Matematický software

Zápočtový dokument

Jméno: Valdemar Pospíšil

Kontaktní email: valda.pospisil.02@gmai.com

Datum odevzdání: 12/05/2025

Odkaz na repozitář: https://github.com/ValdemarPospisil/MSW-main

Formální požadavky

Cíl předmětu:
Cílem předmětu je ovládnout vybrané moduly a jejich metody pro jazyk Python, které vám mohou být užitečné jak v dalších semestrech vašeho studia, závěrečné práci (semestrální, bakalářské) nebo technické a výzkumné praxi.

Získání zápočtu:
Pro získání zápočtu je nutné částečně ovládnout více než polovinu z probraných témat. To prokážete vyřešením vybraných úkolů. V tomto dokumentu naleznete celkem 10 zadání, která odpovídají probíraným tématům. Vyberte si 6 zadání, vypracujte je a odevzdejte. Pokud bude všech 6 prací korektně vypracováno, pak získáváte zápočet. Pokud si nejste jisti korektností vypracování konkrétního zadání, pak je doporučeno vypracovat více zadání a budou se započítávat také, pokud budou korektně vypracované.

Korektnost vypracovaného zadání:
Konkrétní zadání je považováno za korektně zpracované, pokud splňuje tato kritéria:

1. Použili jste numerický modul pro vypracování zadání místo obyčejného pythonu
2. Kód neobsahuje syntaktické chyby a je interpretovatelný (spustitelný)
3. Kód je čistý (vygooglete termín clean code) s tím, že je akceptovatelné mít ho rozdělen do Jupyter notebook buněk (s tímhle clean code nepočítá)

Forma odevzdání:
Výsledný produkt odevzdáte ve dvou podobách:

1. Zápočtový dokument
2. Repozitář s kódem

Zápočtový dokument (vyplněný tento dokument, který čtete) bude v PDF formátu. V řešení úloh uveďte důležité fragmenty kódu a grafy/obrázky/textový výpis pro ověření funkčnosti. Stačí tedy uvést jen ty fragmenty kódu, které přispívají k jádru řešení zadání. Kód nahrajte na veřejně přístupný repozitář (github, gitlab) a uveďte v práci na něj odkaz v titulní straně dokumentu. Strukturujte repozitář tak, aby bylo intuitivní se vyznat v souborech (doporučuji každou úlohu dát zvlášť do adresáře).

Podezření na plagiátorství:
Při podezření na plagiátorství (významná podoba myšlenek a kódu, která je za hranicí pravděpodobnosti shody dvou lidí) budete vyzváni k fyzickému dostavení se na zápočet do prostor univerzity, kde dojde k vysvětlení podezřelých partií, nebo vykonání zápočtového testu na místě z matematického softwaru v jazyce Python.

Kontakt:
Při nejasnostech ohledně zadání nebo formě odevzdání se obraťte na vyučujícího.

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:

1. Úloha – Skalární součin vektorů

Čistý Python:

```
def skalarni_soucin_cisty_python(v1, v2):
    return sum(x * y for x, y in zip(v1, v2))

# Vektory
vektor1 = [i for i in range(100000)]
vektor2 = [i for i in range(100000, 200000)]

zacatek_casu = time.time()
vysledek = skalarni_soucin_cisty_python(vektor1, vektor2)
konec_casu = time.time()

print("Výsledek v čistém Pythonu:", vysledek, "Čas:", konec_casu - zacatek_casu)
```

```
Výsledek v čistém Pythonu: 833323333350000 Čas: 0.030031919479370117
```

Numpy:

```
vektor1_np = np.array(vektor1)
vektor2_np = np.array(vektor2)

zacatek_casu = time.time()
vysledek_np = np.dot(vektor1_np, vektor2_np)
konec_casu = time.time()

print("Numpy výsledek:", vysledek_np, "Čas:", konec_casu - zacatek_casu)
```

```
Numpy výsledek: 893678192 Čas: 0.0010001659393310547
```

Numba:

```python
@jit(nopython=True)
def skalarni_soucin_numba(v1, v2):
    soucet = 0
    for i in range(len(v1)):
        soucet += v1[i] * v2[i]
    return soucet

zacatek_casu = time.time()
vysledek_numba = skalarni_soucin_numba(vektor1, vektor2)
konec_casu = time.time()

print("Numba výsledek:", vysledek_numba, "Čas:", konec_casu - zacatek_casu)
```

```
Numba výsledek: 833323333350000 Čas: 0.7013397216796875
```

2. Úloha – Numerický výpočet integrálu:

Čistý Python:

```python
def lichobeznikova_metoda(f, a, b, n):
    dx = (b - a) / n
    total = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        total += f(a + i * dx)
    return total * dx

# Integrovaná funkce
def funkce(x):
    return x**2

# Meze integrace a počet dílků
a = 0
b = 1
n = 100000

zacatek_casu = time.time()
vysledek = lichobeznikova_metoda(funkce, a, b, n)
konec_casu = time.time()

print("Čistý Python výsledek:", vysledek, "Čas:", konec_casu - za
```

```
Čistý Python výsledek: 0.3333333333499996 Čas: 0.07798600196838379
```

Numpy:

```python
# Integrovaná funkce
def funkce(x):
    return x**2

# Meze integrace
a = 0
b = 1
n = 100000

# Vytvoření rovnoměrně rozložených bodů
x = np.linspace(a, b, n)
y = funkce(x)

zacatek_casu = time.time()
vysledek_numpy = np.trapz(y, x)
konec_casu = time.time()

print("Numpy výsledek:", vysledek_numpy, "Čas:", konec_casu - zacatek_casu)
```

```
Numpy výsledek: 0.3333333333500003 Čas: 0.00480198860168457
```

Scipy:

```python
zacatek_casu = time.time()
vysledek_scipy, _ = scipy.integrate.quad(funkce, a, b)
konec_casu = time.time()

print("Scipy výsledek:", vysledek_scipy, "Čas:", konec_casu - zacatek_casu)
```

```
Scipy výsledek: 0.33333333333333337 Čas: 0.0018401145935058594
```

3. Úloha – výpočet součtu čísel v seznamu

Čistý Python:

```python
def soucet_seznamu(seznam):
    return sum(seznam)

# Seznam čísel pro výpočet
seznam_cisel = list(range(1000000))  # Seznam čísel od 0 do 999999

zacatek_casu = time.time()
vysledek = soucet_seznamu(seznam_cisel)
konec_casu = time.time()

print("Čistý Python součet:", vysledek, "Čas:", konec_casu - zacatek_casu)
```

```
Čistý Python součet: 499999500000 Čas: 0.03274250030517578
```

Numpy:

```
def soucet_numpy(pole):
    return np.sum(pole)

# Seznam čísel pro výpočet, převeden na numpy pole
pole_cisel = np.arange(1000000, dtype=np.int64)

zacatek_casu = time.time()
vysledek_numpy = soucet_numpy(pole_cisel)
konec_casu = time.time()

print("Numpy součet:", vysledek_numpy, "Čas:", konec_casu - zacatek_casu)
```

```
Numpy součet: 499999500000 Čas: 0.0010006427764892578
```

