Matematický software

Zápočtový dokument

Jméno: Ngoc Huy Vu

Kontaktní email: ngochuy.vu.99@gmail.com

Datum odevzdání: vyplňte

Odkaz na repozitář: vyplňte

1. Knihovny a moduly pro matematické výpočty

Zadání:
V tomto kurzu jste se učili s některými vybranými knihovnami. Některé sloužily pro rychlé vektorové operace, jako numpy, některé mají naprogramovány symbolické manipulace, které lze převést na numerické reprezentace (sympy), některé mají v sobě funkce pro numerickou integraci (scipy). Některé slouží i pro rychlé základní operace s čísly (numba).

Vaším úkolem je změřit potřebný čas pro vyřešení nějakého problému (např.: provést skalární součin, vypočítat určitý integrál) pomocí standardního pythonu a pomocí specializované knihovny. Toto měření proveďte alespoň pro 5 různých úloh (ne pouze jiná čísla, ale úplně jiné téma) a minimálně porovnejte rychlost jednoho modulu se standardním pythonem. Ideálně proveďte porovnání ještě s dalším modulem a snažte se, ať je kód ve standardním pythonu napsán efektivně.

Řešení:
K měření potřebného času pro provedení kódu používáme pomocí knihoven **timeit**, která má funkce **timeit.timeit()**

a) **Skalární součin**

- **Skalární součin** se definuje mezi dvěma vektory a zachycuje vztah mezi velikostí vektorů a jejich úhlem
- Máme-li dva vektory $\vec{u}(u1, u2)$ a $\vec{v}(v1, v2)$. Pak jejich skalární součin je roven:

$$\vec{u} * \vec{v} = u1v1 + u2v2$$

- Knihovna **numpy** má funkce **dot ()**, která slouží k provádění maticového násobení dvou array (matic) nebo vektorů. Syntaxe je:

  numpy.dot(a,b), kde a, b jsou pole(array)

- Řešení

```
import numpy as np
import timeit

# Standardní Python
def pomoci_standardniho_pythonu(a, b):
    result = 0
    for i in range(len(a)):
        result += a[i] * b[i]
    return result

# NumPy
def pomoci_numpy(a, b):
    return np.dot(a, b)

# Generování dat
a = np.random.rand(1000000)
b = np.random.rand(1000000)


# Měření času pro standardní Python
python_cas = timeit.timeit(lambda: pomoci_standardniho_pythonu(a, b),
number=100)
python_cas = round(python_cas, 2)
print(f"Potřebný  čas  pro  vyřešení  pomocí  standardního  Pythonu:
{python_cas} sekund")

# Měření času pro NumPy
numpy_cas = timeit.timeit(lambda: pomoci_numpy(a, b), number=100)
numpy_cas = round(numpy_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí NumPy: {numpy_cas} sekund")

#Porovnání: (stary_cas/novy_cas)/stary_cas*100
porovnani = round((python_cas-numpy_cas)/python_cas*100, 2)
print(f"Pomocí Numpy se vyřeší rychlejší o přibližně {porovnani}%")
```

b) Průměr seznamu čísel

- Generujeme náhodná data
- Funkce **mean()** slouží k vypočtení průměru
- Řešení

```
import statistics
import timeit
import numpy as np

# Standardní Python
def pomoci__strandardniho_python(data):
    return statistics.mean(data)

# NumPy
def pomoci_numpy(data):
    return np.mean(data)

# Měření času pro standardní Python
python_cas = timeit.timeit(lambda:
pomoci__strandardniho_python(np.random.rand(1000000)), number=20)
print("Standard Python (mean):", python_cas)

# Měření času pro NumPy
numpy_cas = timeit.timeit(lambda: pomoci_numpy(np.random.rand(50)),
number=1000000)
print("NumPy (mean):", numpy_cas)

#Porovnání: (stary_cas/novy_cas)/stary_cas*100
porovnani = round((python_cas-numpy_cas)/python_cas*100, 2)
print(f"Pomocí Numpy se vyřeší rychlejší o přiblížně {porovnani}%")
```

