

Metody Numeryczne

Laboratorium 7: Pierwiastki wielomianu i wartości własne

Wykonał: <imię i nazwisko>

Metoda zaliczenia:

Podczas zajęć należy wykonać poniższe polecenia oraz udzielić odpowiedzi na pytania zamieszczone w treści zadań.

Wszystkie funkcje wymagane w ramach ćwiczenia należy zaimplementować w pliku `main.py`. Poprawność ich działania należy zweryfikować za pomocą testów jednostkowych dostępnych w pliku `test_main.py`.

Cel zajęć:

Celem zajęć jest zapoznanie się z numerycznymi metodami rozwiązywania równań nieliniowych. W związku z tym podczas zajęć będziemy rozważać następujący problem:

Dana jest funkcja $f(x)$. Należy wyznaczyć takie jej argumenty x^* , dla których zachodzi równość $f(x^*) = 0$.

Argumenty spełniające powyższe równanie nazywane są *pierwiastkami równania*.

Wielomian

Dany jest wielomian $w(x)$ w postaci kanonicznej (ogólnej):

$$w(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

jego wektor współczynników wyraża się wzorem:

$$\mathbf{a} = (a_0, a_1, \dots, a_{n-1}, a_n)$$

Uwagi wstępne:

- Funkcje wymagające implementacji (lub zaimplementowane na poprzednich laboratoriach) oznaczone są pochyłą czcionką maszynową (np. `my_func()`).
- W skrypcie stosuje się następujące skróty:
 - `np` - `numpy`,

- `sp` - `scipy`,
- `nppoly` - `numpy.polynomial.polynomial`.

In [3]: `# !python -m pip install numpy scipy matplotlib`

```
import main

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import numpy.polynomial.polynomial as nppoly
```

Zadanie 1.

Dane są wielomiany w postaci iloczynowej:

$$w_1(x) = (x - 1)(x - 2)(x - 3)$$
$$w_2(x) = (x - 1)(x - 2) \cdot \dots \cdot (x - 20)$$

Punkt 1.

1. Oblicz wektor współczynników \mathbf{a}_1 wielomianu $w_1(x)$ w postaci kanonicznej dla zadanego wektora pierwiastków.
2. Sprawdź poprawność wyznaczonego wektora z definicją w sekcji **Cel zajęć**.

Wskazówka: Użyj funkcji `nppoly.polyfromroots()`.

Punkt 2.

Zaimplementuj funkcję `roots_20()` wyznaczającą miejsca zerowe wielomianu na podstawie jego wektora współczynników. Funkcja powinna najpierw lekko zaburzać wejściowe współczynniki za pomocą dodania do nich losowej wartości postaci: $N(0, 1) \cdot 1e-10$.

Wskazówka: Użyj funkcji `np.random.random_sample()` oraz `nppoly.polyroots()`.

Punkt 3.

1. Wyznacz wektor współczynników \mathbf{a}_2 wielomianu $w_2(x)$.
2. W pętli 20 iteracji:
 - A. Wyznacz pierwiastki zaburzonego wielomianu używając `roots_20()`.
 - B. Ustaw zaburzony wektor współczynników jako nowy wektor współczynników.

