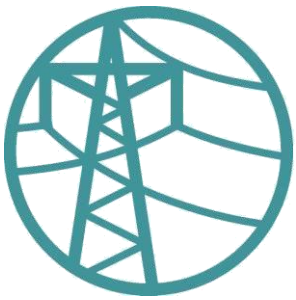


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Kamil Bąk

Implementacja metody elementów skończonych w języku Python

Metody FEM w Robotyce
Sprawozdanie z Laboratorium

Prowadzący zajęcia:
dr inż. prof. PRz Mariusz Borkowski

Rzeszów, 2021

Spis treści

1. Tematyka Sprawozdania

W sprawozdaniu opisze i wyjaśnię kod programu napisany w języku Python , który działa dla podstawowego, prostego przypadku użycia metody elementów skończonych, metody FEM. W przypadku tym z góry zakładamy że współczynniki c i f są zerowe, co powoduje ze równanie upraszcza się do jednorodnego równania Laplace'a. Zakładamy również że na obu brzegach dany mamy warunek Dirichleta, tzn. wartość funkcji na brzegu.

2. Opis Funkcji Pomocniczych

W celu zwiększenia przejrzystości kodu, niektóre z działań zapisano w funkcjach. Dzięki temu w funkcji głównej programu wystarczy jedynie umieścić jednolinijkowe wywołanie funkcji, zamiast kilku lub kilkunastu linii kodu. Ponadto raz utworzoną funkcje można wywoływać kilkukrotnie, co wpływa na zmniejszenie objętości kodu. W tym rozdziale opiszę funkcje, przedstawię ich kod, oraz działanie.

2.1. ManualGeometryDefinition()

Opis funkcji

Geometria problemu może być podana bezpośrednio przez ręczenie utworzenie dwóch tablic i zapisanie ich w pamięci programu.

Pierwsza z macierzy nazwana „Tab_wezlow" przechowuje informacje o węzłach. W pierwszej kolumnie macierzy umieszczone są indeksy węzłów, w drugiej kolumnie natomiast znajdują się współrzędne węzłów na osi x . Każdy wiersz macierzy odpowiada za jeden węzeł.
Druga z macierzy nazywa się „Tab_Elementow" i zawiera informacje na temat elementów. W pierwszej kolumnie macierzy znajdują się indeksy elementów, w drugiej kolumnie indeks globalny węzła który jest początkowym węzłem elementu, w trzeciej kolumnie natomiast indeks globalny końcowego węzła elementu.

Zmienna War_Brzeg przechowuje informacje na temat warunków brzegowych. Informacje są podane za pomocą listy, która składa się z dwóch słowników, każdy słownik opisuje jeden warunek brzegowy. Słowniki składają się z 3 elementów, pierwszy element informuje nas o indeksie węzła na którym podany jest warunek brzegowy, drugi element to typ warunku. Litera „D"

oznacza że jest to warunek Dirichleta, znamy więc wartość funkcji na brzegu, właśnie ta wartość zapisana jest w trzecim elemencie słownika.

Kod

```
def ManualGeometryDefinition():
    Tab_wezlow = np.array([[1, 0],
                           [2, 1],
                           [3, 0.5],
                           [4, 0.75]])

    Tab_Elementow = np.array([[1, 1, 3],
                           [2, 4, 2],
                           [3, 3, 4]])

    War_Brzeg = [{"ind": 1, "typ": 'D', "wartosc": 1},
          {"ind": 2, "typ": 'D', "wartosc": 2}]

    return Tab_wezlow, Tab_Elementow, War_Brzeg
```

Wywołanie funkcji i wyświetlenie wyników

```
WEZLY, ELEMENTY, WB = ManualGeometryDefinition()
print(WEZLY)
print(ELEMENTY)
print(WB)
```

```
C:\Users\DOM\AppData\Local\Programs\Python\Python39\python.exe D:/GitProjects/LastProject/main.py
[[1.   0.  ]
 [2.   1.  ]
 [3.   0.5 ]
 [4.   0.75]]
[[1 1 3]
 [2 4 2]
 [3 3 4]]
[{'ind': 1, 'typ': 'D', 'wartosc': 1}, {'ind': 2, 'typ': 'D', 'wartosc': 2}]
```

2.2. AutomaticGeometryDefinition()

Opis funkcji

Definiowanie Geometrii przez ręczne tworzenie tablic byłoby kłopotliwe, szczególnie w procesie zwiększania liczby elementów w celu poprawienia dokładności obliczeń. Dlatego też, dobrym rozwiązaniem było napisanie funkcji która tworzy macierze będące definicją geometrii automatycznie. Funkcje tą napisałem samodzielnie w ramach jednego z laboratoriów .

Funkcja jako parametry przyjmuje położenie, pierwszego i ostatniego węzła, oraz zadaną liczbę węzłów. Przykładowo podając do niej wartości (0,1,5) utworzy one geometrie składającą się

z 5 węzłów z których pierwszy leży w punkcie 0 na osi x, a ostatni w punkcie 1, odstępy między węzłami będą równe.