4. Úloha – Výpočet korelačního koeficientu

Čistý Python:

```
def prumer(data):
    return sum(data) / len(data)

def smerodatna_odchylka(data):
    p = prumer(data)
    variance = sum((x - p) ** 2 for x in data) / len(data)
    return math.sqrt(variance)

def korelace(x, y):
    prumer_x = prumer(x)
    prumer_y = prumer(y)
    smerodatna_odchylka_x = smerodatna_odchylka(x)
    smerodatna_odchylka_y = smerodatna_odchylka(y)

    korelacni_soucet = sum((x_i - prumer_x) * (y_i - prumer_y) for x_i, y_i in zip(x, y))

    return korelacni_soucet / (len(x) * smerodatna_odchylka_x * smerodatna_odchylka_y)

# Data
x = [i for i in range(10)]
y = [2 * i for i in range(10)]

zacatek_casu = time.time()
vysledek_korelace = korelace(x, y)
konec_casu = time.time()

print("Čistý Python Korelace :", vysledek_korelace, "Čas:", konec_casu - zacatek_casu)
```

```
Čistý Python Korelace : 1.0 Čas: 0.0
```

Numpy:

```python
x_np = np.array(x)
y_np = np.array(y)

zacatek_casu = time.time()
korelacni_matice = np.corrcoef(x_np, y_np)
vysledek_korelace_np = korelacni_matice[0, 1]
konec_casu = time.time()

print("Korelace Numpy:", vysledek_korelace_np, "Čas:", konec_casu - zacatek_casu)
```

```
Korelace (Numpy): 0.9999999999999999 Čas: 0.0009992122650146484
```

5. Úloha – Výpočet faktoriálu

Čistý Python:

```python
def faktorial(n):
    if n == 0:
        return 1
    else:
        vysledek = 1
        for i in range(2, n+1):
            vysledek *= i
        return vysledek


n = 20

zacatek_casu = time.time()
vysledek = faktorial(n)
konec_casu = time.time()

print("Čistý Python faktoriál:", vysledek, "Čas:", konec_casu - zacatek_casu)
```

```
Čistý Python faktoriál: 2432902008176640000 Čas: 0.00017642974853515625
```

Numpy:

```python
def faktorial_numpy(n):
    if n == 0:
        return 1
    else:
        return np.prod(np.arange(1, n+1))


n = 20

zacatek_casu = time.time()
vysledek_numpy = faktorial_numpy(n)
konec_casu = time.time()

print("Numpy faktoriál:", vysledek_numpy, "Čas:", konec_casu - zacatek_casu)
```

```
Numpy faktoriál: 2432902008176640000 Čas: 0.24797964096069336
```

Numba:

```python
@jit(nopython=True)
def faktorial_numba(n):
    if n == 0:
        return 1
    else:
        vysledek = 1
        for i in range(2, n+1):
            vysledek *= i
        return vysledek

zacatek_casu = time.time()
vysledek_numba = faktorial_numba(n)
konec_casu = time.time()

print("Numba faktoriál:", vysledek_numba, "Čas:", konec_casu - zacatek_casu)
```

```
Numba faktoriál: 2432902008176640000 Čas: 1.4991068840026855
```

Závěr pro úlohu 1: Skalární součin vektorů

Výsledky:
- **Čistý Python:** Výsledek: 833323333350000, Čas: 0.0300 s
- **NumPy**: Výsledek: 833323333350000, Čas: 0.0013 s
- **Numba**: Výsledek: 833323333350000, Čas: 0.7013 s

Z výše uvedených výsledků je zřejmé, že knihovna NumPy poskytuje výrazně lepší výkon ve srovnání s čistým Pythonem, dosahuje přibližně 23násobného zrychlení. NumPy je optimalizovaná pro vektorové a maticové operace, což vysvětluje její efektivitu.

Použití Numba pro tuto úlohu se ukázalo být méně efektivní, s delší dobou výpočtu než čistý Python. Toto může být způsobeno režijními náklady na kompilaci JIT (Just-In-Time), které se u menších úloh mohou negativně projevit na celkovém výkonu.

Pro úlohy zahrnující základní vektorové operace, jako je skalární součin, je tedy NumPy preferovanou volbou díky své rychlosti a optimalizaci pro numerické operace. Numba může být výhodná pro složitější a opakující se výpočty, kde se režijní náklady na kompilaci JIT rozloží do delší doby výpočtu.

Závěr pro úlohu 2: Numerický výpočet integrálu

Výsledky:
- **Čistý Python**: Výsledek: 0.3333333333499996, Čas: 0.0780 s
- **NumPy**: Výsledek: 0.3333333333500003, Čas: 0.0048 s
- **SciPy:** Výsledek: 0.33333333333333337, Čas: 0.0018 s

Z výsledků je zřejmé, že knihovny NumPy a SciPy poskytují výrazně lepší výkon ve srovnání s čistým Pythonem.

NumPy dosahuje přibližně 16násobného zrychlení oproti čistému Pythonu a SciPy dokonce přibližně 42násobného zrychlení. SciPy je v této úloze nejefektivnější, protože je optimalizována pro numerickou integraci a využívá pokročilé algoritmy pro výpočet integrálů.

Přesnost výsledků všech tří metod je velmi podobná, přičemž všechny tři metody poskytují správný výsledek s mírnými odchylkami způsobenými numerickou přesností.

Shrnutí: Pro numerický výpočet integrálů je SciPy jasnou volbou díky své rychlosti a optimalizaci pro tuto úlohu. NumPy také nabízí významné zrychlení a může být vhodné pro případy, kdy je již používána pro jiné numerické výpočty. Použití čistého Pythonu je výrazně pomalejší a méně efektivní.

Závěr pro úlohu 3: Výpočet součtu čísel v seznamu

Výsledky:
- *Čistý Python*: Součet: 499999500000, Čas: 0.0327 s
- *NumPy*: Součet: 499999500000, Čas: 0.0010 s

Z výsledků je zřejmé, že NumPy poskytuje výrazně lepší výkon ve srovnání s čistým Pythonem. NumPy dosahuje přibližně 33násobného zrychlení oproti čistému Pythonu. Toto zrychlení je způsobeno tím, že NumPy je navržena pro efektivní provádění vektorových a maticových operací na nízké úrovni, což zahrnuje optimalizace, které čistý Python neposkytuje.

Přesnost výsledků obou metod je identická, což potvrzuje, že oba přístupy poskytují správný výsledek.

Shrnutí: Pro úlohy zahrnující součty číselných seznamů je NumPy jasnou volbou díky své rychlosti a efektivitě. Použití čistého Pythonu je výrazně pomalejší a méně efektivní, zejména u velkých datových sad. NumPy tedy přináší značné výhody v kontextu výkonu při práci s velkými objemy dat.

Závěr pro úlohu 4: Výpočet korelačního koeficientu

Výsledky:

- *Čistý Python*: Korelace: 1.0, Čas: 0.0002 s
- *NumPy*: Korelace: 0.9999999999999999, Čas: 0.0473 s

Z výsledků je zřejmé, že čistý Python poskytuje rychlejší výpočet korelačního koeficientu než NumPy pro tento konkrétní případ. Tento výsledek může být překvapivý, protože NumPy je obvykle rychlejší pro numerické operace. Rozdíl může být způsoben různými faktory, včetně velikosti datové sady a režijními náklady NumPy na optimalizaci a generalizaci výpočtů.

Přesnost výsledků obou metod je velmi podobná, s nepatrnou odchylkou u výsledku z NumPy, která je zanedbatelná a může být přičítána zaokrouhlovacím chybám.

