 21 zawiera rozwiązanie dla n = 10000.




**Rysunek 20:** Wykres z rozwiązaniem dla $n = 10$




**Rysunek 21:** Wykres z rozwiązaniem dla $n = 10000$

## 5.3 Zadanie 3

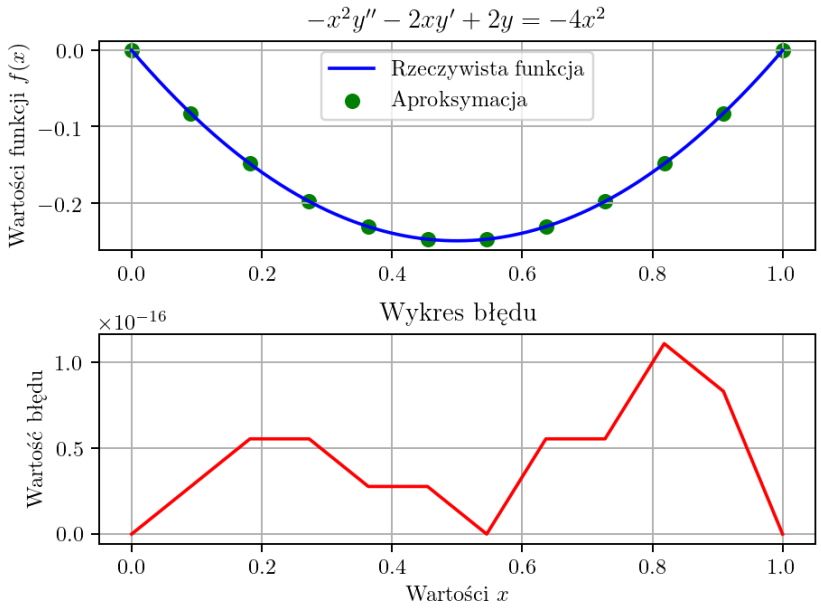
Mamy równanie różniczkowe II rzędu postaci:

$$\begin{cases} -x^2y'' - 2xy' + 2y = -4x^2 \\ y|_{x=0} = 0 \\ y|_{x=1} = 0 \end{cases} \qquad x \in [0,1]$$

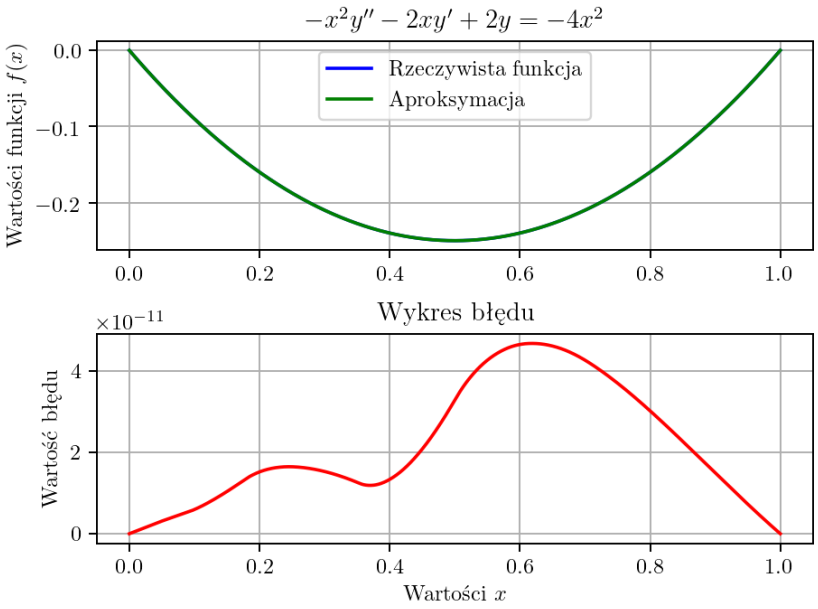
Rozwiązanie analityczne:

$$y = x^2 - x$$

Rozwiązanie numeryczne dla n = 10 przedstawione jest na rysunku 22. Rysunek 23 zawiera rozwiązanie dla n = 10000.