- C. Wyrysuj na jednym wykresie wyznaczone części rzeczywiste pierwiastków, a na drugim części urojone (w każdej iteracji dorysowuj pierwiastki na tym samym rysunku).
3. Określ, które pierwiastki są najbardziej wrażliwe na zaburzenia oraz opisz wnioski wynikające z utworzonego wykresu.
4. Zaproponuj sposób oszacowania uwarunkowania każdego z pierwiastków.

```
In [13]: # ===== Twoja implementacja tutaj =====
w1_roots = [1,2,3]
w2_roots = [x for x in range(1,21)]
a1 = nppoly.polyfromroots(w1_roots)
a2 = nppoly.polyfromroots(w2_roots)
print(a1)
print(a2)

def roots_20(coef: np.ndarray) -> tuple[np.ndarray, np.ndarray] | None:
    if not isinstance(coef, np.ndarray): return None
    if coef.ndim != 1: return None
    a = coef + np.random.random_sample() * (10**(-10))
    return (a, nppoly.polyroots(a))
    """Funkcja wyznaczająca miejsca zerowe wielomianu funkcją
    `nppoly.polyroots()`, najpierw lekko zaburzając wejściowe współczynniki
    wielomianu (N(0,1) * 1e-10).

    Args:
        coef (np.ndarray): Wektor współczynników wielomianu (n,).

    Returns:
        (tuple[np.ndarray, np.ndarray]):
            - Zaburzony wektor współczynników (n,),
            - Wektor miejsc zerowych (m,).
        Jeżeli dane wejściowe są niepoprawne funkcja zwraca `None`.
    """

# jak przyjmujesz tę kolejność?
a2 = nppoly.polyfromroots(w2_roots)
roots = []

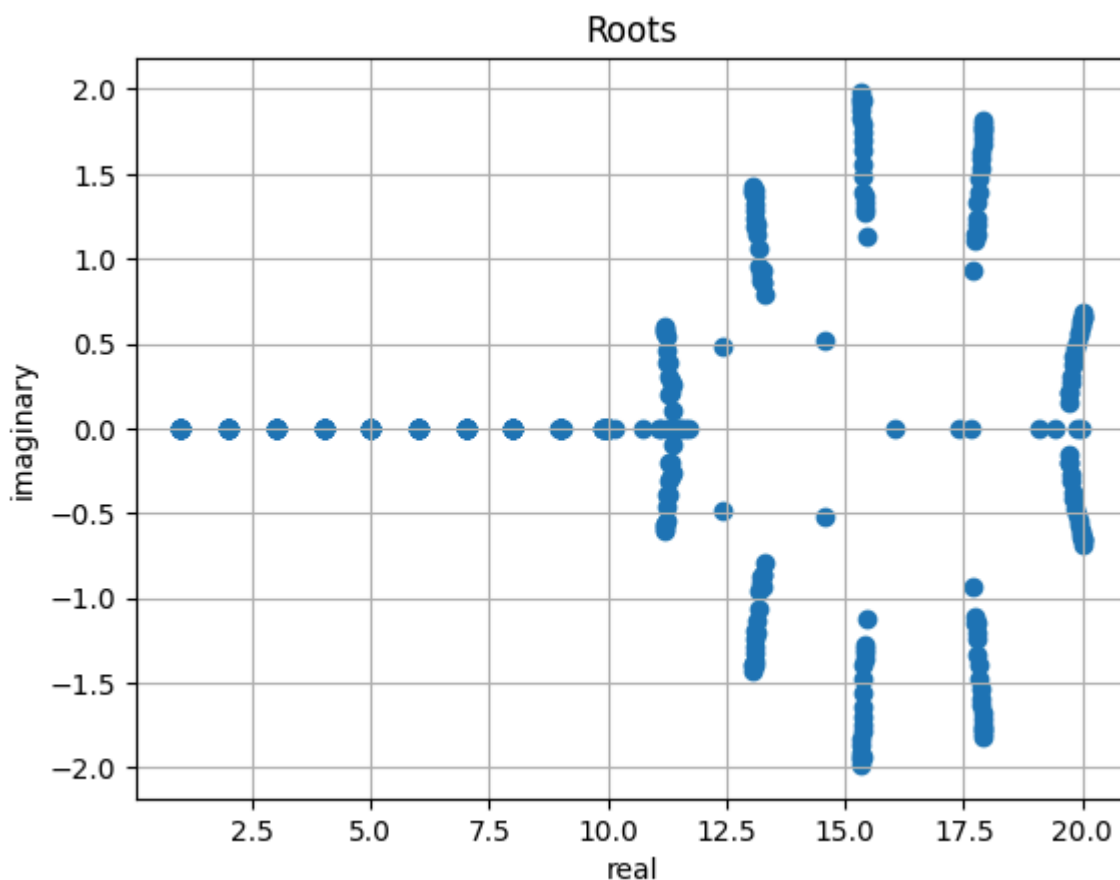
for n in range(20):
    (a2_disturb, rts) = roots_20(a2)
    a2 = a2_disturb
    roots.append(rts)

roots = np.array(roots)
fig, ax = plt.subplots()
ax.scatter(roots.real, roots.imag)
ax.set_title("Roots")
ax.set_xlabel("real")
ax.set_ylabel("imaginary")
ax.grid()
```

```

[-6. 11. -6. 1.]
[ 2.43290201e+18 -8.75294804e+18 1.38037598e+19 -1.28709312e+19
 8.03781182e+18 -3.59997952e+18 1.20664780e+18 -3.11333643e+17
 6.30308121e+16 -1.01422999e+16 1.30753501e+15 -1.35585183e+14
 1.13102770e+13 -7.56111184e+11 4.01717716e+10 -1.67228082e+09
 5.33279460e+07 -1.25685000e+06 2.06150000e+04 -2.10000000e+02
 1.00000000e+00]

```



Na podstawie wykresu widać, że pierwiastki z prawej części (około 11–20) są najbardziej niestabilne i najmocniej się rozpraszają, szczególnie te bliżej 15–18. Pierwiastki początkowe zachowują się stabilnie. Uwarunkowanie pierwiastków można oszacować jako odwrotność wartości pochodnej wielomianu w danym pierwiastku.

