Matematický software

Zápočtový dokument

Jméno:	Ngoc Huy Vu
Kontaktní email:	ngochuy.vu.99@gmail.com
Datum odevzdání:	04.09.2023
Odkaz na repozitář:	https://github.com/NgocHuyVu/Matematicky_software

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) **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}%")
```

- **Výsledek**

```
Potřebný čas pro vyřešení pomocí standardního Pythonu: 12.401021600002423 sekund
Potřebný čas pro vyřešení pomocí NumPy: 0.17210130000603385 sekund
Pomocí Numpy se vyřeší rychlejší o přiblížně 98.61%
```

b) 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}%")
```

- **Výsledek**

```
Potřebný čas pro vyřešení pomocí standardního Pythonu: 12.94 sekund
Potřebný čas pro vyřešení pomocí NumPy: 0.02 sekund
Pomocí Numpy se vyřeší rychlejší o přibližně 99.85%
```

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 přiblížně {porovnani}%")
```

- **Výsledek**

```
Potřebný čas pro vyřešení pomocí standardního Pythonu: 6.37 sekund
Potřebný čas pro vyřešení pomocí NumPy: 4.31 sekund
Pomocí Numpy se vyřeší rychlejší o přiblížně 32.34%
```

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}%")
```

- **Výsledek**

```
Potřebný čas pro vyřešení pomocí standardního Pythonu: 34.0 sekund
Potřebný čas pro vyřešení pomocí NumPy: 0.5 sekund
Pomocí Numpy se vyřeší rychlejší o přibližně 98.53%
```

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}%")
```

- **Výsledek**

```
Potřebný čas pro vyřešení pomocí standardního Pythonu: 16.75 sekund
Potřebný čas pro vyřešení pomocí NumPy: 1.17 sekund
Pomocí Numpy se vyřeší rychlejší o přibližně 93.01%
```

f) Výpočet určitého integrálu

```
import sympy as sp
import timeit

# Symbolická proměnná
x = sp.symbols('x')

zadani = input("Zadejte funkci f(x) např. x**2 + 2*x + 1 : ")

try:
    f = sp.sympify(zadani)
except sp.SympifyError:
    print("Chyba při zpracování vstupu.")
    exit(1)

# interval určítého inegrálu (a,b)
a = 1
b = 5

def pomoci_sympy():
    return sp.integrate(f, (x, a, b))

def pomoci_strandardniho_python():
    # Definice diskretizace intervalu a krok pro sumaci
    n = 10000
    dx = (b - a) / n

    # Sumace hodnot funkce na diskretizovaném intervalu
    integral = sum(f.subs(x, a + i * dx) for i in range(n))

    # Násobení krokem pro výpočet určitého integrálu
    return integral * dx

# Měření času pro standardní Python
python_cas = timeit.timeit(lambda: pomoci_strandardniho_python)
print(f"Potřebný čas pro vyřešení pomocí standardního Pythonu: {python_cas}
sekund")
# Měření času pro NumPy
numpy_cas = timeit.timeit(lambda: pomoci_sympy)
print(f"Potřebný čas pro vyřešení pomocí SymPy: {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 přiblížně {porovnani}%")
```

- Výsledek