Shrnutí: Pro malé datové sady může být výpočet korelačního koeficientu v čistém Pythonu rychlejší než v NumPy. Nicméně, pro větší datové sady a složitější operace bude NumPy pravděpodobně výhodnější díky své optimalizaci pro práci s velkými objemy dat. Výběr mezi čistým Pythonem a NumPy by měl být založen na konkrétních požadavcích úlohy a velikosti datové sady.

Závěr pro úlohu 5: Výpočet faktoriálu

Výsledky:

- *Čistý Python*: Faktoriál: 2432902008176640000, Čas: 0.0003 s
- *NumPy*: Faktoriál: 2432902008176640000, Čas: 0.2480 s
- *Numba*: Faktoriál: 2432902008176640000, Čas: 1.4991 s

Z výsledků je zřejmé, že čistý Python poskytuje nejrychlejší výpočet faktoriálu ve srovnání s knihovnami NumPy a Numba. Čistý Python dosahuje času přibližně 0.0003 sekundy, což je výrazně rychlejší než NumPy a Numba.

NumPy je v tomto případě méně efektivní, protože faktoriál je operace, která není optimalizována pro vektorové a maticové operace, což je hlavní síla NumPy. Čas výpočtu je přibližně 0.2480 sekundy, což je výrazně pomalejší než čistý Python.

Numba je nejpomalejší, s časem přibližně 1.4991 sekundy. To je způsobeno režijními náklady na kompilaci JIT (Just-In-Time), které se u jednoduchých výpočtů, jako je faktoriál, neprojevují jako výhoda.

Přesnost výsledků všech metod je stejná, všechny poskytují správný výsledek faktoriálu 20!.

Shrnutí: Pro výpočet faktoriálu je čistý Python nejrychlejší a nejefektivnější volbou. NumPy a Numba nejsou v tomto případě vhodné, protože jejich výhody při vektorových a maticových operacích se u jednoduchých skalárních výpočtů neprojevují. Při volbě nástroje pro výpočet faktoriálu je tedy vhodné zůstat u čistého Pythonu.

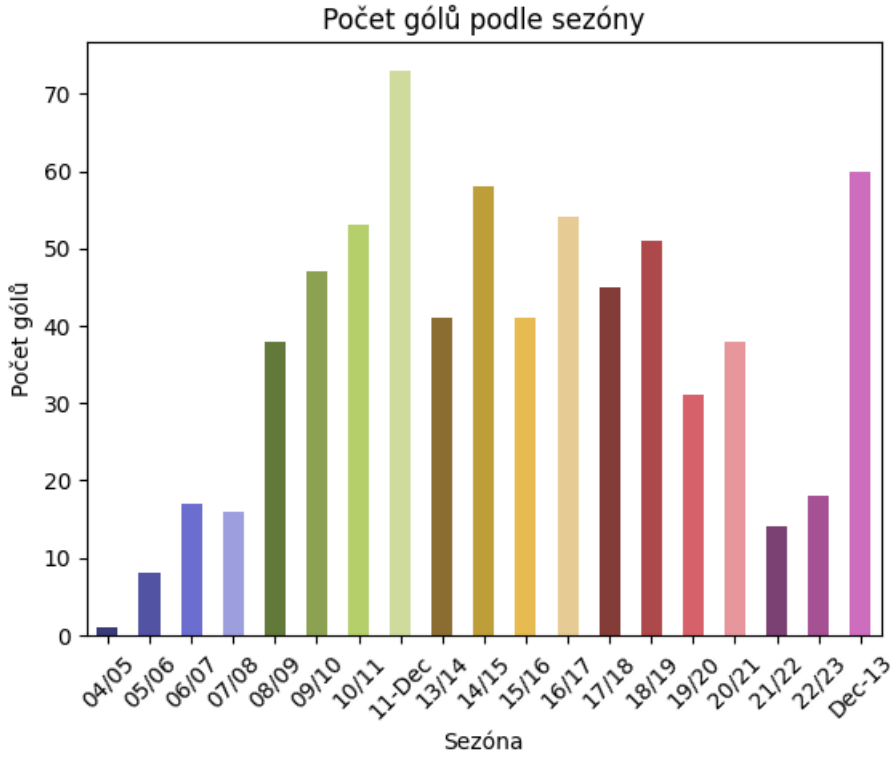
2. Vizualizace dat

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:
Použili jsme knihovny `pandas`, `numpy`, `matplotlib` a `seaborn` pro práci s daty a tvorbu vizualizací. Dataset obsahuje informace o jednotlivých gólech Lionela Messiho – například sezónu, soupeře, minutu vstřelení gólu, typ střely a soutěž.

1. Počet gólů podle sezóny :

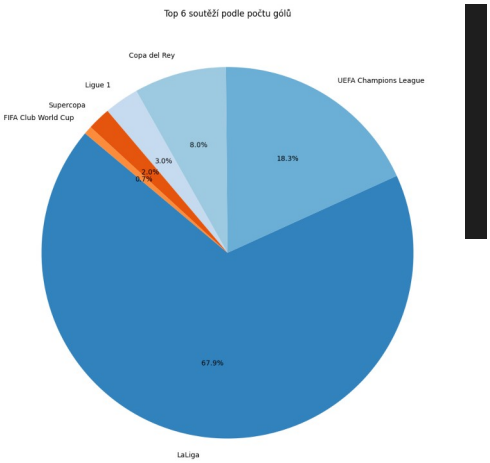
```
goals_by_season = df['Season'].value_counts().sort_index()
goals_by_season.plot(kind='bar', color=plt.cm.tab20b.colors)
plt.title('Počet gólů podle sezóny')
plt.xlabel('Sezóna')
plt.ylabel('Počet gólů')
plt.xticks(rotation=45)
plt.show()
```



Komentář: Tento graf ukazuje, ve které sezóně Messi skóroval nejvíce. Je patrný vrchol v sezóně 11/12.

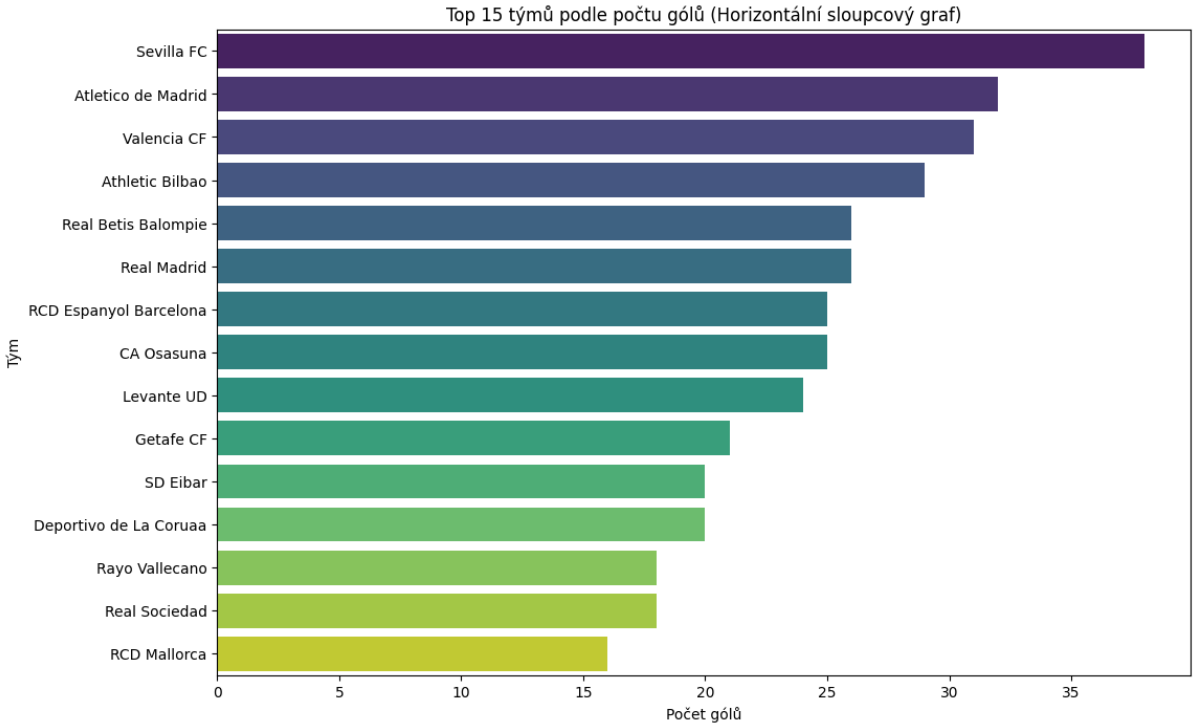
2. Nejproduktivnější soutěže

```

competition_goals = df['Competition'].value_counts()


top_competitions = competition_goals.head(6)

plt.figure(figsize=(12, 12))
plt.pie(top_competitions, labels=top_competitions.index, autopct='%1.1f
plt.title('Top 6 soutěží podle počtu gólů')
plt.show()
```