Zadanie 2.

Na podstawie wiedzy z wykładu wiadomo, że wartości własne macierzy kwadratowej są pierwiastkami wielomianu charakterystycznego tej macierzy oraz, że każdy wielomian posiada swoją macierz stowarzyszoną (macierz Frobeniusa). Wynika z tego, że **poszukiwanie pierwiastków wielomianu jest równoważne poszukiwaniu wartości własnych macierzy stowarzyszonej.**

Dany jest wielomian:

$$w_3(x) = (x - 1)^8$$

Należy wyznaczyć numerycznie jego miejsca zerowe na różne sposoby i porównać otrzymane wyniki z wartościami analitycznymi.

Punkt 1.

Przygotowanie macierzy stowarzyszonej.

1. Zaimplementuj funkcję `frob_a()` tworzącą **macierz Frobeniusa** dla zadanego wektora współczynników wielomianu.
2. Wyznacz macierz Frobeniusa \mathbf{F}_{w_3} dla wielomianu $w_3(x)$.

Punkt 2.

Wyznaczenie pierwiastków wielomianu.

1. Wyznacz wartości własne macierzy \mathbf{F}_{w_3} za pomocą funkcji `np.linalg.eigvals()`.
2. Dokonaj rozkładu Schura macierzy \mathbf{F}_{w_3} i na tej podstawie wyznacz wartości własne macierzy \mathbf{F}_{w_3} . Użyj funkcji `sp.linalg.schur()`.
3. Wyznacz pierwiastki wielomianu $w_3(x)$ przy użyciu funkcji `nppoly.polyroots()`.

Punkt 3.

Zestawienie wyników.

1. Utwórz wykres typu *scatter plot* i wyrysuj na nim wartości pierwiastków wyznaczonych w **Punkcie 2.** oraz pierwiastki obliczone analitycznie na płaszczyźnie zespolonej.
2. Opisz wnioski wynikające z wykresu.

```
In [29]: # ===== Twoja implementacja tutaj =====
def frob_a(coef: np.ndarray) -> np.ndarray | None:

    if not isinstance(coef, np.ndarray):
        return None
    if coef.ndim != 1:
        return None
    if coef.size < 2:
        return None
    if coef[0] == 0:
        return None
    if coef.size == 2:
        return np.array([[-coef[1] / coef[0]]], dtype=float)

    I = np.eye(len(coef)-2)

    z = [0] * (len(coef)-2)
```

```

zI = np.column_stack((z,I))

a = []
for n in range(1,len(coef)):
    a.append(-1*coef[-n] / coef[0])

f = np.row_stack((zI,a))
return f

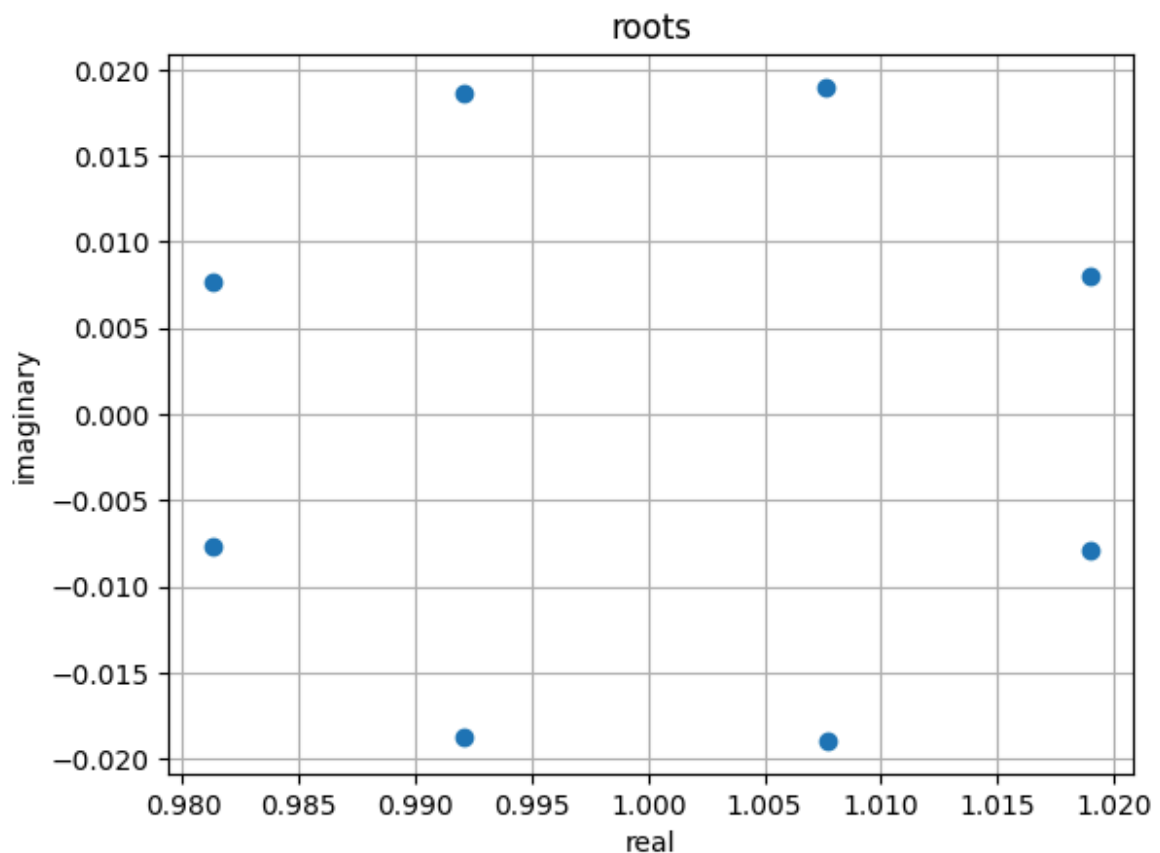
w3_roots = np.array([1]*8)
w3 = nppoly.polyfromroots(w3_roots)
frob = frob_a(w3)

eigen_values = np.linalg.eigvals(frob)
T, Z = sp.linalg.schur(frob, output='complex')
eigen_schur = np.diag(T)

plt.scatter(eigen_schur.real,eigen_schur.imag)
plt.title("roots")
plt.xlabel("real")
plt.ylabel("imaginary")
plt.grid()

