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

Řešení:

Jako první jsem si naimportoval potřebné knihovny a nadefinoval seznamy pro výstupy daných metod pro následnou vizualizaci.

Následně jsem si nadefinoval zkoumané funkce. Jedná se o polynomickou, harmonickou a logaritmickou funkci. Před definováním jsem si z knihovny SymPy. SymPy je mocná knihovna pro symbolickou matematiku v jazyce Python. Když je potřeba definovat proměnnou 'x' jako symbol pomocí funkce sp.symbols('x') můžete provádět různé algebraické manipulace a symbolicky řešit rovnice pomocí SymPy.

```
x = sp.symbols('x')

def polynomicka(x):
    return x**2

def harmonicka(x):
    return 2*sp.sin(x**2)

def logaritmicka(x):
    return sp.log(x**4)
```

Po definování zkoumaných funkcí, přišla řada na zvolení a implementování metod, které nám budou sloužit pro řešení daných funkcí.

Jako první jsem definoval **lichoběžníkovou metodu**:

```
#LICHOBEZNIKOVA METODA

def lichobeznik_method(funkce, a, b, n):
    start_time = datetime.datetime.now()
    integral = 0  # hodnota integrálu
    krok = (b - a) / n #definice velikosti kroku
    x_i = a #přetypování začátku

    for _ in range(n): #cyklus v počtu iteraci
        aktualni_pozice = x_i + krok  # aktualizace pozice pro každý interval

        iterace = (funkce.subs(x, x_i) + funkce.subs(x, aktualni_pozice)) * krok / 2  # výpočet plochy lichoběžníku
        #  pro nahrazení symbolu x za konkrétní hodnoty
        # 1. proměnná = znak, který nahradím hodnotou druhé proměnné : (značka,hodnota)

        integral += iterace
        x_i = aktualni_pozice  # posun na další interval

    timer_licho = (datetime.datetime.now() - start_time)
    timer_licho_ms = timer_licho.total_seconds() * 1000
    print(f"hodnota lichobeznik: {integral} čas: {timer_licho_ms} ms")

    lichobeznik_output.append((integral, timer_licho_ms))
    return integral, timer_licho_ms
```

Jejím parametrem je daná funkce, **a** - začátek intervalu, **b** - konec intervalu, **n** - počet intervalů (iterací). Následně je zde definována proměnná **integral**, do kterého se každá iterace zaznamenává a sčítá (slouží pro výsledek). Proměnná **krok** nám slouží k rozdělení intervalu od a do b na n stejných kousků, které následně slouží pro výpočet daného lichoběžníku. Proměnná **x_i** slouží pouze k přetypování začátku intervalu **a**.

Celá metoda je pomocí funkce **datetime** časové měřena.

Cyklus iterací v range(n)

Při vstupu se přepočte primární pozice na daném lichoběžníku. Následně proběhne iterace, ve které pomocí funkce **.subs()** dochází k substituci symbolu x a dané proměnné, kterou do funkce chceme vložit. Po celém výpočtu daného lichoběžníku dojde řada k zaznamenání výsledné hodnoty do proměnné **integral**. A úplně na konci dojde k posunu počátku iterace.

Po výstupu z cyklu dojde k výpočtu časové náročnosti a následně její převod do ms. **Z celé metody vracím tuple(integral, timer_licho_ms).**

Následně jsem definoval **obdélníkovou metodu:**

```
#OBDELNIKOVA METODA

def obdelnikova_method(funkce, a, b, n):
    start_time = datetime.datetime.now()
    integral = 0  # hodnota integrálu
    krok = (b - a) / n #definice velikosti kroku
    x_i = a #přetypování začátku

    for _ in range(n):
        posun = x_i + krok
        iterace = funkce.subs(x,posun) * (posun-x_i)
        integral += iterace
        x_i = posun

    timer_obdelnik = (datetime.datetime.now() - start_time)
    timer_obdelnik_ms = timer_obdelnik.total_seconds() * 1000
    print(f"hodnota lichobeznik: {integral} čas: {timer_obdelnik_ms} ms")

    obdelnik_output.append((integral, timer_obdelnik_ms))
    return integral, timer_obdelnik_ms
```

Ta je v zásadě dosti podobná jako lichoběžníková, akorát je zde rozdíl ve výpočtu daného **obdélníku v iteraci.**

Následně jsem definoval **NumPy Trapz metodu:**

Tato build-in metoda používá k numerickému výpočtu integrálu pomocí metody lichoběžníků (trapezoidální metoda).