Kod

```
def AutomaticGeometryDefinition (x_a,x_b, n):
    # l_elementow = n-1
    odstep = (x_b - x_a) / (n - 1) #wyliczenie odstepu
    wezly = np.array([1, x_a]) #pierwszy wiersz tablicy wezly
    elementy = np.array([1,1,2]) #pierwszy wiersz tablicy elementy
    for i in range(1, n, 1):
       wezly = np.block([
            [wezly ],
            [i+1, i * odstep+x_a],
        ])
    for j in range(2, n,1):
        elementy = np.block([
            [elementy],
            [j,j,j+1]])
    return wezly,elementy
```

Wywołanie funkcji i wyświetlenie wyników

```
    x_a = 0
    x_b = 1
    n = 5
    WEZLY, ELEMENTY = AutomaticGeometryDefinition (x_a, x_b, n)
    # warunki brzegowe
    WB = [{"ind": 1, "typ": 'D', "wartosc": 1},
          {"ind": n, "typ": 'D', "wartosc": 2}]

    print(WEZLY)
    print('\n')
    print(ELEMENTY)
    print('\n')
    print(WB)
```

```
[[1.   0.  ]
 [2.   0.25]
 [3.   0.5 ]
 [4.   0.75]
 [5.   1.  ]]


[[1 1 2]
 [2 2 3]
 [3 3 4]
 [4 4 5]]


[{'ind': 1, 'typ': 'D', 'wartosc': 1}, {'ind': 5, 'typ': 'D', 'wartosc': 2}]
[{'ind': 1, 'typ': 'D', 'wartosc': 1}, {'ind': 5, 'typ': 'D', 'wartosc': 2}]
```

2.3. ShowGeometry()

Opis funkcji

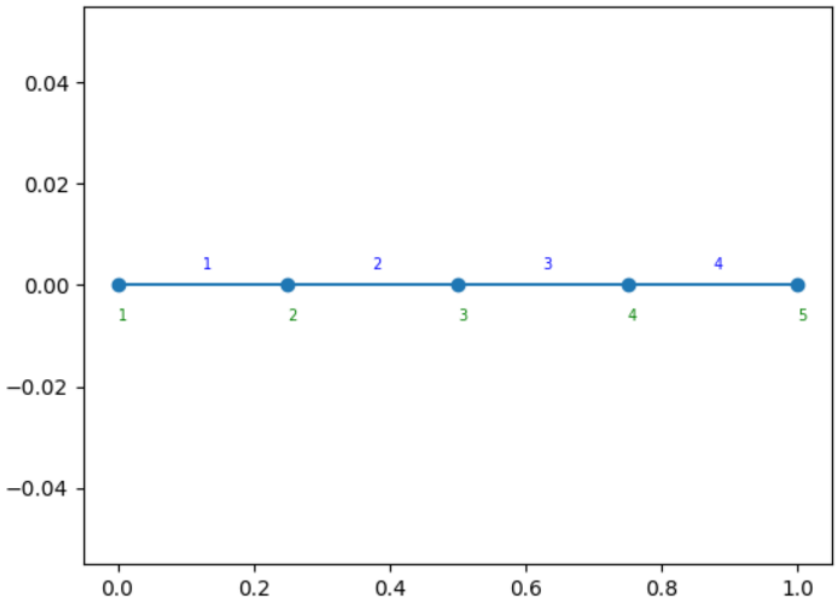
Funkcja rysująca geometrie, również została utworzona przeze mnie na jednych z zajęć. Funkcja przyjmuje jako parametr tablice węzłów, a następnie rysuje geometrie. Na rysunku na zielono podpisane są globalne indeksy węzłów, a na niebiesko indeksy elementów. Funkcja używa funkcji z pakietu matplotlib.

Kod

```
def ShowGeometry (tab_w):
    y = np.zeros(tab_w.shape[0])
    plt.plot(tab_w[:, 1], y, marker='o') #wyrysowanie elementow
    for i in range(0, np.size(y), 1): #podpis wezlow
        plt.text(x=tab_w[i, 1], y=y[i]-0.007, s=int(tab_w[i,
0]),fontsize=7,color ='green' )
        # plt.text(x=tab_w[i, 1]+, y=y[i] - 0.007, s=tab_w[i, 0], fontsize=12,
color='green')
    for i in range(0, np.size(y) - 1, 1): #podpis elementow
        plt.text(x=(tab_w[i, 1] + tab_w[i + 1, 1]) / 2, y=y[i] + 0.003, s=int(i
+ 1),fontsize=7, color='blue')
    plt.show()
```

Wywołanie funkcji i wyświetlenie wyników

```
ShowGeometry (WEZLY)
```



2.4. Allocation()

Opis funkcji

Funkcja Allocation() jako parametr przyjmuje liczbę węzłów, co jednocześnie jest rozmiarem tworzonych macierzy. Funkcja zwraca dwie tablice zer o rozmiarach n x n, oraz n x 1 . Do tworzenia tych tablic korzystamy z funkcji zeros, która tworzy tablice o rozmiarach podanych jako parametry.

Kod

```
def Allocation(n):

    A = np.zeros([n,n])
    b = np.zeros([n,1])

    return A,b
```

Wywołanie funkcji

```
    A, b = Allocation(n)

    print(A)
    print(b)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

2.5. BaseFunctions()

Opis funkcji

Zdefiniowana funkcja wykorzystuje wyrażenia lambda, które pozwalają nam definiować funkcje zapisując ich wzór od zmiennej x. Funkcje lambda działają wektorowo, tzn. że obliczą i zwrócą wartość dla każdego z elementów tablicy . Funkcja jako parametr przyjmuje zmienną stop_fun_baz która oznacza stopień wymaganych funkcji kształtu. Pierwszym elementem zwracanym przez funkcję jest funkcja, (n+1)- elementowa lista (tupla) funkcji bazowych stopnia stop_fun_baz . Drugim elementem zwracanym przez funkcję jest pochodna, (n+1)- elementowa lista (tupla) pochodnych funkcji bazowych stopnia n. Funkcje bazowe nie muszą być liniowe, mogą być funkcjami dowolnego stopnia. Dzięki zastosowaniu funkcji *BaseFunctions()* w takiej postaci program nie traci na ogólności.

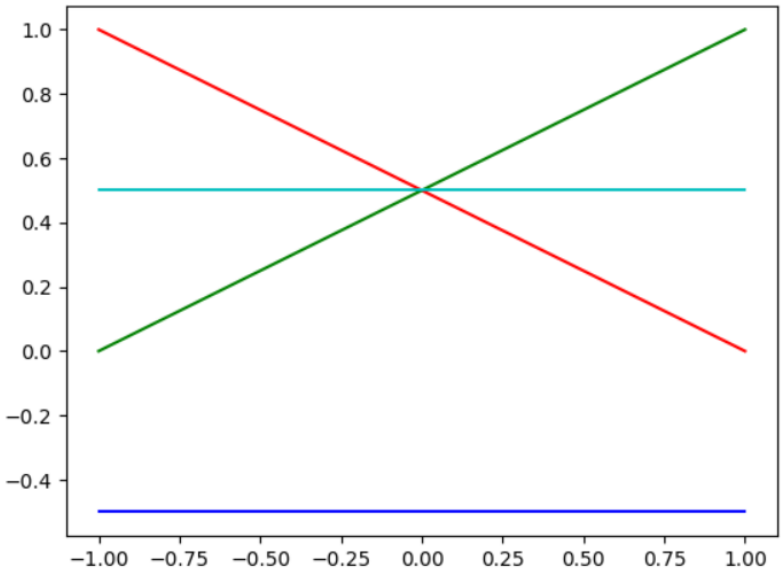
Kod