```
Potřebný čas pro vyřešení pomocí standardního Pythonu: 0.04297979999682866 sekund
Potřebný čas pro vyřešení pomocí SymPy: 0.04294169999775477 sekund
Pomocí Numpy se vyřeší rychlejší o přiblížně 0.09%
```

2. Vizualizace dat

Zadání:
V jednom ze cvičení jste probírali práci s moduly pro vizualizaci dat. Mezi nejznámější moduly patří matplotlib (a jeho nadstavby jako seaborn), pillow, opencv, aj. Vyberte si nějakou zajímavou datovou sadu na webovém portále Kaggle a proveďte datovou analýzu datové sady. Využijte k tomu různé typy grafů a interpretujte je (minimálně alespoň 5 zajímavých grafů). Příklad interpretace: z datové sady pro počasí vyplynulo z liniového grafu, že v létě je vyšší rozptyl mezi minimální a maximální hodnotou teploty. Z jiného grafu vyplývá, že v létě je vyšší průměrná vlhkost vzduchu. Důvodem vyššího rozptylu může být absorpce záření vzduchem, který má v létě vyšší tepelnou kapacitu.

Řešení:

Analýza a vizualizace dat o fotbalových hráčích a nejlepších ligách z hlediska počtu gólů a očekávaných gól (xG)

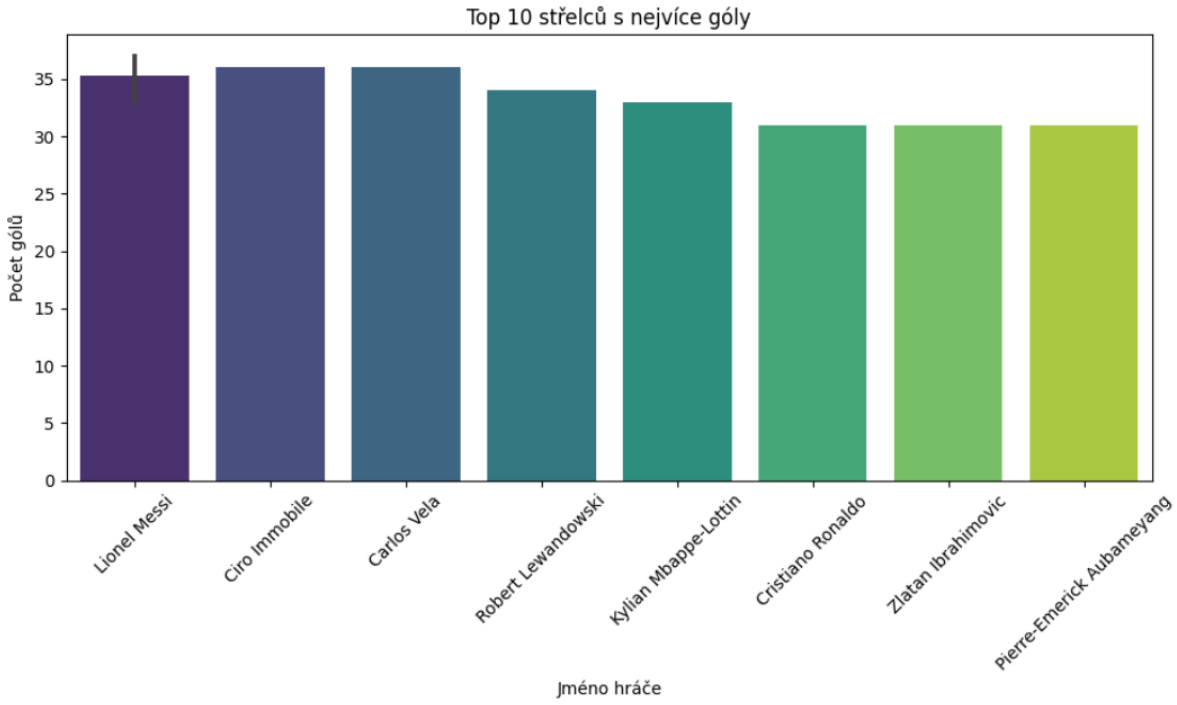
- Zdroj databáze od Kaggle: https://www.kaggle.com/datasets/mohamedhanyyy/top-football-leagues-scorers
- **Databáze**
  - ✓ Je ve formátu CSV.
  - ✓ Obsahuje **počet gólů a očekáváný gól hráčů** z ligách Premier League (Anglie), Ligue 1 (Francie), Bundesliga (Německo), Seria A (Itálie), Campeonato Brasileiro Série A, MLS (USA), Eredivisie (Nizozemsko).
  - ✓ Obsahuje pouze data o 3 sezonách 2016/2017, 2017/2018 a 2018/2019.
- **Moduly pro vizualizaci dat**
  - ✓ **Matplotlib.pyplot** slouží k vytváření široké škály grafů a vizualizací
  - ✓ **Seaborn** jedná se o nadstavbu nad knihovnou Matplotlib, která usnadňuje vytváření atraktivních a informativních grafů. Seaborn poskytuje jednoduché funkce pro vytváření různých typů grafů, jako jsou sloupcové grafy, krabicové grafy, teplotní mapy a další. Tato knihovna také nabízí přednastavené barevné palety a stylizaci grafů, což usnadňuje tvorbu profesionálně vypadajících vizualizací.
  - ✓ Pomocí používání obou modulů může být použit pro zdokonalení vzhledu grafů.

- **Importování knihovnů a načtení dat**

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image, ImageDraw, ImageFont
import cv2

data = pd.read_csv("Data.csv")
```