Zásada metody **numpy.trapz** je založena na aproximaci plochy pod křivkou pomocí trapezoidů. Místo přímek, které jsou použity v metodě lichoběžníků, se zde používají trapezy, které lépe aproximují plochu pod křivkou.

```
#NUMPY TRAPZ METODA  BUILT-IN

def numpy_trapz(funkce,a,b,n):
    start_time = datetime.datetime.now()
    dx = (b - a) / n #výpočet kroku
    x_vec = np.arange(a, b + dx, dx)


    f_func = sp.lambdify(x, funkce)
    #převádí symbolickou funkci na ekvivalentní funkci, která je vyhodnocovatelná pomocí NumPy

    trapz_method = np.trapz(f_func(x_vec), x_vec)

    timer_numpy_trapz = (datetime.datetime.now() - start_time)
    timer_numpy_trapz_ms = timer_numpy_trapz.total_seconds() * 1000

    #timer_numpy_trapz_ms = round(timer_numpy_trapz_ms,3)

    print(f"hodnota numpy_trapz: {trapz_method} čas: {timer_numpy_trapz_ms} ms")

    numpy_trapz_output.append((trapz_method, timer_numpy_trapz_ms))
    return trapz_method, timer_numpy_trapz_ms
```

x_vec je vektor, který obsahuje x-ové hodnoty bodů na intervalu od a do b s krokem dx. Tento vektor je vytvořen pomocí funkce **np.arange**.

Následně definuji proměnou **f_func**, která obsahuje převedenou vstupní funkci pomocí SymPy funkce **lambdify()**, která převádí symbolickou funkci na ekvivalentní, která je následně použitelná pomocí knihovny NumPy.

Následně jsem definoval **analytickou metodu:**

Tato metoda je z knihovny SymPy. Daná funkce se nazývá **integrate()**. Tuto funkci jsem si zvolil jako analytickou, díky její přesnosti. Od ní jsem nakonec měřil výsledné odchylky všech metod.

```python
#ANALYTIC SOLUTION - PRO MNE

def analytic_solution(funkce,vec):
    start_time = datetime.datetime.now()
    analytic_sol = sp.integrate(funkce, vec)
    timer_analytic_sol = (datetime.datetime.now() - start_time)
    timer_analytic_sol_ms = timer_analytic_sol.total_seconds() * 1000

    #timer_analytic_sol_ms = round(timer_analytic_sol_ms,5)

    print(f"ANALYTIC: {float(analytic_sol)} čas: {timer_analytic_sol_ms} ms")

    analytic_output.append((float(analytic_sol), timer_analytic_sol_ms))
    return analytic_sol, timer_analytic_sol_ms
```

Tato metoda přijímá jako vstup předpis dané funkce a následný vektor ve tvaru **(proměnná, od, do).**

```python
#TĚLO PROGRAMU

#nazvy: polynomicka,harmonicka,logaritmicka
#metody: lichobeznik, obdelnik, numpy_trapz, analytic solution


#POLYNOMICKA
analytic_solution(polynomicka(x),(x,0,10))
lichobeznik_method(polynomicka(x), 0, 10, 1000)
obdelnikova_method(polynomicka(x), 0, 10, 1000)
numpy_trapz(polynomicka(x),0,10,1000)


#HARMONICKA
analytic_solution(harmonicka(x),(x,0,10))
lichobeznik_method(harmonicka(x), 0, 10, 1000)
obdelnikova_method(harmonicka(x), 0, 10, 1000)
numpy_trapz(harmonicka(x), 0, 10, 1000)

#LOGARITMICKA
analytic_solution(logaritmicka(x),(x,1,10))
lichobeznik_method(logaritmicka(x), 1, 10, 1000)
obdelnikova_method(logaritmicka(x), 1, 10, 1000)
numpy_trapz(logaritmicka(x), 1, 10, 1000)

print("\n")
```

```python
print("-----------------------------------FINAL OUTPUT-----------------------------------")
#-------------------------------------------------------------------
print("ANALYTICKE SOLUTION:", analytic_output)
print("LICHOBEZNIKOVA SOLUTION:", lichobeznik_output)
print("OBDELNIKOVA SOLUTION:", obdelnik_output)
print("NUMPY TRAPZ METODA:", numpy_trapz_output)


# VÝPOČET ODCHYLEK
odchylky_licho = []
odchylky_obdelnik = []
odchylky_numpy = []

print("\n")
```

