Laboratorium 3
Metody Numeryczne

Instrukcja:
Na zajęciach należy wykonać poniższe zadania, dokonać testu na platformie github, a następnie sporządzić sprawozdanie zawierające odpowiedzi z komentarzami.

Biblioteki niezbędne do wykonania zadania:

(instalacja: "pip install numpy scipy matplotlib memory_profiler")

In [2]:
```
#import main

import numpy as np
import scipy
import matplotlib
import matplotlib.pyplot as plt
import math
from typing import Union, List, Tuple

%load_ext memory_profiler
```

Materiały przygotowujące:

- Standard IEEE 754 PL (https://pl.wikipedia.org/wiki/IEEE_754) EN (https://en.wikipedia.org/wiki/IEEE_754)
- Liczba zmiennoprzecinkowa PL (https://pl.wikipedia.org/wiki/Liczba_zmiennoprzecinkowa) EN (https://en.wikipedia.org/wiki/Floating-point_arithmetic)
- Arytmetyka zmiennoprzecinkowa Python (https://docs.python.org/3.7/tutorial/floatingpoint.html)

Profilowanie kodu:

- timeit (https://docs.python.org/2/library/timeit.html) - profilowanie czasu wykonywania kodu
- memit (https://pypi.org/project/memory-profiler/) - profilowanie pamięci zużywanej przez kod

Zarówno timeit jak i memit wspierają magic command w Jupyter notebook, co obrazuje poniższy przykład:

In [8]:
```
def func(size):
    a = np.random.random((size,size))
    b = np.random.random((size,size))
    c = a + b
    return c

for size in [100, 1000, 10000]:
    print('SIZE: ', size)
    print('Timing: ')
    saved_timing = %timeit -r 5 -n 10 -o func(size)
    saved_timing.average   # średni czas próby
    saved_timing.stdev     # odchylenie standardowe
    print('Memory usage: ')
    %memit func(size)
    print('\n')
```

```
SIZE:  100
Timing: 
298 µs ± 78.7 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)
Memory usage: 
peak memory: 123.07 MiB, increment: 0.26 MiB


SIZE:  1000
Timing: 
30.4 ms ± 780 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)
Memory usage: 
peak memory: 123.16 MiB, increment: 0.09 MiB


SIZE:  10000
Timing: 
3.5 s ± 364 ms per loop (mean ± std. dev. of 5 runs, 10 loops each)
Memory usage: 
peak memory: 2335.03 MiB, increment: 2211.95 MiB
```

Zadanie 1.

Zaimplementuj funkcje *p_diff*, która przyjmuje jako parametry wartości całkowite n i rzeczywiste c oraz zwraca różnicę (co do wartości bezwzględnej) dwóch wyrażeń P_1 oraz P_2:

a) $P_1 = b - b + c$
b) $P_2 = b + c - b$

gdzie $b = 2^n$

Analizując różnicę w otrzymanych wynikach zastosuj warotści:

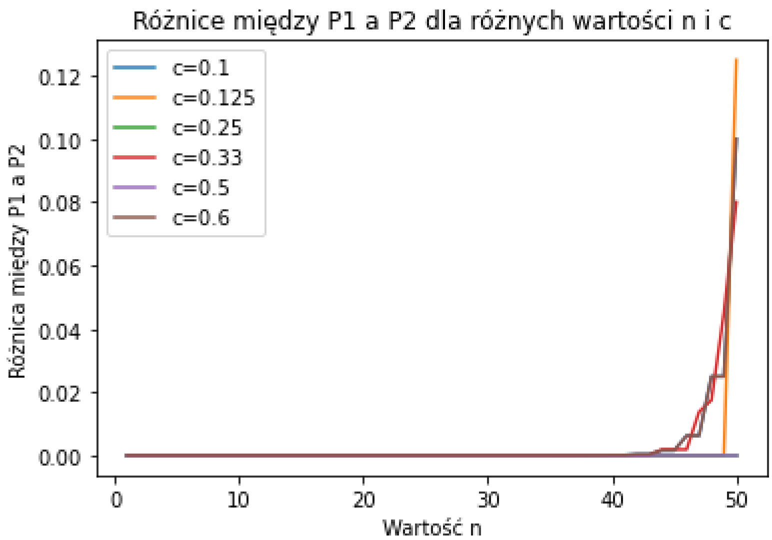
- $n \in \{1, 2, 3...50\}$
- $c \in \{0.1, 0.125, 0.25, 0.33, 0.5, 0.6\}$