```

/tmp/ipykernel_4870/1551276517.py:25: DeprecationWarning: `row_stack` alias is deprecated. Use `np.vstack` directly.
 f = np.row_stack((zI,a))



Wielomian ma pierwiastek wielokrotny w punkcie $x = 1$. Analitycznie wszystkie pierwiastki są dokładnie równe 1, ale numeryczne wyniki rozpraszają się wokół tej wartości. Świadczy to o bardzo dużej wrażliwości pierwiastków wielokrotnych na błędy

numeryczne i minimalne zaburzenia. Widać wyraźnie, że nawet niewielkie zmiany współczynników powodują rozsypanie się pierwiastków w pobliżu jedynki.

Zadanie 3.

Dany jest wielomian:

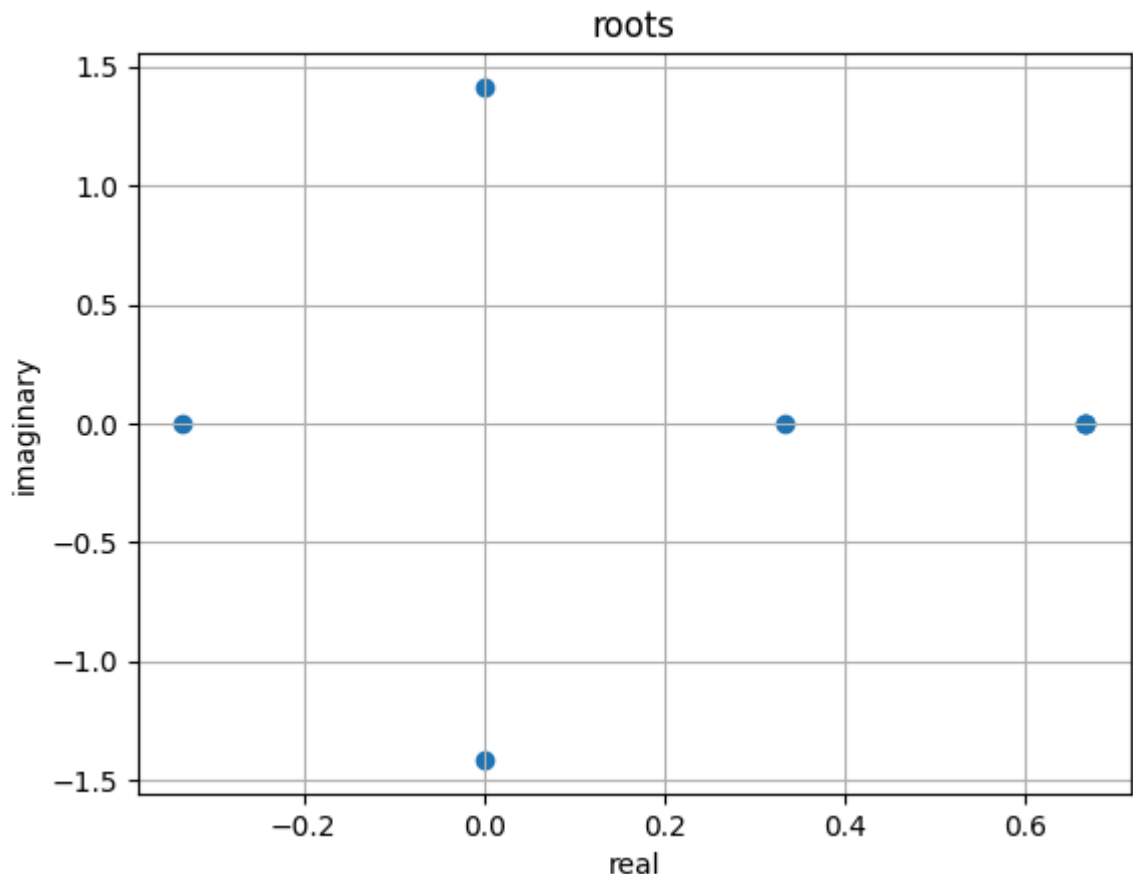
$$w_4(x) = 243x^7 - 486x^6 + 783x^5 - 990x^4 + 558x^3 - 28x^2 - 72x + 16$$