Následně jsem začal analyzovat výsledky daných metod. Jako první jsem začal analyzovat nepřesnosti daných metod od analyticke metody **integrate()**.

```python
for analytic, lichobeznik in zip(analytic_output, lichobeznik_output):
    # funkce zip, slouzi k prochazeni ve for cyklu ve
    # vice seznamech naraz

    odchylka_licho = abs(analytic[0] - lichobeznik[0]) #abs pro nezapornou odchylku (jde nam o rozmer)
    odchylky_licho.append(float(odchylka_licho))

for analytic, obdelnik in zip(analytic_output, obdelnik_output):
    odchylka_obdelnik = abs(analytic[0] - obdelnik[0])
    odchylky_obdelnik.append(float(odchylka_obdelnik))

for analytic, numpy in zip(analytic_output, numpy_trapz_output):
    odchylka_numpy = abs(analytic[0] - numpy[0])
    odchylky_numpy.append(float(odchylka_numpy))

print("ODCHYLKY LICHOBĚŽNÍKOVÁ METODA:", odchylky_licho)
print("ODCHYLKY OBDELNÍKOVÁ METODA:", odchylky_obdelnik)
print("ODCHYLKY NUMPY TRAPZ METODA:", odchylky_numpy)
```

Odchylky jsem si vytáhl z výsledných tuple z každé metody. Pomocí **for cyklu a funkce zip jsem procházel více listů najednou**. Následně jsem **vždy z každého outputu odečet od analytické hodnoty danou metodu v absolutním čísle**.

Následně jsem výsledné odchylky zaznamenal do listu s odchylkami dle metody.

Následně jsem z tuple odebral i časové údaje. Postup byl dost podobný, akorát jsem bral hodnoty na prvním indexu.

```
#PŘEVOD ČASŮ DANÝCH METOD
for analytic, lichobeznik, obdelnik, numpy_meth \
        in zip(analytic_output, lichobeznik_output, obdelnik_output, numpy_trapz_output):

    analytic_cas.append(analytic[1])
    lichobeznik_cas.append(lichobeznik[1])
    obdelnik_cas.append(obdelnik[1])
    numpy_trapz_cas.append(numpy_meth[1])

print(numpy_trapz_cas)
print(obdelnik_cas)
print(lichobeznik_cas)
print(analytic_cas)
```

Ano, mohl jsem si hodnoty přiřazovat rovnou ve funkci, ale chtěl jsem mít v terminálu hezký výstup dvojce pro kontrolu. Proto jsem si vyselektoval chtěné údaje pomocí for cyklu a zipu až v dalších krocích.

Po selekci dat, nastal čas na jejich vizualizaci. Pro zobrazení časové náročnosti jsem se rozhodl pro použití sloupcových grafů. **Na x ose jsou zobrazeny dané funkce a na ose y jsou sloupce všech metod.**

```
#GRAF ČASY


x = np.arange(len(analytic_cas))

width = 0.2  # Šířka sloupce

plt.bar(x - width, lichobeznik_cas, width=width, label="Lichobeznikova metoda")
plt.bar(x, obdelnik_cas, width=width, label="Obdelnikova metoda", align='center')
plt.bar(x + width, analytic_cas, width=width, label="Analytic method")
plt.bar(x + 1.5 * width, numpy_trapz_cas, width=width, label= "Numpy Trapz method", align='edge')

# Nastavení popisků os a titulku
plt.xlabel("Funkce")
plt.ylabel("Čas (ms)")
plt.title("Časová náročnost metod")

# Nastavení značek na ose x
plt.xticks(x, x + 1)
plt.xticks(x, ['Polynomicka', 'Harmonicka', 'Logaritmicka'])

# Zobrazení legendy
plt.legend()

# Zobrazení mřížky
plt.grid()


# Zobrazení grafu
plt.show()
```

Při porovnání daných metod jsem zjistil, že metoda NumPy Trapz po prvním běhu byla až neuvěřitelně rychlá. Důvod by měl být následující.

Při prvním volání metody **numpy.trapz** dochází k inicializaci a přípravě interních struktur a optimalizací. To může zahrnovat například překlad a optimalizaci před kompilovaného kódu nebo inicializaci interních paměťových struktur.