```
def BaseFunctions(stop_fun_baz):

    if stop_fun_baz == 0:
        funkcja = (lambda x: 1 + 0 * x)
        pochodna = (lambda x: 0 * x)

    elif stop_fun_baz == 1:

        funkcja = (lambda x: -1 / 2 * x + 1 / 2, lambda x: 0.5 * x + 0.5)
        pochodna = (lambda x: -1 / 2 + 0 * x, lambda x: 0.5 + 0 * x)


    else:
        raise Exception("Nieoczekiwany stop_fun_baz w BaseFunctions().")

    return funkcja, pochodna
```

Wywołanie funkcji i wyświetlenie wyników

Poniżej znajduje się wywołanie funkcji dla stopnia funkcji bazowych równych 1. Następnie otrzymane wyniki są wyświetlane za pomocą funkcji z pakietów numpy i matplotlib.

```
stopien_fun_bazowych = 1
phi, dphi = BaseFunctions(stopien_fun_bazowych)

x = np.linspace(-1,1, 101)
plt.plot(x, phi[0](x), 'r' )
plt.plot(x, phi[1](x), 'g' )
plt.plot(x, dphi[0](x), 'b' )
plt.plot(x, dphi[1](x), 'c' )
plt.show()
```



2.6. Aij()

Opis funkcji

Macierz Ml należy wypełnić wartościami związanymi z danym elementem. Przy obliczaniu tych wartości używamy funkcji Aij(). Reprezentuje ona funkcje podcałkową, której wzór widoczny jest poniżej.

$$\int_{x_a^k}^{x_b^k} -\phi^{(k1)\prime}\phi^{(k1)\prime} + c\phi^{(k1)}\phi^{(k1)}\,\mathrm{d}x$$

Funkcja Aij() jako parametry przyjmuje pochodna_fun_i, pochodna_fun_j pochodne funkcji kształtu związanych z i-tym oraz j- tym węzłem, przyjmuje również funkcja_i, funkcja_j funkcje kształtu związane z i-tym oraz j- tym węzłem, oraz parametr c którym na początku funkcji głównej programu definiujemy jako 0.

Funkcja dla zadanych parametrów ma zwrócić wartość funkcji podcałkowej.

Kod

```
def Aij(pochodna_fun_i, pochodna_fun_j, c, funkcja_i, funkcja_j):

    funkcja_podcalkowa = lambda x: -pochodna_fun_i(x) * pochodna_fun_j(x) + c *
funkcja_i(x) * funkcja_j(x)

    return funkcja_podcalkowa
```

2.7. ShowSolution()

Opis funkcji

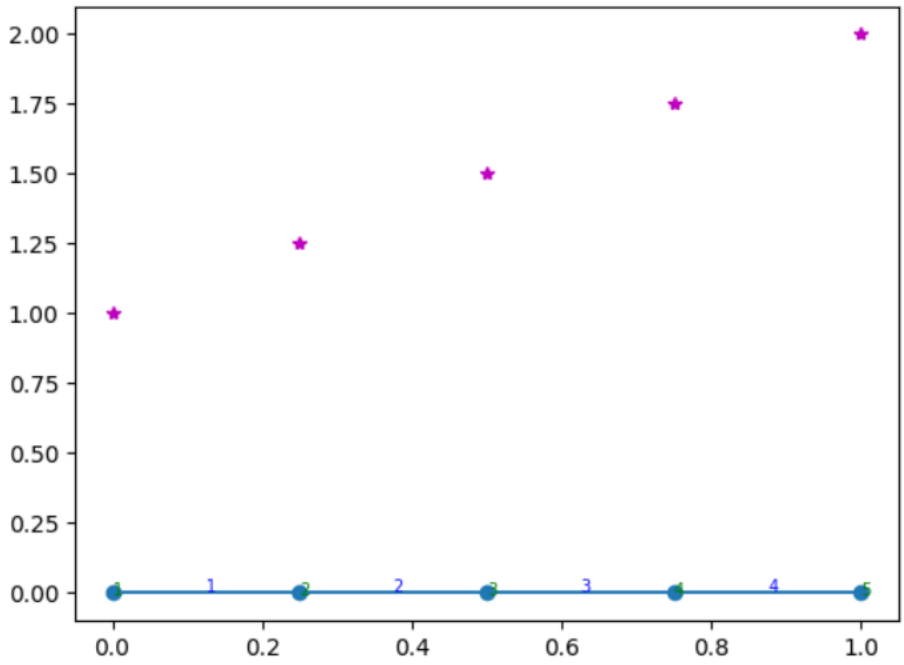
Do wyrysowania rozwiązania używana jest funkcja ShowSolution(). Funkcja wykorzystuje funkcje *ShowGeometry()*, a następnie na wyrysowaną geometrie nakłada rozwiązanie , które podane jest do funkcji w postaci wektora u. Do wyrysowania rozwiązania korzystamy z funkcji pakietu matplotlib.

Kod