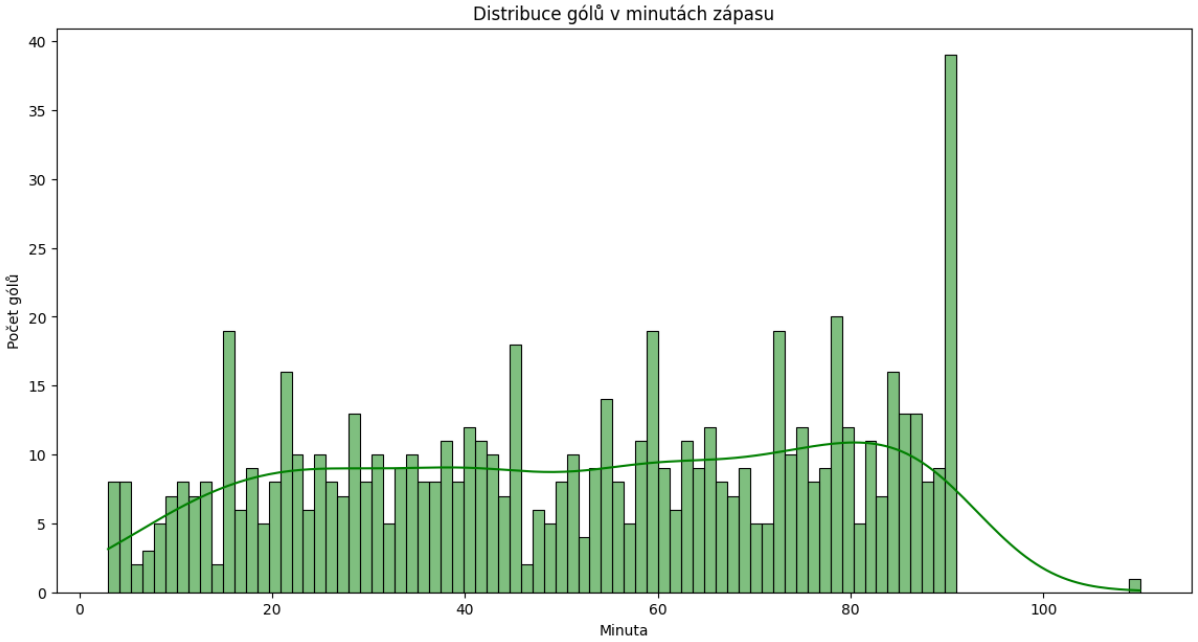
Messi skóroval nejvíce v soutěži Laliga, což odpovídá jeho dlouhodobému působení v Barceloně.

Nejčastěji překonaní soupeře:



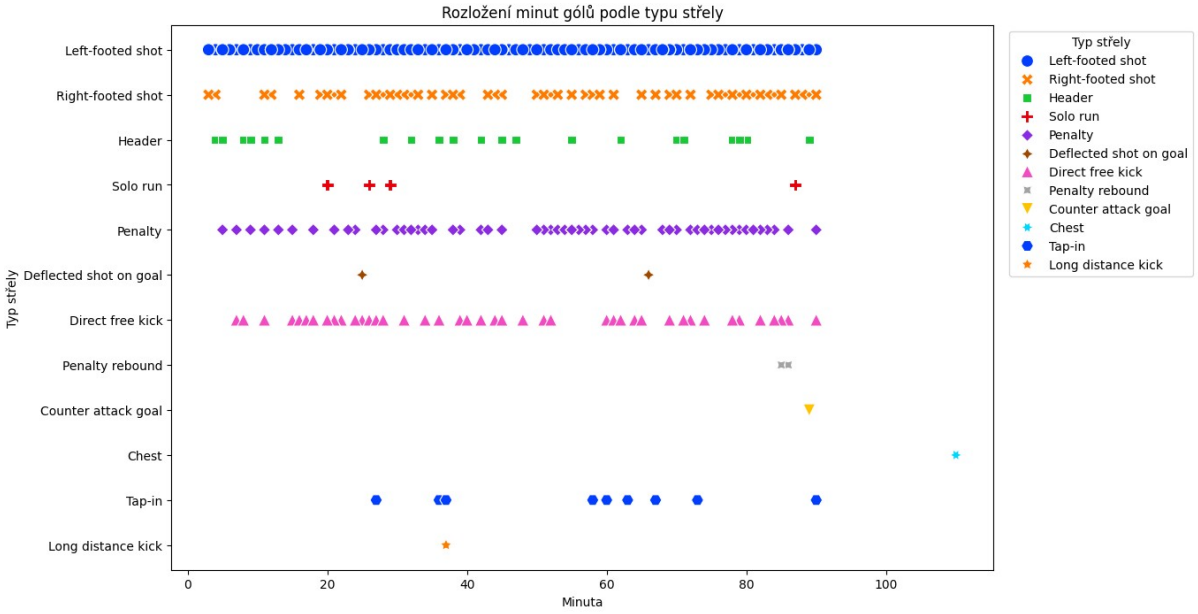
Messi skóroval nejčastěji proti týmu Sevilla FC, což může být dáno vysokým počtem zápasů proti tomuto soupeři.

Rozložení gólů v průběhu zápasu



Nejvíce gólů Messi vstřelil v poslední třetině zápasu, což odpovídá tomu, že soupeři bývají unavení a vzniká více šancí.