c) **Výpočet faktoriálu**

- **Faktoriál čísla n** je roven součinu všech přirozených čísel, která jsou menší nebo rovna číslu n.
- Např. 5! = 5 * 4 *3 * 2 *1 = 120
- **Funkce .prod** spočítá součin všech prvků pole
- Rešení

```
import timeit
import numpy as np

n = 100
# Standardní Python
def reseni_pomoci_standardni_python(n):
    vysledek = 1
    for i in range(1, n+1):
        vysledek *= i
    return vysledek

def reseni_pomoci_numpy(n):
    return np.prod(np.arange(1, n+1))

# Měření času pro standardní Python
python_cas = timeit.timeit(lambda: reseni_pomoci_standardni_python(n))
python_cas = round(python_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí standardního Pythonu:
{python_cas} sekund")

# Měření času pro NumPy
numpy_cas = timeit.timeit(lambda: reseni_pomoci_numpy(n))
numpy_cas = round(numpy_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí NumPy: {numpy_cas} sekund")

#Porovnání: (stary_cas/novy_cas)/stary_cas*100
porovnani = round((python_cas-numpy_cas)/python_cas*100, 2)
print(f"Pomocí Numpy se vyřeší rychlejší o přibližně {porovnani}%")
```

d) Umocnění prvků matice

- Funkce **numpy.square** slouží k umocnění prvků matice

- Řešení

```
import timeit
import numpy as np

def reseni_pomoci_standardni_python(matrix):
    n = len(matrix)
    result = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            result[i][j] = matrix[i][j] ** 2
    return result

def reseni_pomoci_numpy(matrix):
    return np.square(matrix)

matice = np.random.rand(10, 10)
# Měření času pro standardní Python
python_cas = timeit.timeit(lambda:
reseni_pomoci_standardni_python(matice))
python_cas = round(python_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí standardního Pythonu:
{python_cas} sekund")

# Měření času pro NumPy
numpy_cas = timeit.timeit(lambda: reseni_pomoci_numpy(matice))
numpy_cas = round(numpy_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí NumPy: {numpy_cas} sekund")

#Porovnání: (stary_cas/novy_cas)/stary_cas*100
porovnani = round((python_cas-numpy_cas)/python_cas*100, 2)
print(f"Pomocí Numpy se vyřeší rychlejší o přibližně {porovnani}%")
```

e) Sinus

- Funkce numpy.linspace slouží pro generování pole úhlů
- Řešení

```
import timeit
import numpy as np
import math

def reseni_pomoci_standardni_python(uhly):
    return [math.sin(uhel) for uhel in uhly]

def reseni_pomoci_numpy(uhly):
    return np.sin(uhly)

uhly = np.linspace(0, 2*np.pi, 100)

# Měření času pro standardní Python
python_cas = timeit.timeit(lambda:
reseni_pomoci_standardni_python(uhly))
python_cas = round(python_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí standardního Pythonu:
{python_cas} sekund")

# Měření času pro NumPy
numpy_cas = timeit.timeit(lambda: reseni_pomoci_numpy(uhly))
numpy_cas = round(numpy_cas, 2)
print(f"Potřebný čas pro vyřešení pomocí NumPy: {numpy_cas} sekund")

#Porovnání: (stary_cas/novy_cas)/stary_cas*100
porovnani = round((python_cas-numpy_cas)/python_cas*100, 2)
print(f"Pomocí Numpy se vyřeší rychlejší o přibližně {porovnani}%")
```

6. Generování náhodných čísel a testování generátorů

Zadání:
Tento úkol bude poněkud kreativnější charakteru. Vaším úkolem je vytvořit vlastní generátor semínka do pseudonáhodných algoritmů. Jazyk Python umí sbírat přes ovladače hardwarových zařízení různá fyzická a fyzikální data. Můžete i sbírat data z historie prohlížeče, snímání pohybu myší, vyzvání uživatele zadat náhodné úhozy do klávesnice a jiná unikátní data uživatelů.

Řešení:

- Mersenne Twister je pseudo-generátor náhodných čísel.
- Python používá Mersenne Twister pro modul **random**
- Hra „3 cây“ (tři karty)
  - ✓ Je klasická vietnamská karetní hra.
  - ✓ Používá francouzské karty, ale nepoužíváme karty s hodnotou 10, J, Q a K. Hra je pro 2 až 6 hráče. Rozdá se každému hráči 3 karet.
  - ✓ Princip výpočtu bodů je následující: karta A má hodnotu 1 bodu, karty 2, 3... 9 mají hodnotu odpovídající svému číslu. Celkový počet bodů je součet hodnot těchto karet, avšak pouze se zbytkem po dělení 10. Například pro 3 karty: 2, 3, 8 bude výsledný bodový počet 3 (součet je 13). Hráč s vyšším bodovým počtem vyhrává.
  - ✓ Pokud mají dva hráči stejný počet bodů, pak se porovnává nejvyšší karta v jejich kartách podle preferovaného pořadí sady: Kříže > Káry > Piky > Srdce. Pokud jsou karty stejného druhu, pak se porovnává jejich hodnota podle pořadí: A > 9 > 8 > ... > 2.
- Řešení

```
import random

# Vytvoření balíčku karet
suits = ['Srdce', 'Káry', 'Piky', 'Kříže']
values = ['A', '2', '3', '4', '5', '6', '7', '8', '9']
deck = [(value, suit) for suit in suits for value in values]

# Odstranění všech desítek a obrázkových karet (J, Q, K)
deck = [card for card in deck if card[0] != '10' and card[0] not in
['J', 'Q', 'K']]

# Funkce pro výpočet bodů pro tři karty
def calculate_score(cards):
    values_dict = {'A': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7':
7, '8': 8, '9': 9}
    total_score = sum([values_dict[card[0]] for card in cards])
    return total_score % 10
```

```python
# Funkce pro porovnání karet podle pravidel
def compare_cards(cards1, cards2):
    score1 = calculate_score(cards1)
    score2 = calculate_score(cards2)

    if score1 != score2:
        return score1 - score2

    # Stejné skóre, porovnání nejvyšších karet
    max_card1 = max(cards1, key=lambda card: (values.index(card[0]),
suits.index(card[1])))
    max_card2 = max(cards2, key=lambda card: (values.index(card[0]),
suits.index(card[1])))

    return (values.index(max_card1[0]), suits.index(max_card1[1])) -
(values.index(max_card2[0]), suits.index(max_card2[1]))

# Funkce pro vytvoření a rozdání karet hráčům
def deal_cards(num_players):
    random.shuffle(deck)
    hands = [deck[i:i+3] for i in range(0, num_players * 3, 3)]
    return hands

# Hlavní funkce
def main():
    num_players = int(input("Zadejte počet hráčů (maximálně 6): "))
    if num_players < 2 or num_players > 6:
        print("Neplatný počet hráčů.")
        return

    hands = deal_cards(num_players)

    for i, hand in enumerate(hands):
        print(f"Hráč {i+1}: {[f'{card[0]} {card[1]}' for card in
hand]}")

    winner = max(enumerate(hands), key=lambda x: (calculate_score(x[1]),
max([(values.index(card[0]), suits.index(card[1])) for card in x[1]])))
    print(f"Hráč {winner[0] + 1} vyhrál se skóre
{calculate_score(winner[1])}")

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

7. Metoda Monte Carlo

Zadání:
Metoda Monte Carlo představuje rodinu metod a filozofický přístup k modelování jevů, který využívá vzorkování prostoru (například prostor čísel na herní kostce, které mohou padnout) pomocí pseudonáhodného generátoru čísel. Jelikož se jedná spíše o filozofii řešení problému, tak využití je téměř neomezené. Na hodinách jste viděli několik aplikací (optimalizace portfolia aktiv, řešení Monty Hall problému, integrace funkce, aj.). Nalezněte nějaký zajímavý problém, který nebyl na hodině řešen, a získejte o jeho řešení informace pomocí metody Monte Carlo. Můžete využít kódy ze sešitu z hodin, ale kontext úlohy se musí lišit.
Řešení:

- Metoda Monte Carlo je simulační metoda založená na užití stochastických procesů a generování náhodných čísel
- Pomocí mnohonásobného opakování náhodných pokusů lze získat střední hodnotu hledané veličiny. Opakuje experiment s náhodně zvolenými daty s velkým počtem opakování za účelem získání souhrnné statistiky z výsledků experimentu.
- Moje zadání je simulace pro odhad pravděpodobnosti vytažení eso srdce z 52 hracích karet
- **Postup**
  1) Vytvořte simulaci 52 hracích karet
     Generuje náhodnou 52 hracích karet, z nichž obsahuje jednu kartu Eso Srdce a 51 dalších karet. Tento proces můžete mnohokrát opakovat, abyste vytvořili mnoho různých 52 hracích karet.
  2) Spočítejte počet tahů Eso Srdce. Při každých pokusů, zamícháme hrací karty, a táhne první kartu. Pokud první karta je Eso Srdce, zvýšíme hodnotu proměnné o 1, která je počet tahů Eso Srdce
  3) Vypočítejte poměr
     Vypočítáme pravděpodobnost. Dělíme počet tahů Eso Srdce s počtem celkových pokusů
  - Řešení

```
import random

def monte_carlo_simulace(pokus):
    tah_eso_srdce = 0
    for _ in range(pokus):
        karty = list(range(1, 14)) * 4
        random.shuffle(karty)

        if karty[0] == 1:  # eso srdce
            tah_eso_srdce += 1
    pravdepodobnost = (tah_eso_srdce  / pokus) * 100
    return pravdepodobnost
pokus = 100000
vysledek = monte_carlo_simulace(pokus)
print(f"Pravděpodobnosti k vytažení eso srdce z 52 hracích karet je
{vysledek} %")
```