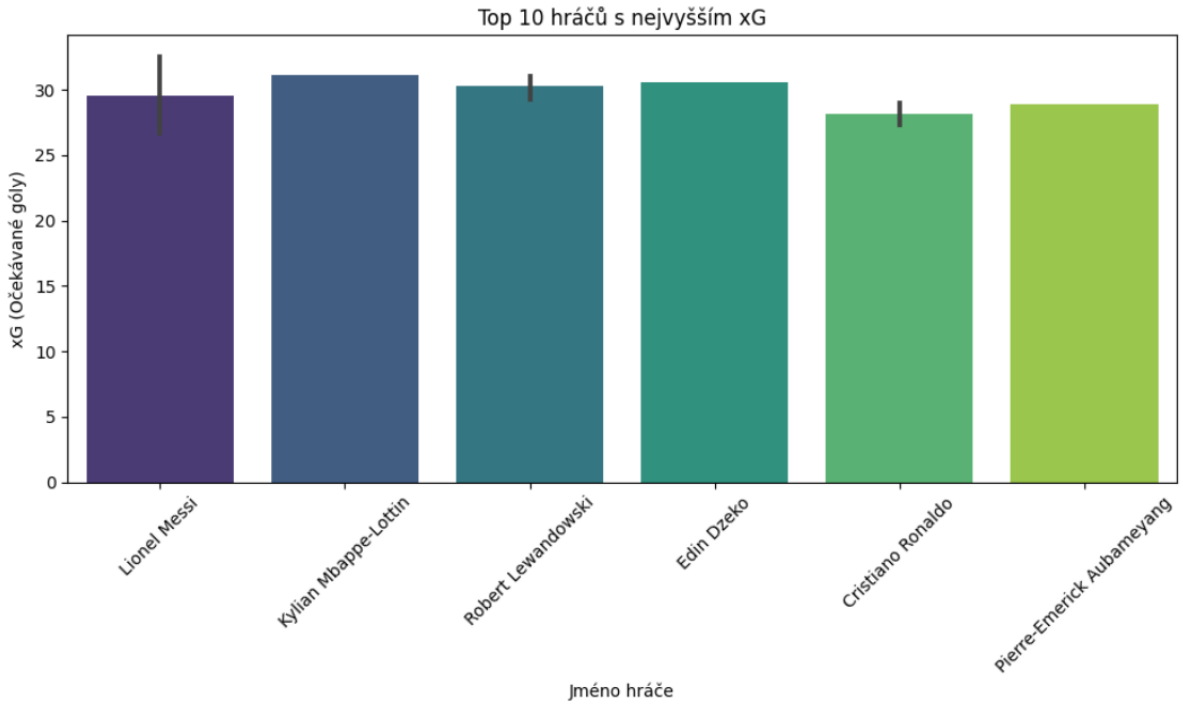
- **Graf 1: Top 10 střelců s nejvíce góly**

```
# Graf 1: Top 10 střelců s nejvíce góly
top10_scorers = data.sort_values(by="Goals", ascending=False).head(10)
plt.figure(figsize=(10, 6))
sns.barplot(x="Player Names", y="Goals", data=top10_scorers,
palette="viridis")
plt.xlabel("Jméno hráče")
plt.ylabel("Počet gólů")
plt.title("Top 10 střelců s nejvíce góly")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



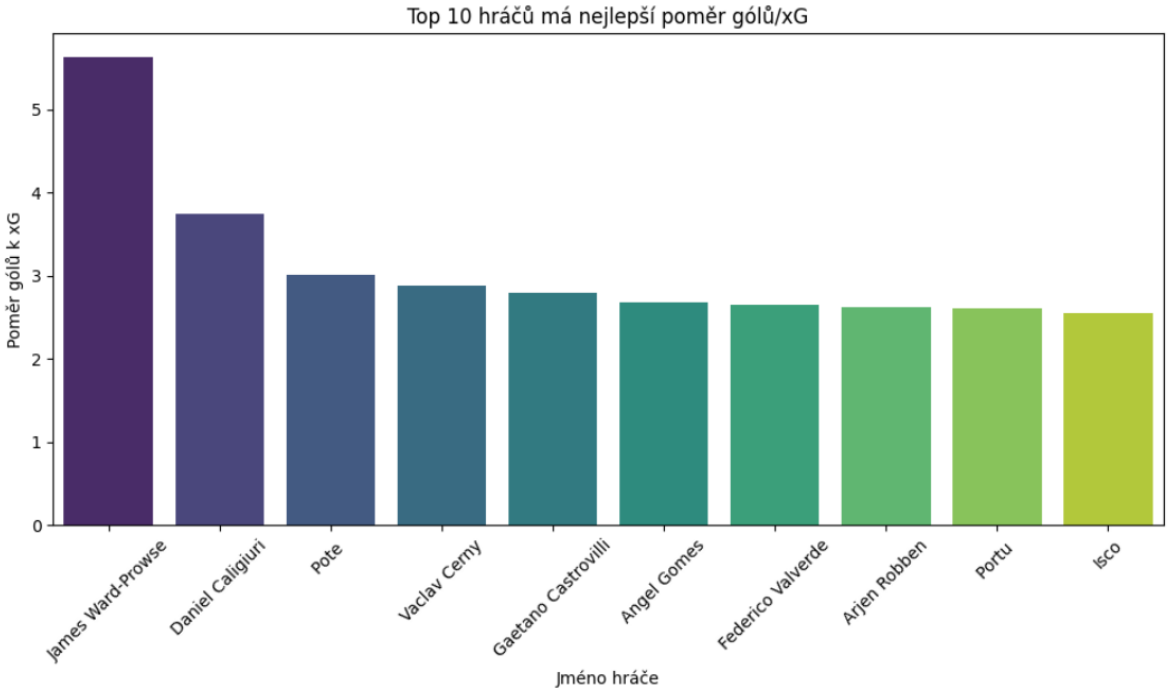
- **Graf 2: Top 10 hráčů s nejvyšším xG**