```
def ShowSolution (Tab_wezlow, rozwiazanie):


    ShowGeometry(Tab_wezlow)

    x = Tab_wezlow[:, 1]

    plt.plot(x, rozwiazanie, 'm*')
    plt.show()
```

Wywołanie funkcji i wyświetlenie wyników

```
ShowSolution(WEZLY, u)
```



3. Funkcja Główna

Import bibliotek

Jeszcze przed funkcją główną importujemy potrzebne nam biblioteki. Biblioteki numpy i scipy posłużą nam do obliczeń matematycznych, a biblioteka matplotlib do rysowania wykresów funkcji.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as spint
```

PREPROCESSING

Rozpoczęcie funkcji głównej

Rozpoczynamy pisani funkcji głównej, na początku zakładamy że parametr c i parametr f są równe zero, dzięki temu równanie uprości się do równania Laplace'a. W naszym przypadku przyjmujemy również że na brzegach nie ma warunków Neumana, a warunki Dirichleta oznacza to że znana jest wartość funkcji na brzegu, a nie jak w przypadku warunku Neumama wartość pochodnej funkcji, tego założenia użyjemy w późniejszej części kodu.

Wyrażenie lambda stosowane jest w celu zadeklarowania funkcji f. Funkcja zadeklarowana powyżej dla każdej wartości wejściowej zwróci wynik 0.

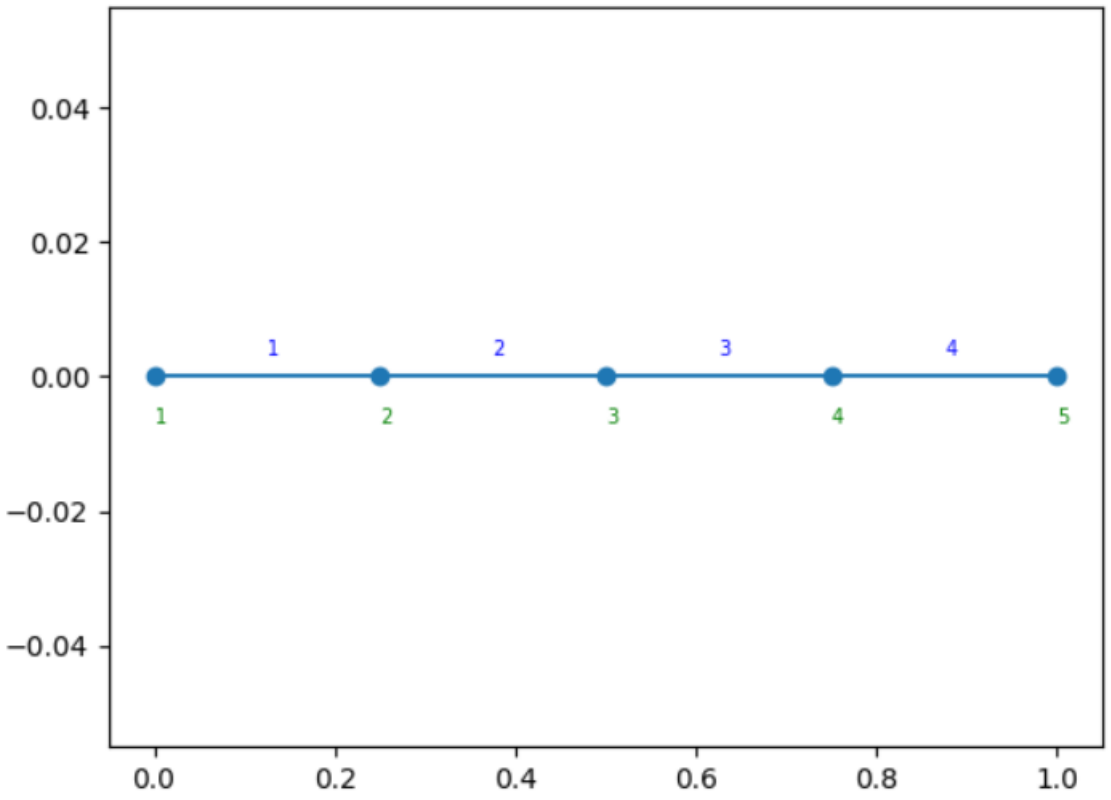
```
if __name__ == '__main__':

    c = 0
    f = lambda x: 0 * x  # wymuszeniee
```

Definicja geometrii

Przedstawienie geometrii problemu można wykonać na dwa sposoby. Pierwszy sposób to utworzenie tablic reprezentujących węzły i elementy w sposób bezpośredni, działania takie są wykonane w funkcji ManualGeometryDefinition() z podrozdziału 1.1.1. Drugim sposobem jest automatyczne wygenerowanie geometrii za pomocą funkcji AutomaticGeometryDefinition () przedstawionej w podrozdziale 1.1.2. Funkcja ta po podaniu położenia pierwszego i ostatniego parametru, oraz zadanej liczby węzłów sama ustali w jakich miejscach powinny znajdować się

pozostałe węzły. Jak widać poniżej w kodzie wybrano drugi sposób. Pierwszy ze sposobów został skomentowany aby nie przeszkadzać w rozwiązaniu. Po zdefiniowaniu geometrii warto ją wyrysować aby upewnić się że wszystko przebiegło pomyślnie. Robimy to za pomocą funkcji ShowGeometry () z podrozdziału 1.1.3.

```
# zadefiniowana przez reczne utworzenie tabel

# WEZLY, ELEMENTY, WB = ManualGeometryDefinition()
# n = np.shape(WEZLY)[0]

#Automatyczne wygenerowanie geometrii

x_a = 0
x_b = 1
n = 5
WEZLY, ELEMENTY = AutomaticGeometryDefinition(x_a, x_b, n)
# warunki brzegowe
WB = [{"ind": 1, "typ": 'D', "wartosc": 1},
      {"ind": n, "typ": 'D', "wartosc": 2}]

ShowGeometry (WEZLY)
```