**Rysunek 22:** Wykres z rozwiązaniem dla $n = 10$




**Rysunek 23:** Wykres z rozwiązaniem dla $n = 10000$

## 6 Porównanie błędów

Zestawienie błędów z poprzednich zadań znajduje się w tabeli 2. Błąd obliczany był poprzez normę maksimum. Oznacza to, że z całego wektora błędów brana pod uwagę jest wartość największa.

**Tabela 2:** Zestawienie błędów z zadań

| $n$ | Zadanie 1.1 | Zadanie 1.2 | Zadanie 2.1 | Zadanie 2.2 | Zadanie 2.3 |
|---|---|---|---|---|---|
| 5 | 4,84E-01 | 3,33E-16 | 1,21E-03 | 1,07E-04 | 5,55E-17 |
| 10 | 1,21E-01 | 3,33E-16 | 3,64E-04 | 3,47E-05 | 1,11E-16 |
| 50 | 5,98E-03 | 1,67E-15 | 1,70E-05 | 1,65E-06 | 1,05E-15 |
| 100 | 1,52E-03 | 1,80E-15 | 4,34E-06 | 4,20E-07 | 2,89E-15 |
| 500 | 6,19E-05 | 2,53E-14 | 1,76E-07 | 1,71E-08 | 7,32E-14 |
| 1000 | 1,55E-05 | 1,49E-13 | 4,42E-08 | 4,27E-09 | 5,47E-13 |
| 5000 | 6,21E-07 | 1,74E-12 | 1,97E-09 | 2,40E-10 | 2,48E-11 |
| 10000 | 1,55E-07 | 2,06E-12 | 8,37E-11 | 5,13E-10 | 4,67E-11 |

Zastanawiający jest fakt, że dla niektórych zadań błąd rósł zamiast maleć.

# 7 Wnioski

Metoda Thomasa rozwiązywania układów trójprzekątniowych została zaimplementowana w sposób poprawny. Wnioskuję to poprzez fakt, że otrzymywałem te same błędy niezależnie od metody.

Niektóre wykresy błędów zachowywały kształt mimo zmiany parametru $n$. Jedyne co się zmieniało to rząd wielkości błędu. Założyłem, że błąd musi być zawsze dodatni, więc na wykresy została nałożona wartość bezwzględna. Przez to wydawać się może, że błąd w pewnych miejscach jest zerowy. Nie jest tak gdyż na przykład na rysunku 15. tam mamy do czynienia z błędem, którego funkcja przypomina pewne złożenie funkcji sinus.

Zadanie 1.1 jest wolniej zbieżne przez postać funkcji $y$. Jak widać na rysunku 13 funkcja rzeczywista jest bardzo zmienna, przez co aproksymacja schematami różnicowymi, które są wykorzystywane, jest trudniejsza niż w przypadku pozostałych zadań, gdzie funkcje są mniej strome i mniej zmienne.

Anomalie występujące w zadaniu 1.2 oraz 2.3 mogą być spowodowane postacią rozwiązania. Te zadania jako jedyne mają postać wielomianową jako rozwiązanie. Kluczowy tutaj jest fakt, że schematy różnicowe korzystają z rozwinięcia funkcji w szereg Taylora. Jest on wykorzystywany do przedstawiania dowolnej funkcji w postaci wielomianu. Zatem powinien on być idealny do aproksymacji rozwiązania, który jest wielomianem. Dokładnie to widzimy w tabeli 2. Po przeprowadzeniu paru testów dla zadania 1.2, którego rozwiązaniem jest wielomian 3 stopnia, po daniu $n = 3$ błąd był jeszcze niższy, a dla zadania z wielomianem 2 stopnia dla $n = 3$ program wskazał błąd równy 0. Taki błąd jest jak najbadziej poprawny, ponieważ przy rozwijaniu funkcji w szereg Taylora przy pochodnej 2 rzędu występuje czynnik kwadratowy. Błędy przy obliczaniu wartości funkcji przy większej liczbie $n$ może wynikać z nakładających się błędów numerycznych.