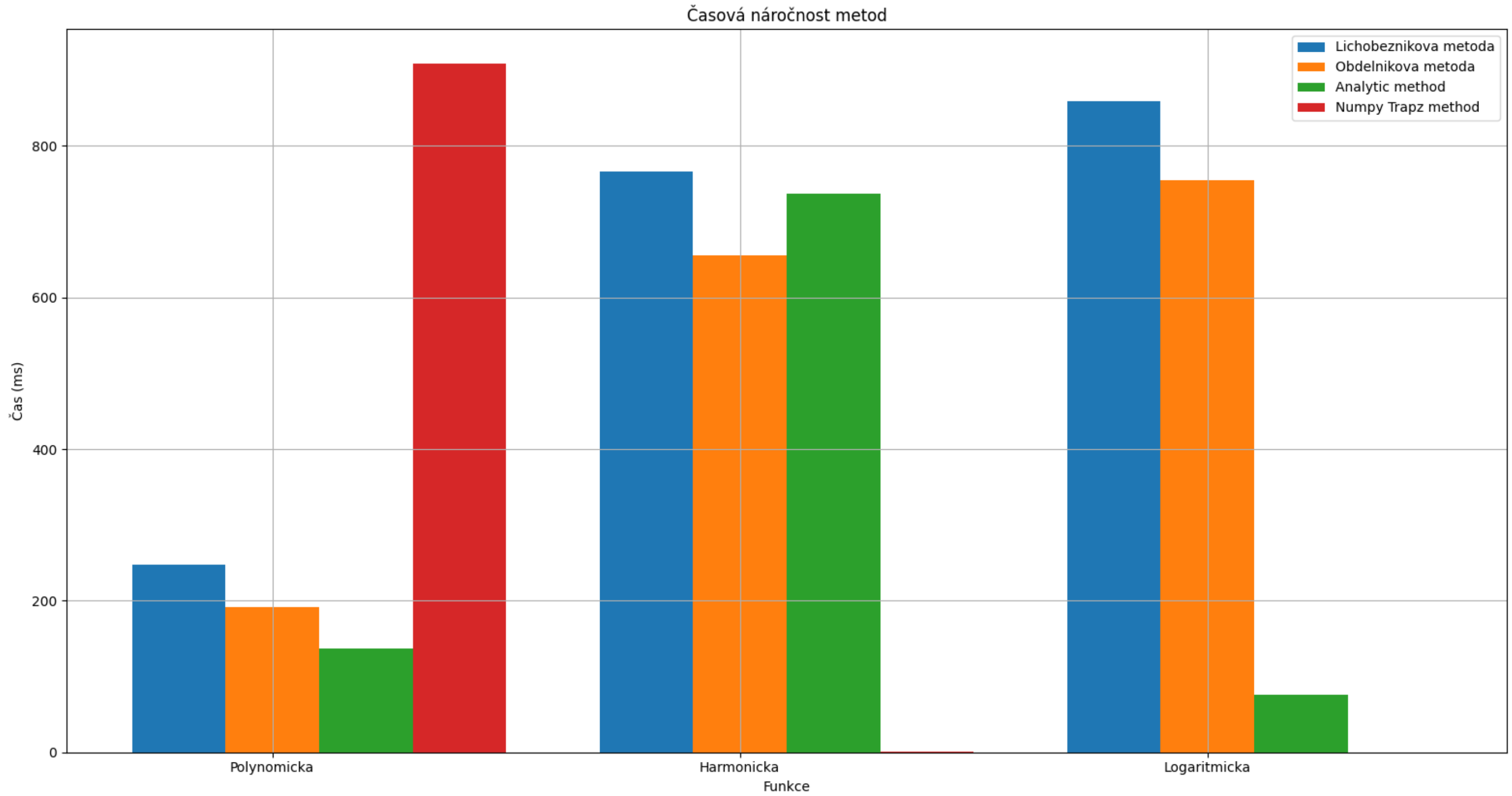
Po prvním volání jsou tyto struktury již připraveny a nemusí se opakovaně inicializovat, což může výrazně snížit časovou náročnost dalších volání.