1. Wyznacz numerycznie miejsca zerowe wielomianu $w_4(x)$ w sposób analogiczny jak w **Zadaniu 2**.
2. Wyrysuj na płaszczyźnie zespolonej wyznaczone pierwiastki oraz pierwiastki wyliczone analitycznie.
3. Porównaj otrzymane wyniki z rezultatami otrzymanymi w **Zadaniu 2**.

Wskazówka: Sprawdź poprawność wyznaczonego wektora współczynników z definicją w sekcji **Cel zajęć**.

```
In [28]: # ===== Twoja implementacja tutaj =====  
w4 = np.array([243, -486, 783, -990, 558, -28, -72, 16])  
frob = frob_a(w4)  
  
eigen_values = np.linalg.eigvals(frob)  
T, Z = sp.linalg.schur(frob, output='complex')  
eigen_schur = np.diag(T)  
  
plt.scatter(eigen_schur.real, eigen_schur.imag)  
plt.title("roots")  
plt.xlabel("real")  
plt.ylabel("imaginary")  
plt.grid()
```

```
/tmp/ipykernel_4870/1860395988.py:25: DeprecationWarning: `row_stack` alias  
is deprecated. Use `np.vstack` directly.  
f = np.row_stack((zI,a))
```



Na wykresie widać, że wielomian ma zarówno pierwiastki rzeczywiste jak i zespolone. Pierwiastki zespolone występują parami sprzężonymi zgodnie z teorią. Wyniki uzyskane metodami numerycznymi są stabilne i nie rozpraszają się jak w zadaniu 2. Oznacza to, że ten wielomian jest lepiej uwarunkowany numerycznie i pierwiastki są jednoznacznie wyznaczalne.

Zadanie 4.

Zadanie polega na przeanalizowaniu w jaki sposób różne przekształcenia macierzy wpływają na pojawianie się błędów numerycznych.

W celu przeprowadzenia analizy skonstruuj trzy macierze diagonalne \mathbf{A}_n dla $n = \{10, 20, 30\}$. Współczynniki macierzy powinny być zdefiniowane jako $a_{ii} = 2^i$ dla $i = \{1, 2, \dots, n\}$ (skorzystaj z funkcji `np.diag()`).

Punkt 1.

1. Dla każdej z macierzy \mathbf{A}_n oblicz wartości własne przy użyciu `np.linalg.eigvals()` i porównaj je z wartościami własnymi wyznaczonymi analitycznie.

Punkt 2.

1. Zdefiniuj funkcję `is_nonsingular()` sprawdzającą czy zadana macierz nie jest singularna.
2. Skonstruuj losowe niesingularne macierze wektorów własnych \mathbf{P}_n , których wartości współczynników są liczbami całkowitymi z zakresu $(0, 100)$. Na podstawie macierzy \mathbf{A}_n i \mathbf{P}_n oblicz macierze \mathbf{B}_n postaci:

$$\mathbf{B}_n = \mathbf{P}_n \mathbf{A}_n \mathbf{P}_n^{-1}$$

3. Oblicz wartości własne macierzy dla uzyskanych macierzy \mathbf{B}_n i porównaj je z wartościami własnymi wyznaczonymi analitycznie.