Następnie odpowiedź i zilustruj wykresami pytania:

1. Jaki wynik powinniśmy otrzymać?
2. Które z liczb mają skończoną a które nieskończoną reprezentację?
3. Dlaczego wyniki się od siebie różnią?
4. Jaki typ błędu tutaj występuje?

5. Czy istnieje możliwość poprawy działania tych wyrażeń, jeżeli tak to w jaki sposób?

In [4]:
```
def p_diff(n, c):
    b = 2 ** n
    p1 = b - b + c
    p2 = b + c - b
    return abs(p1 - p2)

results = []
for n in range(1, 51):
    for c in [0.1, 0.125, 0.25, 0.33, 0.5, 0.6]:
        diff = p_diff(n, c)
        results.append((n, c, diff))


n_values = list(range(1, 51))
c_values = [0.1, 0.125, 0.25, 0.33, 0.5, 0.6]

for c in c_values:
    diffs = [diff for n, _, diff in results if _ == c]
    plt.plot(n_values, diffs, label=f'c={c}')

plt.xlabel('Wartość n')
plt.ylabel('Różnica między P1 a P2')
plt.legend()
plt.title('Różnice między P1 a P2 dla różnych wartości n i c')
plt.show()
```




```
Odpowiedzi na pytania
1. Jaki wynik powinniśmy otrzymać?
Oczekujemy, że wynik będzie zawsze równy 0, ponieważ obie wyrażenia są
identyczne

2. Które z liczb mają skończoną a które nieskończoną reprezentację?
Wartości n są całkowite, więc mają skończoną reprezentację. Wartości c są
dziesiętnymi ułamkami,
które w postaci dziesiętnej mają skończoną reprezentację dla podanych
wartości.

3. Dlaczego wyniki się od siebie różnią?
Wyniki różnią się ze względu na ograniczoną precyzję arytmetyki
zmiennoprzecinkowej w komputerze.
```

```
Niektóre wartości zmiennoprzecinkowe nie mogą być dokładnie
reprezentowane, co prowadzi do błędów obliczeń.

4. Jaki typ błędu tutaj występuje?
Błąd reprezentacji zmiennoprzecinkowej (floating-point representation
error) jest głównym typem błędu w tych obliczeniach.
Jest to wynik ograniczonej precyzji w reprezentowaniu liczb rzeczywistych.

5. Czy istnieje możliwość poprawy działania tych wyrażeń, jeżeli tak to
w jaki sposób?
Można unikać tego rodzaju błędów, stosując odpowiednie metody zaokrąglania
lub biblioteki do obliczeń na liczbach zmiennoprzecinkowych.
Jedną z popularnych bibliotek do obliczeń o dużej precyzji jest
biblioteka decimal w Pythonie.
```

Zadanie 2.

Wartości funkcji e^x można obliczyć w przybliżeniu z szeregu Taylora w następujący sposób:

$$\mathrm{e}^x \approx \sum_{i=0}^{N} \frac{1}{i!} x^i$$

na podstawie przedstawionych informacji zaimplementuj funkcje *exponential* która oblicza e^x z zadaną dokładnością N. Porównaj działanie utworzonej funkcji z numpy.exp (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.exp.html). Odpowiedz na pytania:

1. Jaki typ błędu obrazuje omawiany przykład?
2. Dokonaj analizy błędów bezwzględnych i względnych w zależności od wartości n.