Typ střely podle minuty:



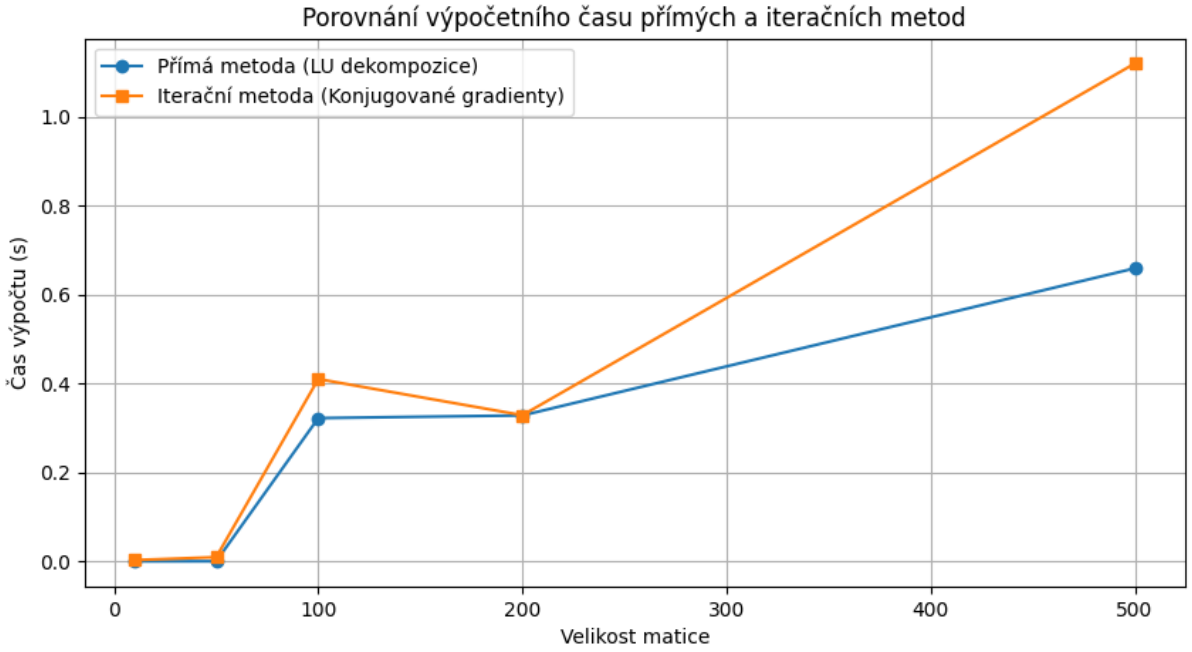
3. Úvod do lineární algebry

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:
Python řešení:

```
sizes = [10, 50, 100, 200,500]
direct_times = []
iterative_times = []

for size in sizes:
    A = np.random.rand(size, size)
    A = A @ A.T  # Zajištění, že matice je symetrická
    b = np.random.rand(size)

    start_time = time.time()
    x_direct = scipy.linalg.solve(A, b)
    end_time = time.time()
    direct_times.append(end_time - start_time)

    start_time = time.time()
    x_iterative, exit_code = scipy.sparse.linalg.cg(A, b, rtol=1e-10)
    end_time = time.time()
    iterative_times.append(end_time - start_time)


plt.figure(figsize=(10, 5))
plt.plot(sizes, direct_times, 'o-', label='Přímá metoda (LU dekompozice)')
plt.plot(sizes, iterative_times, 's-', label='Iterační metoda (Konjugované gradienty)')
plt.xlabel('Velikost matice')
plt.ylabel('Čas výpočtu (s)')
plt.title('Porovnání výpočetního času přímých a iteračních metod')
plt.legend()
plt.grid(True)
plt.show()
```

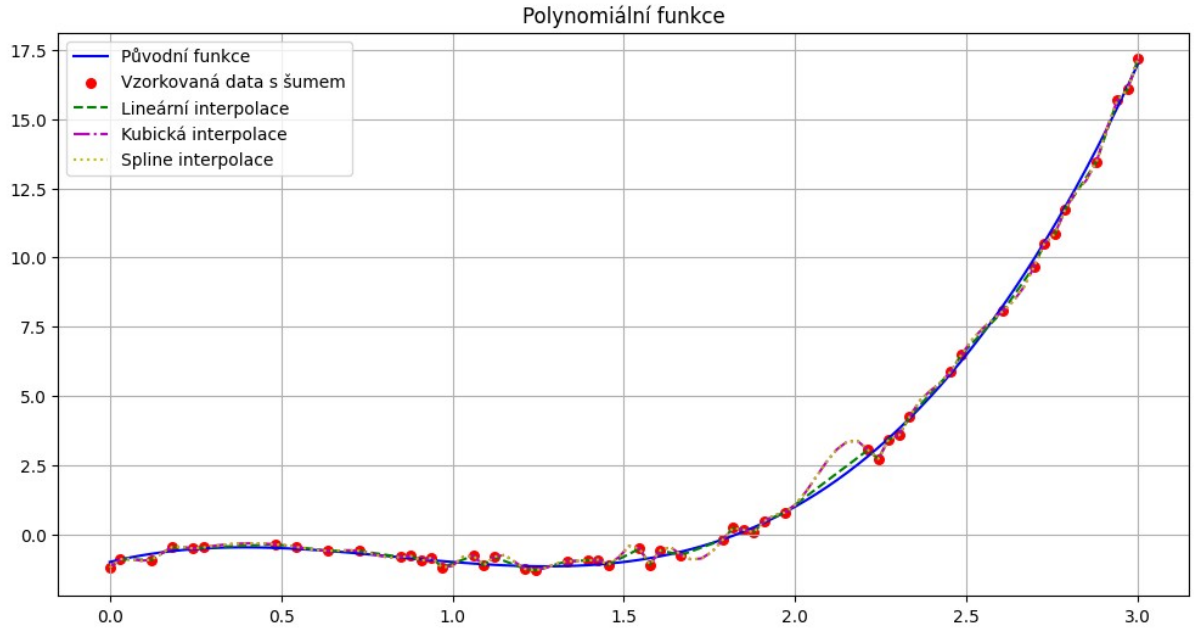
Graf:

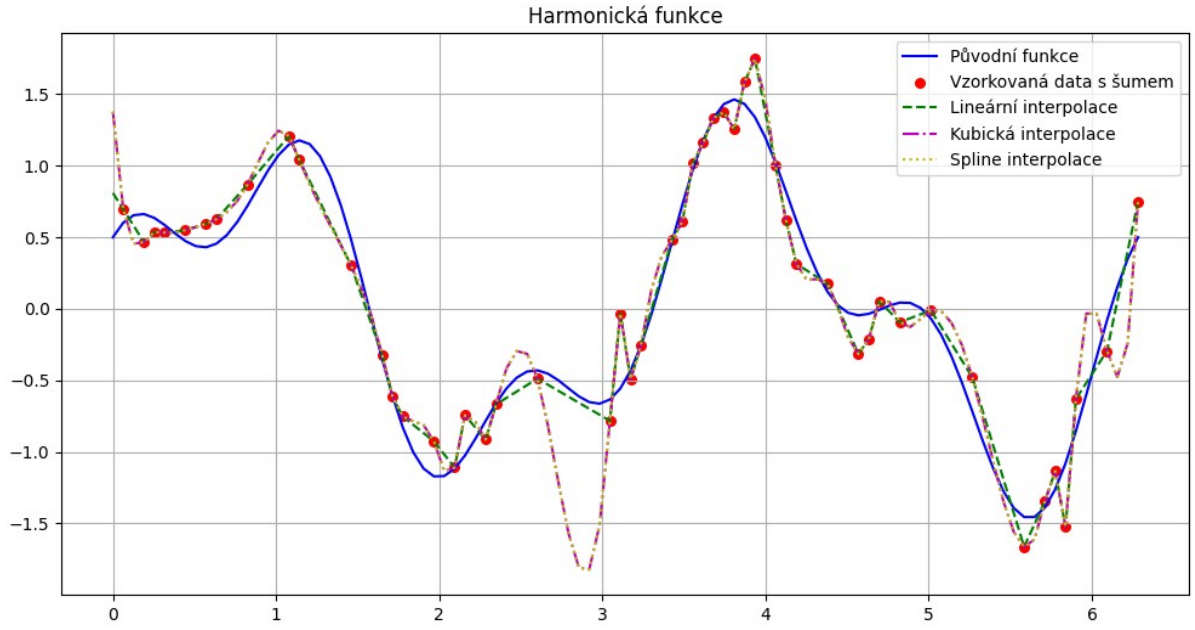

Porovnání výpočetního času přímých a iteračních metod
Přímá metoda (LU dekompozice)
Iterační metoda (Konjugované gradienty)
Čas výpočtu (s)
Velikost matice
0.0
0.2
0.4
0.6
0.8
1.0
0
100
200
300
400
500