```
# Graf 2: Top 10 hráčů s nejvyšším xG
top10_xG_scorers = data.sort_values(by="xG", ascending=False).head(10)
plt.figure(figsize=(10, 6))
sns.barplot(x="Player Names", y="xG", data=top10_xG_scorers,
palette="viridis")

plt.xlabel("Jméno hráče")
plt.ylabel("xG (Očekávané góly)")
plt.title("Top 10 hráčů s nejvyšším xG")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



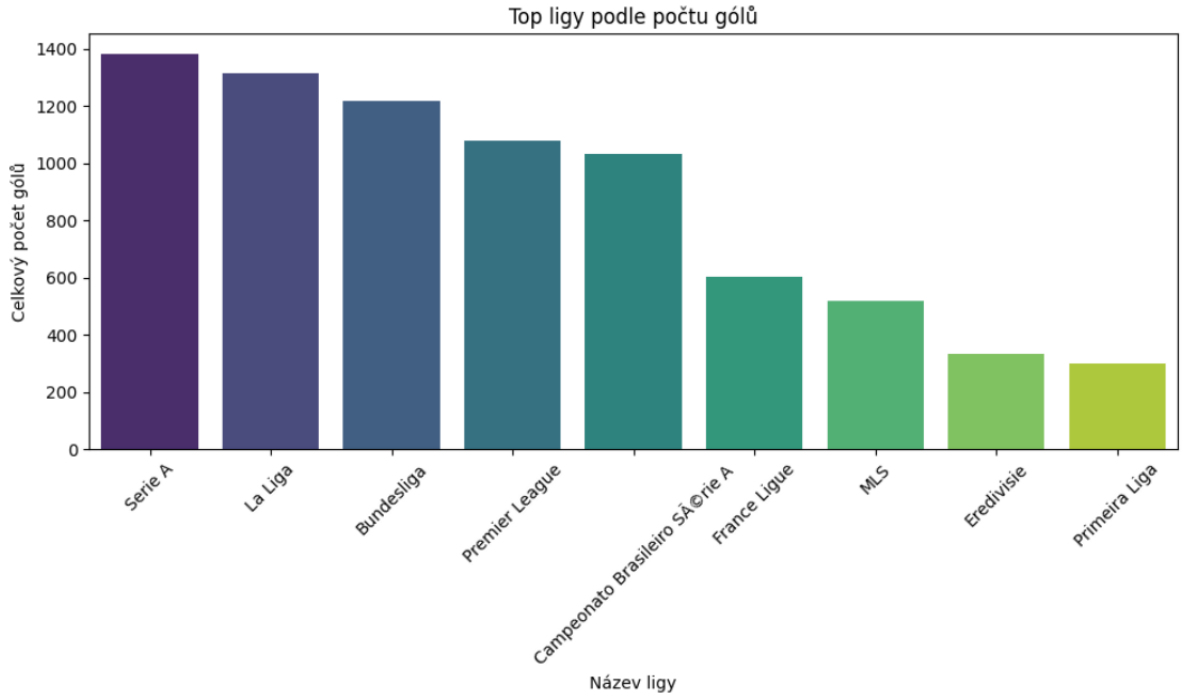
- **Graf 3: Top 10 hráčů s nejlepším poměrem gólů k očekávaným gólům (xG)**

```