In [5]:
```python
def exponential(x,N):
    result = 0
    term = 1
    for n in range(N):
        result += term
        term *= x / (n + 1)
    return result

x = 2.0 #Przykładowa wartość x
N = 10 #Liczba wyrazów w szeregu Taylora

approx_value = exponential(x,N)
exact_value = np.exp(x)
print(f'Przyblizona wartosc: {approx_value}')
print(f'Dokladna wartosc (numpy): {exact_value}')
```

```
Przyblizona wartosc: 7.3887125220458545
Dokladna wartosc (numpy): 7.38905609893065
```

```
1. Jaki typ błędu obrazuje omawiany przykład?
Omawiany przykład obrazuje bledy bezwzgledne i wzgledne.

2. Dokonaj analizy błędów bezwzględnych i względnych w zależności od
wartości .n
```

```
Im większa wartość N, tym mniejszy jest błąd bezwzględny, co oznacza, że
przybliżenie jest dokładniejsze
```

Zadania 3.

Zaimplementuj 2 funkcje $coskx1$ i $coskx2$, realizujące rekurencyjnie przybliżanie wartości $cos(kx)$ w następujący sposób:

- Metoda 1:
  $cos(m+1)x = 2cosx \cdot cos(mx) - cos(m-1)x$
- Metoda 2:
  $cos(mx) = cosx \cdot cos(m-1)x - sinx \cdot sin(m-1)x$
  $sin(mx) = sinx \cdot cos(m-1)x + cosx \cdot sin(m-1)x$

Następnie przeanalizuj otrzymane rezultaty dla różnych k.

Wskazówka Do wyliczenia wartości $sin(x)$, $cos(x)$ (dla $k = 1$) można użyć funkcji biblioteki numpy. Pozostałe wartości dla $k > 1$ należy wyznaczyć rekurencyjnie.

In [6]:
```python
def coskx1(x, k, n):
    if n == 0:
        return np.cos(k * x)
    elif n == 1:
        return 2*np.cos(x)*np.cos(k * x)-np.cos(x)
    else:
        return 2*np.cos(x)*coskx1(x,k,n-1)-coskx1(x,k,n-2)

def coskx2(x, k, n):
    if n == 0:
        return np.cos(k*x)
    elif n == 1:
        return np.cos(x)*coskx2(x,k,n-1)-np.sin(x)*np.sin(k*x)
    else:
        return np.cos(x)*coskx2(x,k,n-1)-np.sin(x)*np.sin(k*x)

x = 1.0
k = 2
n_values = [0, 1, 2, 3]
for n in n_values:
    result1 = coskx1(x,k,n)
    result2 = coskx2(x,k,n)
    exact_value = np.cos(k*x)

print(f'n = {n}:')
print(' ')
print(f'Metoda 1: {result1}')
print(f'Metoda 2: {result2}')
print(f'Dokladna wartosc (numpy): {exact_value}')
```

```
n = 3:

Metoda 1: 0.2836621854632264
Metoda 2: -1.4675634319175765
Dokladna wartosc (numpy): -0.4161468365471424
```

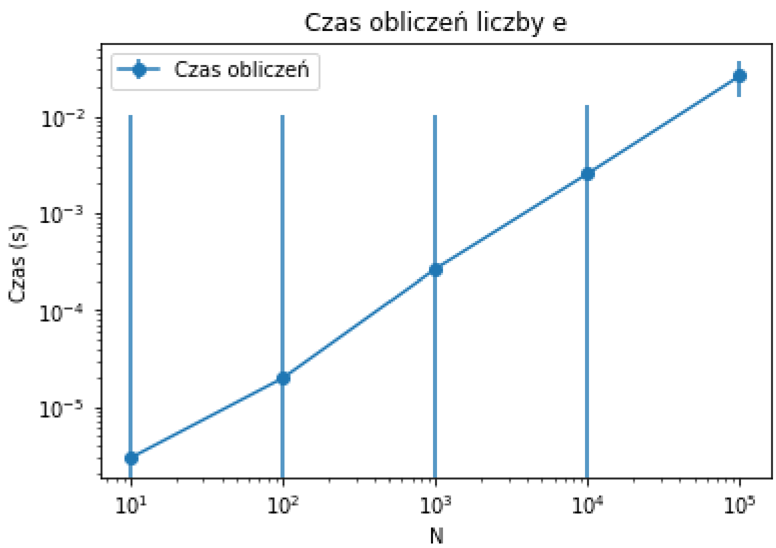
Zadanie 4.

