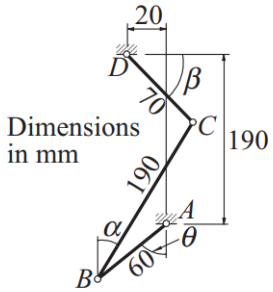
Metody Obliczeniowe Fizyki

Projekt 1 – poprawa

Wykonała: Aleksandra Kwiatkowska
Nr indeksu: 271901

Data oddania: 26.01.2024 r.

Zadanie 14 str. 197 „Numerical Methods in Engineering with Python 3”:



Geometric analysis of the linkage shown resulted in the following table relating the angles θ and β:

θ (deg)	0	30	60	90	120	150
β (deg)	59.96	56.42	44.10	25.72	−0.27	−34.29

Assuming that member AB of the linkage rotates with the constant angular velocity $d\theta/dt = 1$ rad/s, compute $d\beta/dt$ in rad/s at the tabulated values of θ. Use cubic spline interpolation.

Cel:

Wyznaczyć $d\beta/dt$ w rad/s przy podanych wartościach kąta θ za pomocą interpolacji metodą sześciennej funkcji sklejanej.

Wykorzystane metody:

Funkcje sklejane sześcienne (kubiczne) – Cubic Spline Interpolation:

Własności:
1) Gładkość – funkcje sklejane sześcienne są ciągłe do drugiego rzędu, co oznacza, że są dwukrotnie różniczkowalne na każdym przedziale interpolacji.
2) Elipsoidalność (shape-preserving) – zachowują kształt oryginalnej funkcji
3) Interpolacyjność – przechodzą dokładnie przez zadane punkty, przez co dobrze nadają się do interpolacji szeregów danych
4) Stabilność numeryczna – sześcienne funkcje sklejane są stosunkowo stabilne numerycznie i nie są podatne na efekt Rungego (pogorszenie jakości interpolacji wielomianowej, pomimo zwiększenia liczby węzłów)

Macierze trójprzekątniowe - macierze, których jedyne niezerowe elementy znajdują się na trzech przekątnych: głównej, oraz dwóch sąsiednich. Pozostałe elementy są zerowe. Macierze te są wykorzystywane w obliczeniach numerycznych, ponieważ zajmują mniej pamięci, a operacje na nich są bardziej efektywne w porównaniu do macierzy pełnych. Szczególne zastosowanie macierze te znajdują w różniczkowaniu i całkowaniu numerycznym oraz w algorytmach iteracyjnych, w tym w metodzie Gaussa-Seidla.

Metoda Gaussa-Seidla – iteracyjna metoda rozwiązywania układów równań liniowych. Technika numeryczna, która pozwala zbliżyć się do dokładnego rozwiązania poprzez wielokrotne poprawianie przybliżenia.

Przykładowy pseudokod metody Gaussa-Seidla:

```
for k = 1 to max_iterations do
    for I = 1 to n do

        sum1 = sum(a[i][j] * x[j] for j = 1 to i-1)
        sum2 = sum(a[i][j] * x[j] for j = i+1 to n)
        x[i] = (b[i] – sum1 – sum2) / a[i][i]

    end for
end for
```

Wykonanie:

1/ Zaimportowano bibliotekę NumPy. Następnie do programu wprowadzono wartości kątów podane w poleceniu. Przy użyciu funkcji np.radians() zamieniono stopnie na radiany w celu ułatwienia dalszych działań.

```
import numpy as np

# Zadane wartości
theta_degrees = np.array([0, 30, 60, 90, 120, 150])
beta_degrees = np.array([59.96, 56.42, 44.10, 25.72, -0.27, -34.29])

# Zamiana stopni na radiany
theta_radians = np.radians(theta_degrees)
beta_radians = np.radians(beta_degrees)
```

2/ Zaprogramowano zmienne/współczynniki, które będą wykorzystywane w dalszej części programu.

```
# Zdefiniowanie współczynników sześciennej funkcji sklejanej
n = len(theta_radians) - 1 # liczba odcinków na których interpolujemy
h = np.diff(theta_radians) # różnice w wartościach theta

alpha = np.zeros(n) # współczynniki wykorzystywane do obliczania
#drugichpochodnych, zależne od h i wartości funckji w punktach
beta = np.zeros(n) #współczynniki określające składnik liniowy w funkcji sklejanej
delta_beta = np.diff(beta_radians) # różnice pomiędzy uzyskanymi wartościami beta
d = np.zeros(n) # zapewnia płynność interpolacji

c = np.zeros(n + 1) # drugie pochodne funckji sklejanych w zadanych punktach
l = np.ones(n + 1) # elementy diagonalne macierzy używanej do obliczenia drugich
#pochodnych funkcji sklejanej
mu = np.zeros(n) # czynniki wyorzystane do obliczania drugich pochodnych = h[i]/l[i]
z=1 # zmienna pomocnicza dla układu równań
```