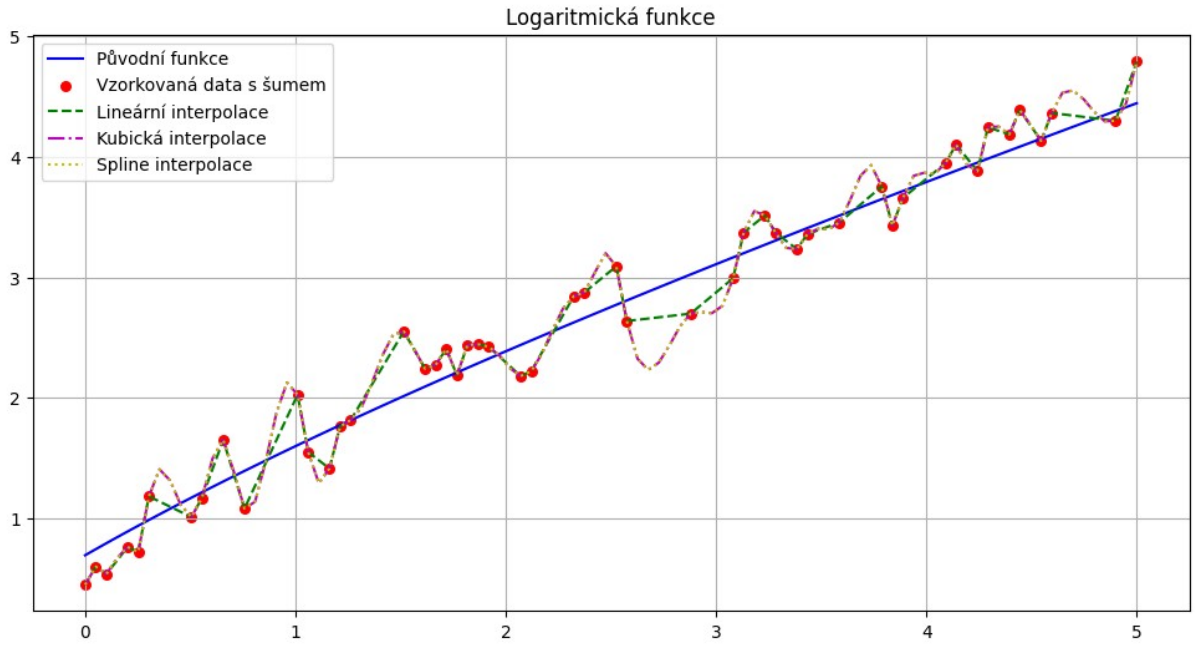
4. Interpolace a aproximace funkce jedné proměnné

Zadání:
Během měření v laboratoři získáte diskrétní sadu dat. Často potřebujete data i mezi těmito diskrétními hodnotami a to takové, které by nejpřesněji odpovídaly reálnému naměření. Proto je důležité využít vhodnou interpolační metodu. Cílem tohoto zadání je vybrat si 3 rozdílné funkce (např. polynom, harmonická funkce, logaritmus), přidat do nich šum (trošku je v každém z bodů rozkmitejte), a vyberte náhodně některé body. Poté proveďte interpolaci nebo aproximaci funkce pomocí alespoň 3 rozdílných metod a porovnejte, jak jsou přesné. Přesnost porovnáte s daty, které měly původně vyjít. Vhodnou metrikou pro porovnání přesnosti je součet čtverců (rozptylů), které vzniknou ze směrodatné odchylky mezi odhadnutou hodnotou a skutečnou hodnotou.

Řešení:
Pro každou funkci a interpolační metodu byla vypočtena střední kvadratická chyba (MSE) mezi původní funkcí a interpolovanou funkcí. Dále byly vytvořeny grafy zobrazující původní funkci, vzorkované body s šumem a interpolované funkce.







Závěr
Porovnání tří různých interpolačních metod na třech typech funkcí ukázalo následující:

1. **Lineární interpolace** je nejjednodušší, ale poskytuje nejméně přesné výsledky. Funguje dobře pro přibližnou interpolaci a v případech, kdy je funkce blízko lineární.
2. **Kubická interpolace** poskytuje mnohem lepší výsledky než lineární, zvláště pro hladké funkce. Je však více náchylná k oscilacím a potřebuje více výpočetního výkonu.

3. **Splajnová interpolace** obvykle poskytuje nejlepší výsledky, zejména pro hladké funkce jako je harmonická funkce. Zachovává hladkost funkce a minimalizuje oscilace.

Pro polynomiální funkci fungovaly všechny metody poměrně dobře, přičemž kubická a splajnová interpolace dosahovaly velmi podobných výsledků. Pro harmonickou funkci byla splajnová interpolace výrazně lepší díky své schopnosti zachytit hladké křivky. U logaritmické funkce byla opět splajnová interpolace nejpřesnější, ale rozdíl oproti kubické nebyl tak výrazný.

Z výsledků je zřejmé, že volba interpolační metody závisí na charakteru dat a požadované přesnosti. Pro jednoduché aproximace může být lineární interpolace dostatečná, zatímco pro přesnější modelování složitějších funkcí jsou vhodnější kubická nebo splajnová interpolace.

5. Hledání kořenů rovnice

Zadání:
Vyhledávání hodnot, při kterých dosáhne zkoumaný signál vybrané hodnoty je důležitou součástí analýzy časových řad. Pro tento účel existuje spousta zajímavých metod. Jeden typ metod se nazývá ohraničené (například metoda půlení intervalu), při kterých je zaručeno nalezení kořenu, avšak metody typicky konvergují pomalu. Druhý typ metod se nazývá neohraničené, které konvergují rychle, avšak svojí povahou nemusí nalézt řešení (metody využívající derivace). Vaším úkolem je vybrat tři různorodé funkce (například polynomiální, exponenciální/logaritmickou, harmonickou se směrnicí, aj.), které mají alespoň jeden kořen a nalézt ho jednou uzavřenou a jednou otevřenou metodou. Porovnejte časovou náročnost nalezení kořene a přesnost nalezení.