# Graf 3: Top 10 hráčů má nejlepší poměr gólů/xG
data["Goal Conversion Rate"] = data["Goals"] / data["xG"]
top10_scorers_conversion = data.sort_values(by="Goal Conversion Rate",
ascending=False).head(10)
plt.figure(figsize=(10, 6))
sns.barplot(x="Player Names", y="Goal Conversion Rate",
data=top10_scorers_conversion, palette="viridis")

plt.xlabel("Jméno hráče")
plt.ylabel("Poměr gólů k xG")
plt.title("Top 10 hráčů má nejlepší poměr gólů/xG")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



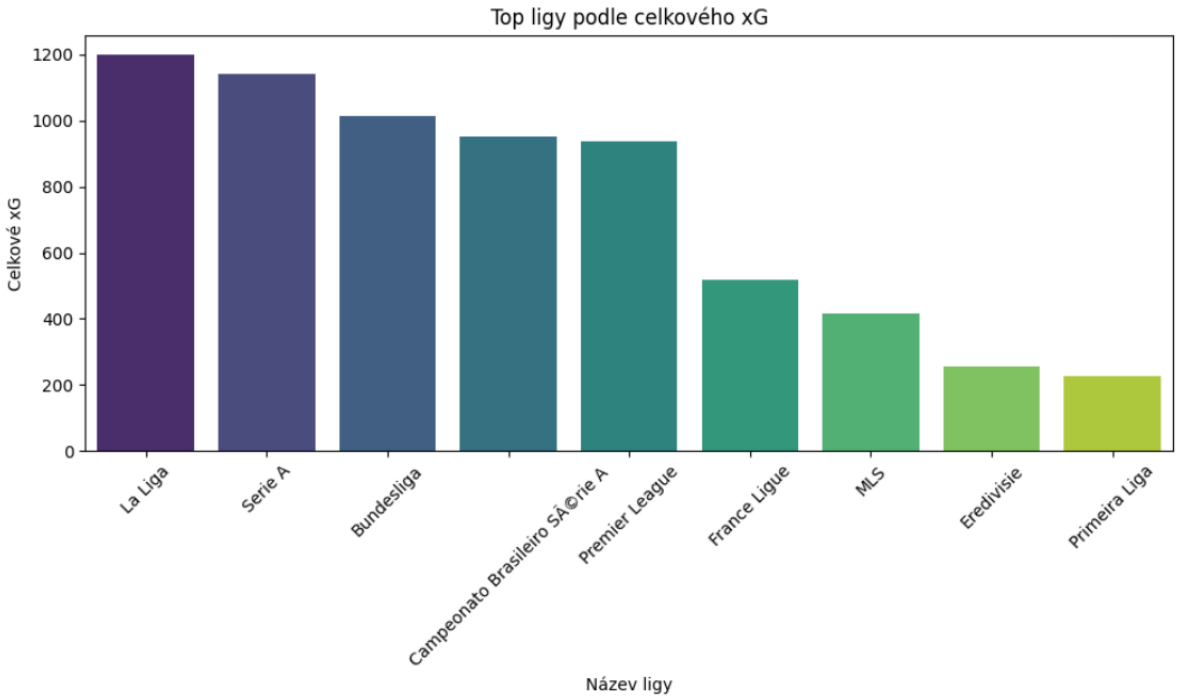
- **Graf 4: Top ligy podle celkového počtu gólů**