Alokacja pamięci

Należy dokonać alokacji pamięci potrzebnej na macierz A i wektor prawej strony b, zadanie to wykonuje funkcja Allocation(), która opisana jest w podrozdziale 1.1.4.

```
A, b = Allocation(n)
```

Definicja funkcji bazowych i ich pochodnych

Definicje funkcji bazowych i ich pochodnych przygotowano w funkcji BaseFunctions(), opis funkcji znajduje się w podrozdziale 1.1.5. Przy wywołaniu funkcji należy określić stopień funkcji bazowej, w naszym przypadku stopień jest równy 1.

```
stopien_fun_bazowych = 1
phi, dphi = BaseFunctions(stopien_fun_bazowych)
```

PROCESSING

Deklaracja zmiennych

Za pomocą wyrażenia np.shape pobieramy liczbę wierszy macierzy, która jest jednoznaczna z liczbą elementów, zapisujemy tą wartość i tworzymy pętle której iterator l będzie iterował po wszystkich elementach.

W celu zabezpieczenia się przed pomyłkami związanymi z tym ze w pythonie pierwsza kolumna lub wiersz mają indeks 0, tworzymy zmienną Glob_Ind_ele i w niej zapisujemy globalny numer elementu. Podobnie tworzymy zmienne ind_Wezel_pocz i ind_Wezel_kon, a następnie zapisujemy do nich globalne indeksy węzłów tworzących dany element. Zmienne te zapisujemy w macierzy indGlobalneWezlow.

Następnie tworzymy macierz lokalną Ml. Macierz ta na początku ma być wypełniona zerami, a jej rozmiar zależy od stopnia funkcji bazowych.

```
ElemsNumber = np.shape(ELEMENTY)[0]

for l in np.arange(0, ElemsNumber):
    Glob_Ind_ele = ELEMENTY[l, 0]
    ind_Wezel_pocz = ELEMENTY[l, 1]  # indeks wezla poczatkowego elemntu ee
    ind_Wezel_kon = ELEMENTY[l, 2]  # indeks wezla koncowego elemntu ee
    indGlobalneWezlow = np.array([ind_Wezel_pocz, ind_Wezel_kon])

    Ml = np.zeros([stopien_fun_bazowych + 1, stopien_fun_bazowych + 1])
```

Obliczanie macierzy lokalnej

Obliczenie macierzy globalnej A wymaga wyliczenia współczynników macierzy lokalnej dla każdego z elementów i przypisania ich do odpowiednich pozycji macierzy globalnej.

Do obliczenia wartości macierzy lokalnej Ml korzystamy z funkcji Aij(), której opis znajduję się w podrozdziale 1.1.6. Zwraca ona zdefiniowaną funkcję podcałkową.

Kiedy otrzymamy już wartości funkcji podcałkowych kolejnym krokiem jest całkowanie. Całki w oprogramowaniu python realizujemy za pomocą pakietu *scipy.integrate* a dokładniej funkcji *quad* do funkcji tej podajemy jako parametry funkcje podcałkową i przedział całkowania, zwraca ona dwie wartości, pierwsza wartość z indeksem 0 jest wynikiem całkowania, druga to oszacowanie błędów. Nas interesuje ta pierwsza wartość.

Wynik należy przemnożyć przez Jakobian J w celu zgodności wartości całki na przedziale unormowanym i całki na danym elemencie. Dla funkcji jednowymiarowych Jakobianem jest połowa przedziału.

Działania powtarzamy czterokrotnie, dla każdego z elementów lokalnej macierzy Ml, definiując dla każdego z elementów indeksy m i n.

```
        x_a = WEZLY[ind_Wezel_pocz - 1, 1]  #indeksy pythonowe
        x_b = WEZLY[ind_Wezel_kon - 1, 1]

        J = (x_b - x_a) / 2 #Jakobian

        m = 0
        n = 0
        Ml[m, n] = J * spint.quad(Aij(dphi[m], dphi[n], c, phi[m], phi[n]), -1,
1)[0]

        m = 0
        n = 1
        Ml[m, n] = J * spint.quad(Aij(dphi[m], dphi[n], c, phi[m], phi[n]), -1,
1)[0]

        m = 1
        n = 0
        Ml[m, n] = J * spint.quad(Aij(dphi[m], dphi[n], c, phi[m], phi[n]), -1,
1)[0]

        m = 1
        n = 1
        Ml[m, n] = J * spint.quad(Aij(dphi[m], dphi[n], c, phi[m], phi[n]), -1,
1)[0]
```

Agregacja wyników w macierzy globalnej

Następnie należy zająć się agregacją tablicy globalnej, tzn umieszczeniem elementów z tablic lokalnych Ml w tablicy globalnej. Do tego celu posłuży nam funkcja *ix_()* z pakietu numpy.

Wzory macierzy globalnej widoczne poniżej są w postaci, w której dodajemy wartości z macierzy lokalnych do wcześniejszej postaci macierzy globalnej. Z tego względu nie możemy

zastosować po prostu przypisania elementów macierzy Ml do macierzy A. Należy tak jak poniżej zastosować dodawanie.