3/ Drugie pochodne funkcji sklejanej wyznaczono metodą Gaussa-Seidla

```
# Wyznaczenie drugich pochodnych funkcji sklejanej metodą Gaussa-Seidla
for i in range(1, n):
    alpha[i] = (3 / h[i]) * (delta_beta[i] / h[i] - delta_beta[i - 1] / h[i - 1])
    l[i] = 2 * (theta_radians[i + 1] - theta_radians[i - 1]) - h[i - 1] * mu[i - 1]
    mu[i] = h[i] / l[i]
    z = (alpha[i] - h[i - 1] * z) / l[i]

for j in range(n - 1, -1, -1):
    c[j] = z = z - mu[j] * c[j + 1]

for i in range(n):
    d[i] = (c[i + 1] - c[i]) / (3 * h[i])
    beta[i] = (beta_radians[i + 1] - beta_radians[i]) / h[i] - h[i] * (c[i + 1] + 2 * c[i]) / 3
    alpha[i] = beta_radians[i]
```

4/ Zdefiniowano funkcję interpolującą

```python
# Zdefiniowanie funkcji interpolującej
def beta_function(theta):
    for i in range(n):
        if theta_radians[i] <= theta <= theta_radians[i + 1]:
            delta_theta = theta - theta_radians[i]
            result = alpha[i] + beta[i] * delta_theta + c[i] * delta_theta**2 + d[i] * delta_theta**3
            return np.degrees(result)
```

5/ Sprawdzono działanie funkcji

```python
# Przetestowanie funkcji
test_theta = np.radians(np.linspace(0, 150, 36))
result_beta = np.array([beta_function(theta) for theta in test_theta])


for theta, beta in zip(np.degrees(test_theta), result_beta):
    print(f'Theta: {theta:.2f} degrees, Beta: {beta:.2f} degrees')
```

Funkcja zwraca:

```
Theta: 0.00 degrees, Beta: 59.96 degrees
Theta: 4.29 degrees, Beta: 59.93 degrees
Theta: 8.57 degrees, Beta: 59.73 degrees
Theta: 12.86 degrees, Beta: 59.38 degrees
Theta: 17.14 degrees, Beta: 58.88 degrees
Theta: 21.43 degrees, Beta: 58.22 degrees
Theta: 25.71 degrees, Beta: 57.40 degrees
Theta: 30.00 degrees, Beta: 56.42 degrees
Theta: 34.29 degrees, Beta: 55.19 degrees
Theta: 38.57 degrees, Beta: 53.80 degrees
Theta: 42.86 degrees, Beta: 52.23 degrees
Theta: 47.14 degrees, Beta: 50.49 degrees
Theta: 51.43 degrees, Beta: 48.55 degrees
Theta: 55.71 degrees, Beta: 46.43 degrees
Theta: 60.00 degrees, Beta: 44.10 degrees
Theta: 64.29 degrees, Beta: 42.19 degrees
Theta: 68.57 degrees, Beta: 40.06 degrees
Theta: 72.86 degrees, Beta: 37.70 degrees
Theta: 77.14 degrees, Beta: 35.09 degrees
Theta: 81.43 degrees, Beta: 32.23 degrees
Theta: 85.71 degrees, Beta: 29.11 degrees
Theta: 90.00 degrees, Beta: 25.72 degrees
Theta: 94.29 degrees, Beta: 22.98 degrees
Theta: 98.57 degrees, Beta: 19.94 degrees
Theta: 102.86 degrees, Beta: 16.59 degrees
Theta: 107.14 degrees, Beta: 12.91 degrees
Theta: 111.43 degrees, Beta: 8.88 degrees
Theta: 115.71 degrees, Beta: 4.49 degrees
Theta: 120.00 degrees, Beta: -0.27 degrees
Theta: 124.29 degrees, Beta: -4.41 degrees
Theta: 128.57 degrees, Beta: -8.88 degrees
Theta: 132.86 degrees, Beta: -13.63 degrees
Theta: 137.14 degrees, Beta: -18.60 degrees
Theta: 141.43 degrees, Beta: -23.73 degrees
Theta: 145.71 degrees, Beta: -28.98 degrees
Theta: 150.00 degrees, Beta: -34.29 degrees
```

Funkcja zwraca prawidłowe wartości dla punktów zadanych w poleceniu oraz wydaje się prawidłowo interpolować wartości kąta beta dla punktów pośrednich.

6/ Pochodną $d\beta/d\theta$ wyznaczono przy użyciu funkcji np.gradient().
Następnie korzystając z reguły łańcuchowej wyliczono szukane wartości $d\beta/dt = d\beta/d\theta * d\theta/dt$.

```
# Do obliczenia pochodnej dbeta/dtheta wykorzystujemy gotową funckję na gradient
derivative_radians = np.gradient(beta_radians, theta_radians)

# Z treści zadania wiemy, że dtheta/dt = 1 rad/s
dtheta_dt = 1

# Obliczamy dbeta.dt korzystając z reguły łańcuchowej dbeta/dt = dbeta/dtheta * dtheta/dt
derivative_dt = dtheta_dt * derivative_radians

# Wyświetlamy wyniki
given_theta = [0, 30, 60, 90, 120, 150]
for theta, dbeta_dt in zip(given_theta, derivative_dt):
    print(f'Theta: {theta:.2f} degrees, dBeta/dt: {dbeta_dt:.4f} radians/s')
```

```
Theta: 0.00 degrees, dBeta/dt: -0.1180 radians/s
Theta: 30.00 degrees, dBeta/dt: -0.2643 radians/s
Theta: 60.00 degrees, dBeta/dt: -0.5117 radians/s
Theta: 90.00 degrees, dBeta/dt: -0.7395 radians/s
Theta: 120.00 degrees, dBeta/dt: -1.0002 radians/s
Theta: 150.00 degrees, dBeta/dt: -1.1340 radians/s
```