```
# Graf 4: Top ligy podle počtu gólů
total_goals_by_league =
data.groupby("League")["Goals"].sum().reset_index()
top_leagues = total_goals_by_league.sort_values(by="Goals",
ascending=False).head(10)

plt.figure(figsize=(10, 6))
sns.barplot(x="League", y="Goals", data=top_leagues, palette="viridis")
plt.xlabel("Název ligy")
plt.ylabel("Celkový počet gólů")
plt.title("Top ligy podle počtu gólů")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



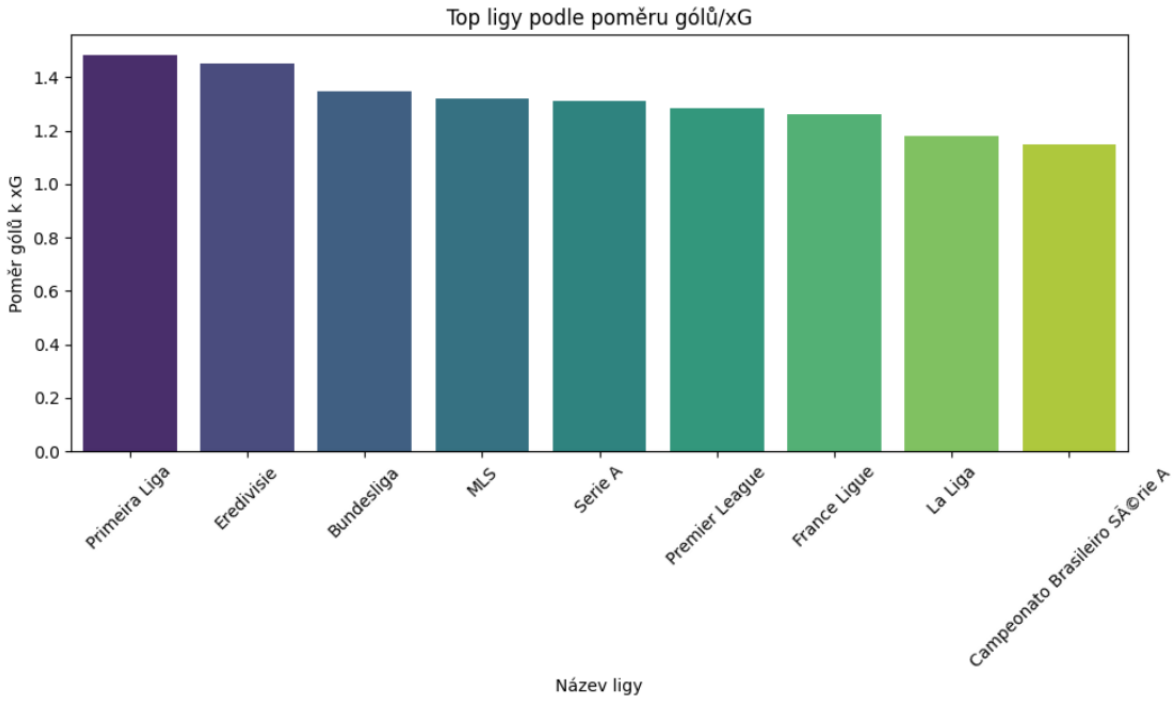
- **Graf 5: Top ligy podle celkového očekávaného počtu gólů (xG)**

```
# Graf 5: Top ligy podle celkového očekávaného počtu gólů (xG)
total_xG_by_league = data.groupby("League")["xG"].sum().reset_index()
top_leagues = total_xG_by_league.sort_values(by="xG",
ascending=False).head(10)

plt.figure(figsize=(10, 6))
sns.barplot(x="League", y="xG", data=top_leagues, palette="viridis")
plt.xlabel("Název ligy")
plt.ylabel("Celkové xG")
plt.title("Top ligy podle celkového xG")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



- **Graf 6: Top ligy podle průměrného poměru gólů k očekávaným gólům (xG)**