1. inicjalizacja

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2. iteracja 1

$$\mathbf{M}^{(1)} = \begin{bmatrix} M_{11}^{\,1} & M_{12}^{\,1} \\ M_{21}^{\,1} & M_{22}^{\,1} \end{bmatrix} = \begin{bmatrix} M_{11} & M_{13} \\ M_{31} & M_{33} \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} M_{11}^{\,1} & 0 & M_{12}^{\,1} & 0 \\ 0 & 0 & 0 & 0 \\ M_{21}^{\,1} & 0 & M_{22}^{\,1} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3. iteracja 2

$$\mathbf{M}^{(2)} = \begin{bmatrix} M_{11}^{\,2} & M_{12}^{\,2} \\ M_{21}^{\,2} & M_{22}^{\,2} \end{bmatrix} = \begin{bmatrix} M_{44} & M_{42} \\ M_{24} & M_{22} \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} M_{11}^{\,1} & 0 & M_{12}^{\,1} & 0 \\ 0 & M_{22}^{\,2} & 0 & M_{21}^{\,2} \\ M_{21}^{\,1} & 0 & M_{22}^{\,1} & 0 \\ 0 & M_{12}^{\,2} & 0 & M_{11}^{\,2} \end{bmatrix}$$

4. iteracja 3

$$\mathbf{M}^{(3)} = \begin{bmatrix} M_{11}^{\,3} & M_{12}^{\,3} \\ M_{21}^{\,3} & M_{22}^{\,3} \end{bmatrix} = \begin{bmatrix} M_{33} & M_{34} \\ M_{43} & M_{44} \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} M_{11}^{\,1} & 0 & M_{12}^{\,1} & 0 \\ 0 & M_{22}^{\,2} & 0 & M_{21}^{\,2} \\ M_{21}^{\,1} & 0 & M_{22}^{\,1} + M_{11}^{\,3} & M_{12}^{\,3} \\ 0 & M_{12}^{\,2} & M_{21}^{\,3} & M_{11}^{\,2} + M_{22}^{\,3} \end{bmatrix}$$

```
        A[np.ix_(ind_Glob_Wezlow - 1, ind_Glob_Wezlow - 1)] = \
            A[np.ix_(ind_Glob_Wezlow - 1, ind_Glob_Wezlow - 1)] + Ml

        # print(Ml)
        # print('\n')
    print(A)
    print('\n')
```

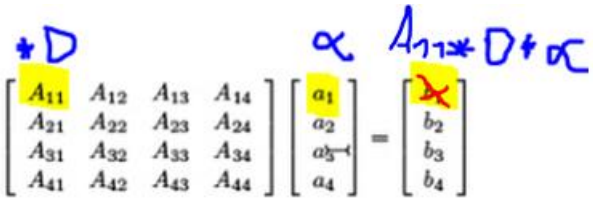
Macierz globalna A dla geometrii automatycznie utworzonej przez funkcję (5 węzłów pierwszy w 0, ostatni w 1) ma następującą postać.

```
[[-0.0625  0.0625  0.      0.      0.    ]
 [ 0.0625 -0.125   0.0625  0.      0.    ]
 [ 0.      0.0625 -0.125   0.0625  0.    ]
 [ 0.      0.      0.0625 -0.125   0.0625]
 [ 0.      0.      0.      0.0625 -0.0625]]
```

Uwzględnienie warunków brzegowych

W przypadku warunków Dirichleta znamy wartości funkcji w węźle początkowym i w węźle końcowym. Możemy więc zastosować „sztuczkę” która wymusi przyjęcie przez dany element zadanej wartości. Przyjmijmy że a1 to wartość węzła początkowego i wynosi alfa. Mnożymy element w wierszu pierwszym stojący na przekątnej głównej macierzy A przez jakąś

bardzo dużą liczbę, następnie element b1 zastępujemy elementem z macierzy A stojącym na przekątnej głównej pomnożonym przez tą samą dużą liczbę oraz przez zadaną wartość alfa. Opisane działania przedstawia ilustracja poniżej.



W poniższym kodzie zastosowano tą sztuczkę do wymuszenia wartości znanych z warunków brzegowych w rozwiązaniu.

Typ warunku sprawdzany jest za pomocą instrukcji warunkowej if, litera „D” oznacza warunek Dirichleta. Następnie odczytywany jest indeks globalny węzła początkowego i przypisywany do zmiennej iwp . Odczytywana jest również wartość funkcji określona w tym węźle przez warunek brzegowy. Pobrany Indeks węzła należy pomniejszyć o 1, ponieważ python zaczyna numeracje od 0, a my w naszej funkcji definiującej geometrie zaczynaliśmy numeracje od 1. Jako bardzo dużą liczbę definiujemy WZMACNIACZ o wartości 10 do potęgi 14. Dla węzła początkowego i końcowego wykonujemy te same czynności, uwzględniając jedynie różnicę w indeksach.

```
# UWZGLEDNIENIE WARUNKOW BRZEGOWYCH
    if WB[0]['typ'] == 'D':
        ind_wezla = WB[0]['ind']
        wart_war_brzeg = WB[0]['wartosc']

        iwp = ind_wezla - 1

        WZMACNIACZ = 10**14 # pewna duża wartosc

        b[iwp] = A[iwp,iwp]*WZMACNIACZ*wart_war_brzeg
        A[iwp, iwp] = A[iwp,iwp]*WZMACNIACZ


    if WB[1]['typ'] == 'D':
        ind_wezla = WB[1]['ind']
        wart_war_brzeg = WB[1]['wartosc']

        iwp = ind_wezla - 1

        WZMACNIACZ = 10**14

        b[iwp] = A[iwp,iwp]*WZMACNIACZ*wart_war_brzeg
        A[iwp, iwp] = A[iwp,iwp]*WZMACNIACZ
```