Wskazówka: Niesingularność macierzy \mathbf{P}_n można zapewnić generując je w pętli, w której warunkiem stopu będzie wynik funkcji zaimplementowanej w **Podpunkcie 1**.

Punkt 3.

1. Bazując na macierzach \mathbf{P}_n wygeneruj macierze ortogonalne \mathbf{Q}_n (można do tego użyć rozkładu QR). Na podstawie macierzy \mathbf{A}_n i \mathbf{Q}_n oblicz macierze \mathbf{C}_n postaci:

$$\mathbf{C}_n = \mathbf{Q}_n \mathbf{A}_n \mathbf{Q}_n^{-1}$$

2. Oblicz wartości własne macierzy dla uzyskanych macierzy \mathbf{C}_n i porównaj je z wartościami własnymi wyznaczonymi analitycznie.

Punkt 4.

1. Bazując na analitycznych wartościach własnych dla wielomianów charakterystycznych macierzy \mathbf{A}_n wygeneruj macierze Frobeniusa \mathbf{F}_n korzystając z funkcji zaimplementowanej w **Zadaniu 2**.
2. Oblicz wartości własne macierzy dla uzyskanych macierzy \mathbf{F}_n i porównaj je z wartościami własnymi wyznaczonymi analitycznie.

Punkt 5.

Porównaj wyniki otrzymane we wszystkich punktach. Opisz z czego wynikają powstałe błędy obliczeń.

```
In [ ]: # ===== Twoja implementacja tutaj =====
def A_n(n):
    diag_elements = 2 ** np.arange(1, n+1)
    return np.diag(diag_elements)

A10 = A_n(10)
A20 = A_n(20)
A30 = A_n(30)
A10_eigen = np.linalg.eigvals(A10)
A20_eigen = np.linalg.eigvals(A20)
```

```

A30_eigen = np.linalg.eigvals(A30)
print(A10_eigen)
print(A20_eigen)
print(A30_eigen)

def is_nonsingular(A: np.ndarray) -> bool | None:
    if not isinstance(A, np.ndarray) or A.ndim != 2 or A.shape[0] != A.shape[1]:
        return None

    det = np.linalg.det(A)

    eps = 1e-12

    return abs(det) > eps

ns = [10, 20, 30]

def random_nonsingular_P(n: int) -> np.ndarray:
    """Losowa niesingularna macierz całkowita z (0,100)."""
    while True:
        P = np.random.randint(0, 100, size=(n, n)) # całkowite z [0, 100)
        ok = is_nonsingular(P)
        if ok:
            return P

Pn = {}
Bn = {}
Bn_eigen = {}

for n in ns:
    A = A_n(n)
    P = random_nonsingular_P(n)
    Pinv = np.linalg.inv(P)

    B = P @ A @ Pinv

    Pn[n] = P
    Bn[n] = B

    eig_B = np.linalg.eigvals(B)
    Bn_eigen[n] = eig_B

    eig_analytic = np.diag(A)

    eig_B_sorted = np.sort(eig_B.real)
    eig_analytic_sorted = np.sort(eig_analytic)

    diff = np.max(np.abs(eig_B_sorted - eig_analytic_sorted))
    print(f"[Punkt 2] n = {n}, max różnica eigen(B_n) vs analityczne =",

Qn = {}
Cn = {}
Cn_eigen = {}

for n in ns:
    A = A_n(n)
    P = Pn[n]

    Q, R = np.linalg.qr(P.astype(float))

```

```

Qn[n] = Q

C = Q @ A @ Q.T
Cn[n] = C

eig_C = np.linalg.eigvals(C)
Cn_eigen[n] = eig_C

eig_analytic = np.diag(A)

eig_C_sorted = np.sort(eig_C.real)
eig_analytic_sorted = np.sort(eig_analytic)

diff = np.max(np.abs(eig_C_sorted - eig_analytic_sorted))
print(f"[Punkt 3] n = {n}, max różnica eigen(C_n) vs analityczne =",

Fn = {}
Fn_eigen = {}

for n in ns:

    lambdas = 2 ** np.arange(1, n+1)

    coef = nppoly.polyfromroots(lambdas)

    coef_frob = coef[::-1]

    F = frob_a(coef_frob)
    Fn[n] = F

    eig_F = np.linalg.eigvals(F)
    Fn_eigen[n] = eig_F

    eig_F_sorted = np.sort(eig_F.real)
    eig_analytic_sorted = np.sort(lambdas)

    diff = np.max(np.abs(eig_F_sorted - eig_analytic_sorted))
    print(f"[Punkt 4] n = {n}, max różnica eigen(F_n) vs analityczne =",

    n_values = [10, 20, 30]

max_error_A = np.zeros(len(n_values))
mean_error_A = np.zeros(len(n_values))

max_error_B = np.zeros(len(n_values))
mean_error_B = np.zeros(len(n_values))

max_error_C = np.zeros(len(n_values))
mean_error_C = np.zeros(len(n_values))

max_error_F = np.zeros(len(n_values))
mean_error_F = np.zeros(len(n_values))

def eig_errors(eig_num, eig_anal):
    eig_num_sorted = np.sort(eig_num.real)
    eig_anal_sorted = np.sort(eig_anal.real if np.iscomplexobj(eig_anal)
    diff = np.abs(eig_num_sorted - eig_anal_sorted)
    return diff.max(), diff.mean()

for i, n in enumerate(n_values):