```
# Graf 6: Top ligy podle průměrného poměru gólů k očekávaným gólům (xG)
data["Goal Conversion Rate"] = data["Goals"] / data["xG"]
conversion_rate_by_league = data.groupby("League")["Goal Conversion
Rate"].mean().reset_index()
top_leagues = conversion_rate_by_league.sort_values(by="Goal Conversion
Rate", ascending=False).head(10)
plt.figure(figsize=(10, 6))
sns.barplot(x="League", y="Goal Conversion Rate", data=top_leagues,
palette="viridis")
plt.xlabel("Název ligy")
plt.ylabel("Poměr gólů k xG")
plt.title("Top ligy podle poměru gólů/xG")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



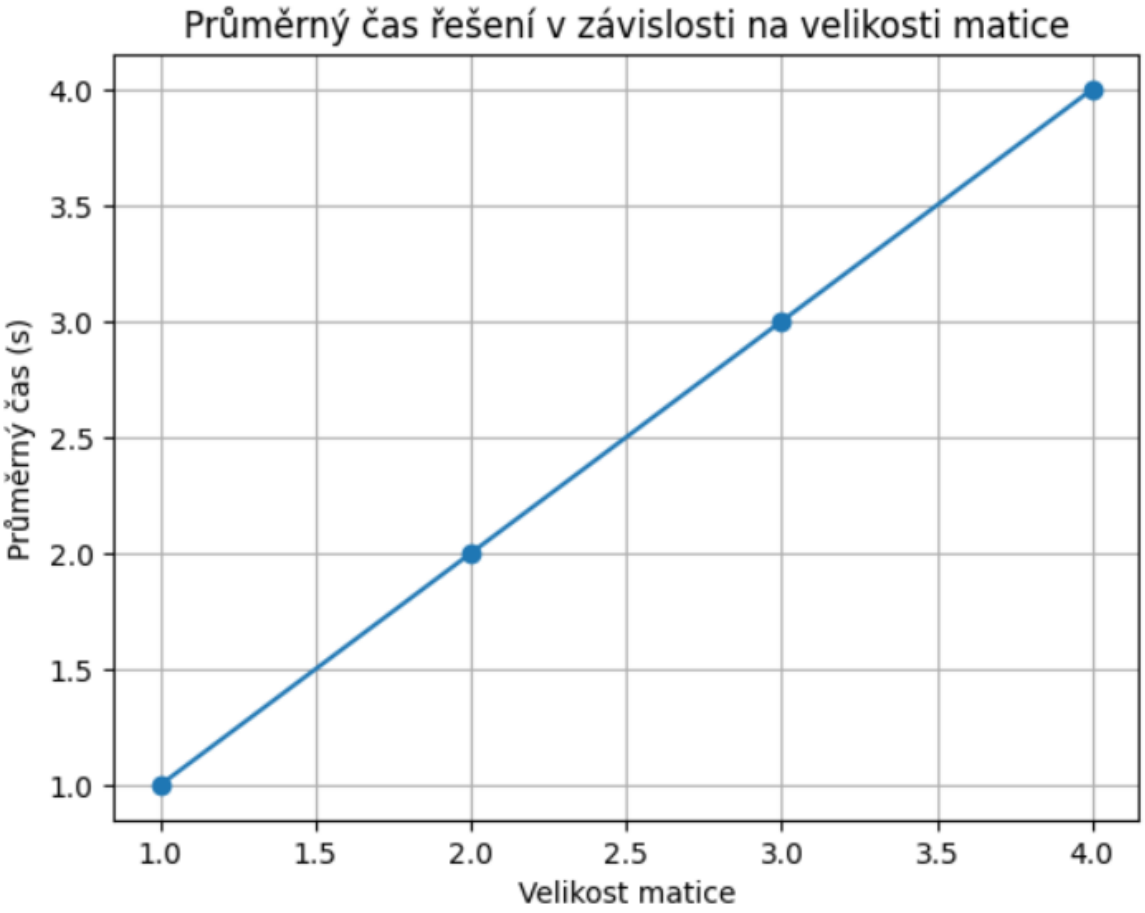
3. Úvod do lineární algebry

Zadání:
Důležitou částí studia na přírodovědecké fakultě je podobor matematiky zvaný lineární algebra. Poznatky tohoto oboru jsou základem pro oblasti jako zpracování obrazu, strojové učení nebo návrh mechanických soustav s definovanou stabilitou. Základní úlohou v lineární algebře je nalezení neznámých v soustavě lineárních rovnic. Na hodinách jste byli obeznámeni s přímou a iterační metodou pro řešení soustav lineárních rovnic. Vaším úkolem je vytvořit graf, kde na ose x bude velikost čtvercové matice a na ose y průměrný čas potřebný k nalezení uspokojivého řešení. Cílem je nalézt takovou velikost matice, od které je výhodnější využít iterační metodu.
Řešení:

- K vypočítání **determinantu matice** používá funkce **np.linalg.det(A)**, kterou je funkce v knihovně **Numpy**.
- Řešení vytvoří náhodné matice různých velikostí (1x1 do 5x5) a poté měří průměrný čas výpočtu determinantu těchto matic pro každou velikost. Pro vypočítání průměrného času výpočtu opakujeme pětkrát.