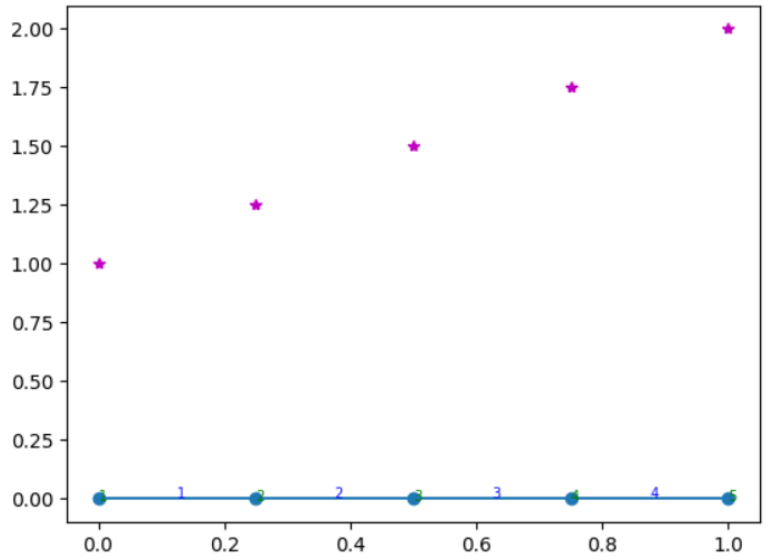
Rozwiązanie równania i wyrysowanie wyniku

W celu rozwiązania równania używamy funkcji *solve()* z pakietu numpy, podpakietu linalog. Jako parametry funkcji podajemy macierz globalną A i wektor prawej strony równania b.

Do wyrysowania wyników używamy natomiast funkcji ShowSolution(), która została dokładniej opisana w podrozdziale 1.1.7.

```
# Rozwiazanie ukl row lin
u = np.linalg.solve(A,b)
print(u)
ShowSolution(WEZLY, ELEMENTY, WB, u)
```

Poniżej przedstawiam wynik wykreślony przez funkcję ShowSolution().



4. Wzbogacenie działającego programu o własne, dodatkowe funkcje

4.1. Funkcja automatycznie generująca geometrie i funkcja rysująca geometrie

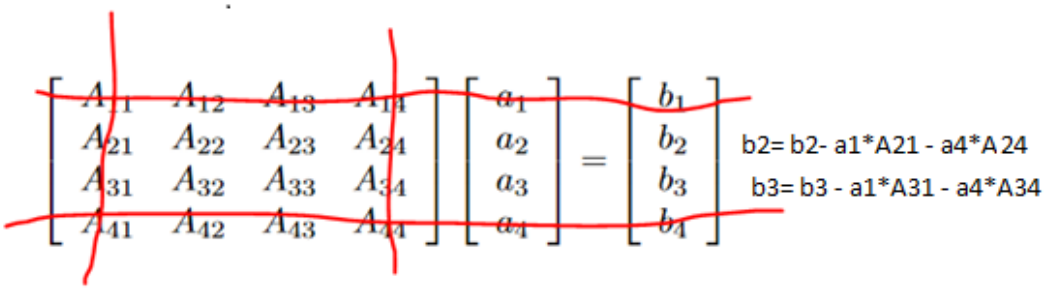
Funkcje ShowGeometry() i AutomaticGeometryDefinition() utworzyłem podczas jednych z zajęć laboratoryjnych, Pierwsza funkcja na podstawie podanej do niej macierzy reprezentującej węzły rysuje geometrie, tzn. rysuje węzły a następnie podpisuje różnymi kolorami globalne numery węzłów i elementów. Druga funkcja na podstawie podanych do niej parametrów (współrzędna pierwszego i ostatniego węzła, oraz ilość węzłów) tworzy reprezentujące geometrie problemu macierze węzłów i elementów, węzły utworzone za pomocą tej funkcji są w równych odległościach od siebie.

Funkcje zostały dokładnie opisane w podrozdziałach 2.2 i 2.3.

4.2. Implementacja warunku brzegowego Dirichleta – drugie podejście numeryczne

Opis

W przypadku warunków Dirichleta znamy wartości funkcji w węźle początkowym i w węźle końcowym. Możemy więc wykreślić wiersze i kolumny macierzy które są związane z tymi znanymi już wartościami w węzłach. W przypadku generowanym przez funkcję AutomaticGeometryDefinition() znamy wartości w węzłach które globalnie mają numer 1 i n. Wykreślamy więc pierwszy i n-ty wiersz, oraz pierwszą i n-tą kolumnę. Co przedstawiono poniżej. Warto zwrócić uwagę na to że przed wykreśleniem kolumn należy uwzględnić ich wartość względem parametru elementów wektora b. Przykładowo jeżeli znamy wartość a1 z poniższego rysunku to musimy zwrócić uwagę na to że trzeba zmienić wartość elementu b2 na b2= b2-a1*A21. Analogicznie wykonujemy takie działanie dla innych elementów wektora b, oprócz b1 który został wykreślony. Działania te zostały uwzględnione w kodzie w pętli for. Oprócz wykreślenia wierszy i kolumn z macierzy A, należy również pamiętać o zmniejszeniu liczby elementów wektora b.