```

```

A = A_n(n)
eig_anal = np.diag(A)

# A_n (diag)
eig_A = np.linalg.eigvals(A)
max_error_A[i], mean_error_A[i] = eig_errors(eig_A, eig_anal)

# B_n = P A P^-1
eig_B = Bn_eigen[n]
max_error_B[i], mean_error_B[i] = eig_errors(eig_B, eig_anal)

# C_n = Q A Q^T
eig_C = Cn_eigen[n]
max_error_C[i], mean_error_C[i] = eig_errors(eig_C, eig_anal)

# F_n (Frobenius)
eig_F = Fn_eigen[n]
max_error_F[i], mean_error_F[i] = eig_errors(eig_F, eig_anal)

for i, n in enumerate(n_values):
    print(f"n = {n}")
    print(f"    {'Macierz':<18} {'Błąd maksymalny':<18} {'Błąd średni':<18}
    print(f"    {'-'*55}")
    print(f"    {'A_' + str(n) + ' (diag)':<18} {max_error_A[i]:<18.2e} {m
    print(f"    {'B_' + str(n) + ' (P*A*P^-1)':<18} {max_error_B[i]:<18.2e
    print(f"    {'C_' + str(n) + ' (Q*A*Q^T)':<18} {max_error_C[i]:<18.2e}
    print(f"    {'F_' + str(n) + ' (Frobenius)':<18} {max_error_F[i]:<18.2

```

```
[ 2.    4.    8.   16.   32.   64.  128.  256.  512. 1024.]
[2.000000e+00 4.000000e+00 8.000000e+00 1.600000e+01 3.200000e+01
 6.400000e+01 1.280000e+02 2.560000e+02 5.120000e+02 1.024000e+03
 2.048000e+03 4.096000e+03 8.192000e+03 1.638400e+04 3.276800e+04
 6.553600e+04 1.310720e+05 2.621440e+05 5.242880e+05 1.048576e+06]
[2.000000000e+00 4.000000000e+00 8.000000000e+00 1.600000000e+01
 3.200000000e+01 6.400000000e+01 1.280000000e+02 2.560000000e+02
 5.120000000e+02 1.024000000e+03 2.048000000e+03 4.096000000e+03
 8.192000000e+03 1.638400000e+04 3.276800000e+04 6.553600000e+04
 1.310720000e+05 2.621440000e+05 5.242880000e+05 1.048576000e+06
 2.097152000e+06 4.194304000e+06 8.388608000e+06 1.677721600e+07
 3.355443200e+07 6.710886400e+07 1.34217728e+08 2.68435456e+08
 5.36870912e+08 1.07374182e+09]
[Punkt 2] n = 10, max różnica eigen(B_n) vs analityczne = 3.03401748169562
78e-12
[Punkt 2] n = 20, max różnica eigen(B_n) vs analityczne = 1.02227016185452
16e-06
[Punkt 2] n = 30, max różnica eigen(B_n) vs analityczne = 2.11736187338829
04e-06
[Punkt 3] n = 10, max różnica eigen(C_n) vs analityczne = 1.70530256582424
04e-12
[Punkt 3] n = 20, max różnica eigen(C_n) vs analityczne = 1.39698386192321
78e-09
[Punkt 3] n = 30, max różnica eigen(C_n) vs analityczne = 1.9073486328125e
-06
[Punkt 4] n = 10, max różnica eigen(F_n) vs analityczne = 1.59161572810262
44e-12
[Punkt 4] n = 20, max różnica eigen(F_n) vs analityczne = 2.21189111471176
15e-09
[Punkt 4] n = 30, max różnica eigen(F_n) vs analityczne = 1056964608.0
n = 10
```