- K vykreslení grafu používá knihovnu **Matplotlib**.

```
import numpy as np
import time
import matplotlib.pyplot as plt

def vyreseni_matic(velikost_matic):
    matice = np.random.rand(velikost_matic, velikost_matic)
    result = np.linalg.det(matice) # Determinant matic
    time.sleep(velikost_matic)
    return result

velikost_matic = list(range(1, 5))  # Velikosti matic od 1x1 do 10x10
prumerny_cas = []

for size in velikost_matic:
    casy = []
    for i in range(5):
        start = time.time()
        vyreseni_matic(size)
        end = time.time()
        casy.append(end - start)
    prumerny_cas.append(np.mean(casy))
# Vykreslení grafu
plt.plot(velikost_matic, prumerny_cas, marker='o')
plt.title('Průměrný čas řešení v závislosti na velikosti matice')
plt.xlabel('Velikost matice')
plt.ylabel('Průměrný čas (s)')
plt.grid(True)
plt.show()
```

- **Výsledek**


Průměrný čas řešení v závislosti na velikosti matice
Průměrný čas (s)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Velikost matice

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
```

```
# 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()
```

- **Výsledek**

```
Hráč 1: ['2 Kříže', '6 Piky', '2 Káry']
Hráč 2: ['5 Piky', 'A Káry', '4 Kříže']
Hráč 3: ['8 Srdce', '3 Káry', '6 Kříže']
Hráč 4: ['3 Piky', '4 Káry', '7 Kříže']
Hráč 5: ['5 Srdce', '9 Káry', '4 Piky']
Hráč 6: ['7 Srdce', 'A Piky', '2 Srdce']
Hráč 5 vyhrál se skóre 8
```

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) **Vytvoříme 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 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} %")
```

- **Výsledek**

```
Pravděpodobnosti k vytažení eso srdce z 52 hracích karet je 7.639 %
```