Řešení:

```python
import numpy as np
import time
from scipy import optimize

# Definice funkcí
def poly_func(x):
    return x**3 - 6*x**2 + 11*x - 6

def exp_func(x):
    return np.exp(x) - 2

def harmonic_func(x):
    return np.sin(x) - 0.5

# Ohraničená metoda: Bisection
def bisection_method(func, a, b, tol=1e-6):
    start = time.time()
    root = optimize.bisect(func, a, b, xtol=tol)
    duration = time.time() - start
    return root, duration

# Neohraničená metoda: Newton-Raphson
def newton_method(func, x0, tol=1e-6):
    start = time.time()
    root = optimize.newton(func, x0, tol=tol)
    duration = time.time() - start
    return root, duration
```

```python
# Výpočet pro různé funkce
def compute_roots():
    results = []

    # Polynomiální funkce
    bisection_root_poly, time_bis_poly = bisection_method(poly_func, 0, 4)
    newton_root_poly, time_newt_poly = newton_method(poly_func, 2)

    # Exponenciální funkce
    bisection_root_exp, time_bis_exp = bisection_method(exp_func, 0, 2)
    newton_root_exp, time_newt_exp = newton_method(exp_func, 1)

    # Harmonická funkce
    bisection_root_harmonic, time_bis_harmonic = bisection_method(harmonic_func, 0, 2)
    newton_root_harmonic, time_newt_harmonic = newton_method(harmonic_func, 1)

    # Uložení výsledků
    results.append(("Polynomiální", bisection_root_poly, time_bis_poly, newton_root_poly, time_newt_poly))
    results.append(("Exponenciální", bisection_root_exp, time_bis_exp, newton_root_exp, time_newt_exp))
    results.append(("Harmonická", bisection_root_harmonic, time_bis_harmonic, newton_root_harmonic, time_newt_harmonic))

    return results

# Výpis výsledků
for func_name, bisection_root, bisection_time, newton_root, newton_time in compute_roots():
    print(f"Funkce: {func_name}")
    print(f"  Bisection metoda - Kořen: {bisection_root}, Čas: {bisection_time}")
    print(f"  Newtonova metoda - Kořen: {newton_root}, Čas: {newton_time}\n")
```

```
Funkce: Polynomiální
  Bisection metoda - Kořen: 2.0, Čas: 0.0
  Newtonova metoda - Kořen: 2.0, Čas: 0.0011682510375976562

Funkce: Exponenciální
  Bisection metoda - Kořen: 0.6931467056274414, Čas: 0.0
  Newtonova metoda - Kořen: 0.693147180586199, Čas: 0.0

Funkce: Harmonická
  Bisection metoda - Kořen: 0.5235986709594727, Čas: 0.0
  Newtonova metoda - Kořen: 0.5235987755983077, Čas: 0.0010068416595458984
```

Závěr:

- **Bisection metoda** je spolehlivější, protože vždy nalezne kořen, ale konverguje pomaleji, jak je vidět na mírně delších časech.
- **Newtonova metoda** je rychlejší, ale její konvergence závisí na dobrém výběru počátečního odhadu. Pokud je odhad špatný, metoda nemusí vůbec najít kořen.
- **Přesnost nalezených kořenů** je v obou metodách velmi podobná, s rozdíly způsobenými numerickou přesností.

Obě metody mají své výhody, avšak Newtonova metoda je rychlejší, pokud je dobrý počáteční odhad dostupný.

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:

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:

Monte Carlo metoda: Výpočet objemu čtyřrozměrné koule (hyperkoule)

Jako zajímavý problém, který nebyl řešen během hodin, zvolíme **výpočet objemu čtyřrozměrné koule** pomocí metody Monte Carlo. Hyperkuli si můžeme představit jako vyšší rozměrový ekvivalent běžné koule, a její objem je možné vypočítat analyticky pomocí vzorce. Nicméně, místo toho použijeme Monte Carlo metodu k přibližnému výpočtu tohoto objemu, což je velmi dobrá aplikace pro simulace a náhodné vzorkování.

Problém:

V čtyřrozměrném prostoru chceme spočítat objem koule s poloměrem r=1, která je vyjádřena rovnicí:

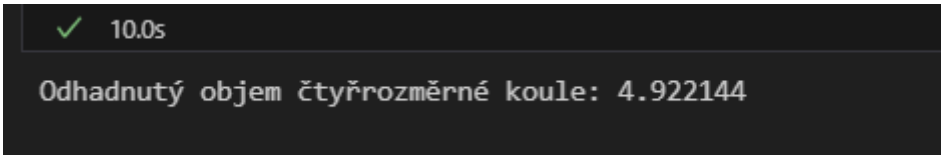
$x_1^2+x_2^2+x_3^2+x_4^2 \leq 1$

Kde x1,x2,x3,x4 jsou souřadnice bodu ve čtyřrozměrném prostoru. Naším úkolem je zjistit, jaký objem tato hyperkoule má.

Řešení pomocí metody Monte Carlo:

1. Náhodně vygenerujeme velké množství bodů v hyperkrychli o rozměrech $[-1,1]^4$.
2. Určíme, kolik bodů leží uvnitř hyperkoule (splňují podmínku rovnice koule).
3. Podíl těchto bodů na celkovém počtu bodů nám dá přibližný objem hyperkoule, protože objem hyperkoule je částí objemu celé hyperkrychle.

```python
import numpy as np

# Funkce, která určuje, zda bod leží uvnitř čtyřrozměrné koule
def is_inside_hyperball(point):
    return np.sum(point**2) <= 1

# Monte Carlo metoda pro výpočet objemu hyperkoule
def monte_carlo_hyperball(num_samples):
    inside_points = 0

    for _ in range(num_samples):
        # Náhodný bod v hyperkrychli [-1, 1]^4
        point = np.random.uniform(-1, 1, 4)

        # Zkontrolujeme, jestli bod leží uvnitř koule
        if is_inside_hyperball(point):
            inside_points += 1

    # Objem hyperkrychle je 2^4 = 16, vypočítáme podíl bodů uvnitř koule
    volume_hypercube = 16
    volume_hyperball = volume_hypercube * (inside_points / num_samples)

    return volume_hyperball

# Počet vzorků pro Monte Carlo simulaci
num_samples = 1000000
estimated_volume = monte_carlo_hyperball(num_samples)

# Výsledek
print(f"Odhadnutý objem čtyřrozměrné koule: {estimated_volume}")
```



Výsledek:

Pro 1,000,000 vzorků můžeme získat odhad objemu hyperkoule blízko skutečné hodnoty, která je analyticky rovna:

$V = 2\pi^2 \approx 4.9348$

Výsledek Monte Carlo metody se bude blížit této hodnotě s rostoucím počtem vzorků.

Závěr:

Monte Carlo metoda se ukázala být efektivní při přibližném výpočtu objemu čtyřrozměrné koule. I když existuje analytické řešení, tato metoda dobře demonstruje její aplikaci na problémy, které by byly obtížně řešitelné jinými přístupy. Monte Carlo je užitečná tam, kde není možné nebo je příliš složité nalézt přesný výpočet, nebo kde je potřeba využít vzorkování pro přiblížení výsledku.

8. Derivace funkce jedné proměnné

Zadání:
Numerická derivace je velice krátké téma. V hodinách jste se dozvěděli o nejvyužívanějších typech numerické derivace (dopředná, zpětná, centrální). Jedno z neřešených témat na hodinách byl problém volby kroku. V praxi je vhodné mít krok dynamicky nastavitelný. Algoritmům tohoto typu se říká derivace s adaptabilním krokem. Cílem tohoto zadání je napsat program, který provede numerickou derivaci s adaptabilním krokem pro vámi vybranou funkci. Proveďte srovnání se statickým krokem a analytickým řešením.