Macierz	Błąd maksymalny	Błąd średni

A_10 (diag)	0.00e+00	0.00e+00
B_10 (P*A*P^-1)	3.03e-12	1.12e-12
C_10 (Q*A*Q^T)	1.71e-12	3.29e-13
F_10 (Frobenius)	1.59e-12	4.21e-13

n = 20

Macierz	Błąd maksymalny	Błąd średni

A_20 (diag)	0.00e+00	0.00e+00
B_20 (P*A*P^-1)	1.02e-06	3.87e-07
C_20 (Q*A*Q^T)	1.40e-09	1.63e-10
F_20 (Frobenius)	2.21e-09	4.83e-10

n = 30

Macierz	Błąd maksymalny	Błąd średni

A_30 (diag)	0.00e+00	0.00e+00
B_30 (P*A*P^-1)	2.12e-06	5.50e-07
C_30 (Q*A*Q^T)	1.91e-06	1.36e-07
F_30 (Frobenius)	1.06e+09	7.05e+07

```
/tmp/ipykernel_4870/1551276517.py:25: DeprecationWarning: `row_stack` alia
s is deprecated. Use `np.vstack` directly.
f = np.row_stack((zI,a))
```

Macierze B_n i C_n zachowują bardzo małe błędy zarówno średnie jak i maksymalne. Oznacza to, że podobieństwo macierzowe PAP^{-1} oraz przekształcenie ortogonalne QAQ^T zachowują wartości własne w sposób numerycznie stabilny. Wartości własne macierzy B_n i C_n praktycznie pokrywają się z wartościami analitycznymi.

Dla macierzy Frobeniusa F_n błędy rosną bardzo szybko wraz z wymiarem n . Dla $n = 30$ błąd maksymalny osiąga bardzo duże wartości rzędu 10^9 , co pokazuje że metoda oparta na macierzy Frobeniusa jest numerycznie niestabilna dla większych wymiarów. Jest to związane ze złym uwarunkowaniem macierzy Frobeniusa i kumulacją błędów obliczeniowych.

Podsumowując, najbardziej stabilne numerycznie okazały się metody oparte na podobieństwie macierzowym (macierze B_n i C_n), natomiast metoda z macierzą Frobeniusa jest wrażliwa na błędy numeryczne i daje bardzo duże odchylenia szczególnie dla większych macierzy.

Przykładowa forma zestawienia wyników:

```
for i, n in enumerate(n_values):
    print(f"    {'Macierz':<18} {'Błąd maksymalny':<18} {'Błąd średni':<18}")
    print(f"    {'-'*55}")
    print(f"    {'A_' + str(n) + ' (diag)':<18} {max_error_A[i]:<18.2e} {mean_error_A[i]:<18.2e}")
    print(f"    {'B_' + str(n) + ' (P*A*P^-1)':<18} {max_error_B[i]:<18.2e} {mean_error_B[i]:<18.2e}")
    print(f"    {'C_' + str(n) + ' (Q*A*Q^T)':<18} {max_error_C[i]:<18.2e} {mean_error_C[i]:<18.2e}")
    print(f"    {'F_' + str(n) + ' (Frobenius)':<18} {max_error_F[i]:<18.2e} {mean_error_F[i]:<18.2e}\n")
```

Rezultat:

Macierz	Błąd maksymalny	Błąd średni

A_10 (diag)	0.00e+00	0.00e+00
B_10 (P*A*P^-1)	0.00e+00	0.00e+00
C_10 (Q*A*Q^T)	0.00e+00	0.00e+00
F_10 (Frobenius)	0.00e+00	0.00e+00

Macierz	Błąd maksymalny	Błąd średni

A_20 (diag)	0.00e+00	0.00e+00
B_20 (P*A*P^-1)	0.00e+00	0.00e+00
C_20 (Q*A*Q^T)	0.00e+00	0.00e+00
F_20 (Frobenius)	0.00e+00	0.00e+00

Macierz	Błąd maksymalny	Błąd średni

A_30 (diag)	0.00e+00	0.00e+00
B_30 (P*A*P^-1)	0.00e+00	0.00e+00
C_30 (Q*A*Q^T)	0.00e+00	0.00e+00
F_30 (Frobenius)	0.00e+00	0.00e+00

Materiały uzupełniające:

- [Scipy Lecture Notes](#)
- [NumPy for Matlab users](#)
- [Python Tutorial - W3Schools](#)
- [NumPy](#)
- [Matplotlib](#)
- [Anaconda](#)
- [Learn Python for Data Science](#)
- [Learn Python](#)
- [Wujek Google i Ciocia Wikipedia](#)