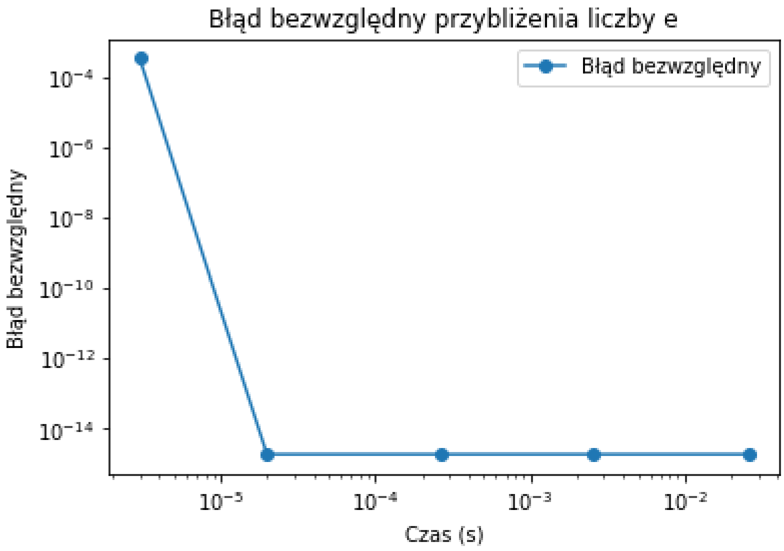
Używając funkcji timeit oraz memit zbadaj czas działania oraz zużycie pamięci funkcji z Zadania 2 w zależności od różnych wartości N.
Sporządź wykresy:

- czasu obliczenia danego przybliżenia liczby e w zależności od N. W tym celu wykorzystaj funkcje [errorbar (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.errorbar.html)](https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.errorbar.html) oraz zwracane przez *timeit* wartości średnie oraz ich odchylenie standardowe.
- błędu bezwzględnego przybliżenia liczby e od czasu jego wykonania.

Wskazówka Użyj opcji -o (output) dla *timeit* aby zapisać wynik do zmiennej. Opcje -r (runs) i -n (ilość pentli) decydują o ilości wykonanych prób.
Do wizualizacji wyników użyj skali logarytmicznej.

In [7]:
```
from functools import partial
from timeit import timeit

def measure_time(func, N, x=2.0):
    time = timeit(partial(func, x, N), number=10000, globals=globals()) / 10
    return time

def exponential(x, N):
    result = 0
    term = 1
    for n in range(N):
        result += term
        term *= x / (n + 1)
    return result

N_values = [10, 100, 1000, 10000, 100000]
times = []

for N in N_values:
    time = measure_time(exponential, N)
    times.append(time)

plt.figure(figsize=(6, 4))
plt.errorbar(N_values, times, yerr=np.std(times), fmt='o-',label='Czas oblic
plt.xlabel('N')
plt.ylabel('Czas (s)')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.title('Czas obliczeń liczby e')
plt.show()

exact_value = np.exp(2.0)
errors = [np.abs(exponential(2.0, N) - exact_value) for N in N_values]
plt.figure(figsize=(6, 4))
plt.errorbar(times, errors, fmt='o-', label='Błąd bezwzględny')
plt.xlabel('Czas (s)')
plt.ylabel('Błąd bezwzględny')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.title('Błąd bezwzględny przybliżenia liczby e')
plt.show()
```




Błąd bezwzględny przybliżenia liczby e
Błąd bezwzględny
Błąd bezwzględny
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
Czas (s)