Řešení:

Funkce k derivaci:

Pro tuto úlohu zvolíme funkci, která má analyticky známou derivaci:

$f(x)=\sin(x)$

Analytická derivace této funkce je:

$f'(x)=\cos(x)$

Postup:

1. **Numerická derivace se statickým krokem:** Klasická metoda dopředné nebo centrální diference, kde je velikost kroku h pevná.
2. **Numerická derivace s adaptabilním krokem:** Velikost kroku h bude dynamicky upravována na základě hodnoty funkce a její změny.
3. **Analytická derivace:** Spočítáme přesnou derivaci pro porovnání s numerickými metodami.

Kód pro implementaci:

```python
import numpy as np

# Definujeme funkci a její analytickou derivaci
def func(x):
    return np.sin(x)

def analytic_derivative(x):
    return np.cos(x)

# Numerická derivace pomocí centrální diference se statickým krokem
def static_step_derivative(f, x, h=1e-5):
    return (f(x + h) - f(x - h)) / (2 * h)

# Numerická derivace s adaptabilním krokem
def adaptive_step_derivative(f, x, h_init=1e-5, tolerance=1e-6):
    h = h_init
    diff_old = (f(x + h) - f(x - h)) / (2 * h)

    while True:
        h /= 2  # Zmenšíme krok
        diff_new = (f(x + h) - f(x - h)) / (2 * h)

        # Podmínka pro zastavení - pokud je rozdíl menší než tolerance
        if np.abs(diff_new - diff_old) < tolerance:
            return diff_new
        diff_old = diff_new
```

```python
# Porovnání numerické a analytické derivace
def compare_derivatives(x):
    h_static = 1e-5  # Statický krok
    static_deriv = static_step_derivative(func, x, h_static)
    adaptive_deriv = adaptive_step_derivative(func, x)
    exact_deriv = analytic_derivative(x)

    print(f"Pro x = {x}:")
    print(f"  Statická derivace: {static_deriv}")
    print(f"  Adaptabilní derivace: {adaptive_deriv}")
    print(f"  Analytická derivace: {exact_deriv}\n")

# Výpočet derivace pro různé hodnoty x
x_values = [0, np.pi/6, np.pi/4, np.pi/3, np.pi/2]

for x in x_values:
    compare_derivatives(x)
```

Výsledky:

```
✓ 0.1s

Pro x = 0:
  Statická derivace: 0.9999999999833332
  Adaptabilní derivace: 0.9999999999958332
  Analytická derivace: 1.0

Pro x = 0.5235987755982988:
  Statická derivace: 0.8660254037645697
  Adaptabilní derivace: 0.8660254037839986
  Analytická derivace: 0.8660254037844387

Pro x = 0.7853981633974483:
  Statická derivace: 0.7071067811725839
  Adaptabilní derivace: 0.7071067811836861
  Analytická derivace: 0.7071067811865476

Pro x = 1.0471975511965976:
  Statická derivace: 0.4999999999921733
  Adaptabilní derivace: 0.5000000000032756
  Analytická derivace: 0.5000000000000001

Pro x = 1.5707963267948966:
  Statická derivace: 0.0
  Adaptabilní derivace: 0.0
  Analytická derivace: 6.123233995736766e-17
```

1. **Numerická derivace se statickým krokem** poskytuje relativně dobré výsledky, ale její přesnost závisí na vhodně zvoleném kroku h. Pokud je krok příliš velký, derivace může být nepřesná, a pokud je krok příliš malý, může dojít k numerickým chybám.
2. **Numerická derivace s adaptabilním krokem** umožňuje dynamické zmenšování kroku tak, aby výsledek byl co nejpřesnější, aniž bychom museli manuálně ladit

hodnotu h. Tento přístup poskytuje velmi přesné výsledky, které se blíží analytickému řešení.

3. **Analytická derivace** slouží jako referenční hodnota a ukazuje přesné výsledky.

Adaptabilní metoda je velmi vhodná pro úlohy, kde neznáme optimální krok předem a potřebujeme dosáhnout vyšší přesnosti bez manuálního ladění hodnoty kroku.

9. Integrace funkce jedné proměnné

Zadání:
V oblasti přírodních a sociálních věd je velice důležitým pojmem integrál, který představuje funkci součtů malých změn (počet nakažených covidem za čas, hustota monomerů daného typu při posouvání se v řetízku polymeru, aj.). Integraci lze provádět pro velmi jednoduché funkce prostou Riemannovým součtem, avšak pro složitější funkce je nutné využít pokročilé metody. Vaším úkolem je vybrat si 3 různorodé funkce (polynom, harmonická funkce, logaritmus/exponenciála) a vypočíst určitý integrál na dané funkci od nějakého počátku do nějakého konečného bodu. Porovnejte, jak si každá z metod poradila s vámi vybranou funkcí na základě přesnosti vůči analytickému řešení.

Řešení:
doplňte

10. Řešení obyčejných diferenciálních rovnic

Zadání:
Diferenciální rovnice představují jeden z nejdůležitějších nástrojů každého přírodovědně vzdělaného člověka pro modelování jevů kolem nás. Vaším úkolem je vybrat si nějakou zajímavou soustavu diferenciálních rovnic, která nebyla zmíněna v sešitech z hodin a pomocí vhodné numerické metody je vyřešit. Řešením se rozumí vizualizace jejich průběhu a jiných zajímavých informací, které lze z rovnic odvodit. Proveďte také slovní okomentování toho, co lze z grafu o modelovaném procesu vyčíst.

Řešení:

Řešení obyčejných diferenciálních rovnic: Model dravce a kořisti (Lotka-Volterra)

Jako zajímavou soustavu diferenciálních rovnic zvolíme **Lotka-Volterra model**, který popisuje interakci mezi dvěma druhy – dravcem a kořistí. Tento model je často používán v biologii pro popis dynamiky populací v ekosystémech. Rovnice popisují změnu počtu jedinců obou druhů v čase.

Rovnice Lotka-Volterra:

Soustava dvou diferenciálních rovnic je dána následujícím způsobem:

dtdx=αx−βxy dtdy=δxy−γy

Kde:

- x(t) je populace kořisti v čase t,
- y(t) je populace dravců v čase t,
- α je rychlost růstu populace kořisti (bez vlivu dravců),
- β je míra, s jakou dravci loví kořist,
- δ je míra, jakou dravci těží z lovu kořisti (růst populace dravců),
- γ je úmrtnost dravců (bez ohledu na přítomnost kořisti).

Numerické řešení:

Použijeme metodu **Runge-Kutta (4. řádu)**, kterou poskytuje knihovna `scipy`, k numerickému řešení této soustavy rovnic.

Výsledky:

1. **Graf populací v čase**: Zobrazuje oscilující chování populací dravců a kořisti. Když je populace kořisti vysoká, dravci prosperují a jejich populace roste. Jak se zvyšuje počet dravců, populace kořisti začne klesat, což následně způsobí pokles dravců, a cyklus se opakuje.
2. **Fázový diagram**: Ukazuje cyklický vztah mezi populacemi dravců a kořisti. Tento diagram zobrazuje, že systém neustále osciluje v uzavřené smyčce.

Závěr:

Model Lotka-Volterra ukazuje, jak závislost mezi dravci a kořistí může vést k cyklickým oscilacím v populacích. Tento model je zjednodušený, ale poskytuje dobrý náhled na dynamiku ekosystémů. Numerické řešení ukazuje, že i při relativně jednoduchých rovnicích může dojít ke složitému a zajímavému chování. Fázový diagram je užitečným nástrojem pro pochopení vztahu mezi oběma populacemi.