Kod

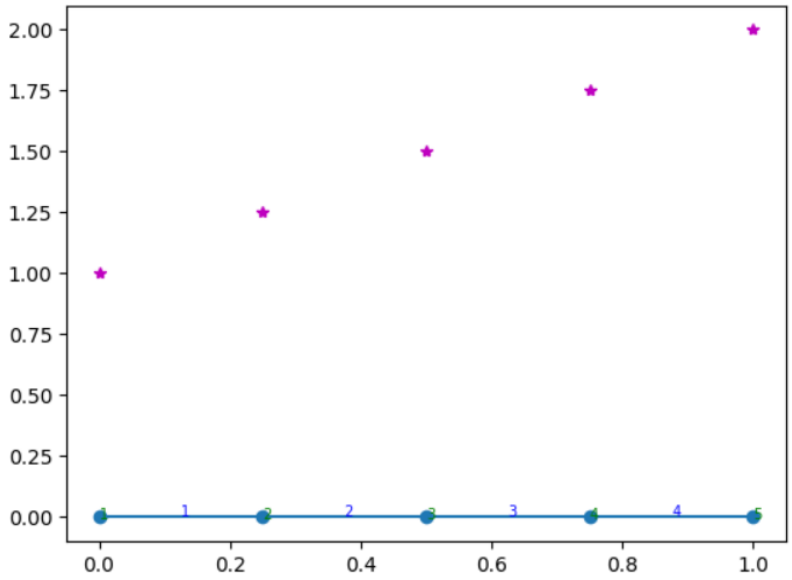
```
if (WB[0]['typ'] == 'D') & (WB[1]['typ'] == 'D'):

    ind_wezla_1 = WB[0]['ind']    # indeks globalny wezla pocz
    wart_war_brzeg_1 = WB[0]['wartosc'] # war wezla pocz

    ind_wezla_2 = WB[1]['ind'] # indeks globalny wezla kon
    wart_war_brzeg_2 = WB[1]['wartosc'] # war wezla kon

    iwp1 = ind_wezla_1 - 1 # indeksy zmienione na indeksy pythona
    iwp2 = ind_wezla_2 - 1

    A=np.delete(A,[iwp1,iwp2],0) # usuwanie wierszy z macierzy A


    b=np.delete(b,[iwp1,iwp2],0) # usuwanie wierszy z wektora b

    n_b = np.shape(b)[0] # wielkosc wektora b

    for i in np.arange(0, n_b): # zmiana wartosci wektora b
        b[i] = b[i] - A[i,iwp1] * wart_war_brzeg_1
        b[i] = b[i] - A[i, iwp2] * wart_war_brzeg_2

    A = np.delete(A, [iwp1, iwp2], 1) # usuwanie kolumn z wartosci A
```

Tak zmodyfikowaną wartość macierzy globalnej A podajemy do funkcji *solve()*, wynik jednak nie będzie zawierał wartości funkcji we wszystkich węzłach. Brakujące elementy to wycięte wcześniej, znane już wartości węzłów z warunków brzegowych. Dodałem je za pomocą funkcji *vstack ()*, z pakietu numpy.

```
   u = np.linalg.solve(A, b)
   u = np.vstack(([wart_war_brzeg_1], u, [wart_war_brzeg_2])) # dodanie wartosci
brzegowych funkcji do rozwiazania
    print(u)
```

Wynik dla zaimplementowanego w ten sposób warunku Dirichleta jest poprawny i pokrywa się z wynikiem uzyskanym w wyniku wcześniej stosowanej metody.

```
[[1.  ]
 [1.25]
 [1.5 ]
 [1.75]
 [2.  ]]
```



4.3. Funkcje bazowe 2 stopnia

Opis

Do funkcji BaseFunctions() opisanej w podrozdziale 2.5 dodałem fragment kodu, który tworzy funkcje bazowe stopnia drugiego. Funkcje te są wielomianami Lagrange'a wyprowadzonymi dla przedziału [-1,1]. Poprawność działania funkcji sprawdziłem poprzez wyrysowanie otrzymanych funkcji i ich pochodnych.

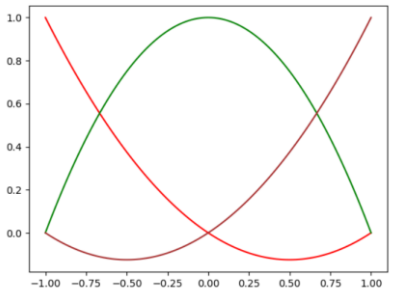
Kod

```
def BaseFunctions(stop_fun_baz):

    if stop_fun_baz == 0:
        funkcja = (lambda x: 1 + 0 * x)
        pochodna = (lambda x: 0 * x)

    elif stop_fun_baz == 1:

        funkcja = (lambda x: -1 / 2 * x + 1 / 2, lambda x: 0.5 * x + 0.5)
        pochodna = (lambda x: -1 / 2 + 0 * x, lambda x: 0.5 + 0 * x)

    elif stop_fun_baz == 2:
        funkcja =(lambda x: 0.5* x * (x - 1), lambda x: -x ** 2 + 1, lambda x:
0.5 * x * (x + 1))
        pochodna =(lambda x: x - 0.5, lambda x: -2 * x, lambda x: x + 0.5)

    else:
        raise Exception("Nieoczekiwany stop_fun_baz w BaseFunctions().")

    return funkcja, pochodna
```

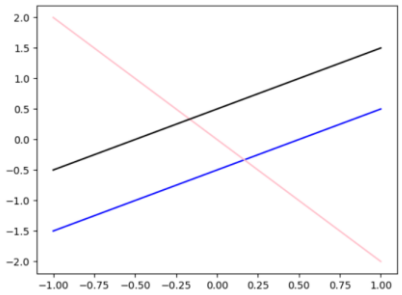
Wywołanie

```
stopien_fun_bazowych = 2
phi, dphi = BaseFunctions(stopien_fun_bazowych)

x = np.linspace(-1,1, 101)
plt.plot(x, phi[0](x), 'red')
plt.plot(x, phi[1](x), 'green')
plt.plot(x, phi[2](x), 'brown')
plt.plot(x, dphi[0](x), 'blue')
plt.plot(x, dphi[1](x), 'pink')
plt.plot(x, dphi[2](x), 'black')
plt.show()
```

Funkcje bazowe



Pochodne funkcji bazowych



Funkcje bazowe i ich pochodne wykreślone razem