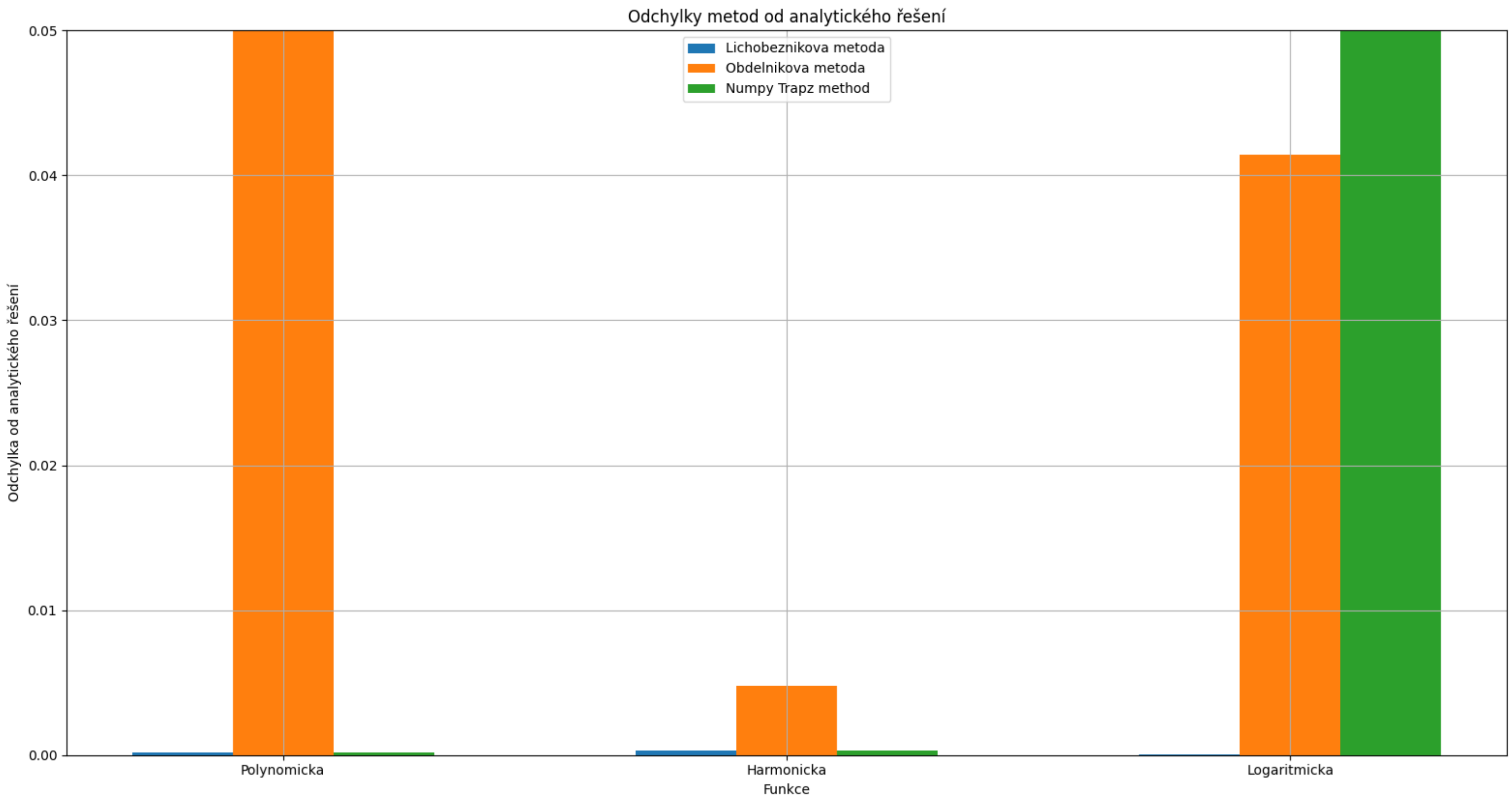
Pro vizualizaci odchylek jsem také využil sloupcové grafy.

```
#GRAF - ODCHYLKY

#každou zvlášť, nulová čára (sloupcově)

# X-ová osa (pořadí výpočtu)
x_axis = range(1, len(odchylky_licho) + 1)

plt.ylim(0, max(max(odchylky_licho), max(odchylky_obdelnik), max(odchylky_numpy)) * 0.1)
# Nastavte vhodný násobek maxima


width = 0.2  # Šířka sloupce

plt.bar(x - width, odchylky_licho, width=width, label="Lichobeznikova metoda")
plt.bar(x, odchylky_obdelnik, width=width, label="Obdelnikova metoda", align='center')
plt.bar(x + width, odchylky_numpy, width=width, label="Numpy Trapz method")

plt.xticks(x, ['Polynomicka', 'Harmonicka', 'Logaritmicka'])

# Nastavení popisků os a titulku
plt.xlabel("Funkce")
plt.ylabel("Odchylka od analytického řešení")
plt.title("Odchylky metod od analytického řešení")

plt.grid()

# Legenda
plt.legend()

# Zobrazení grafu
plt.show()
```

U obdélníkové metody byla odchylka tak vysoká, že se mi nepodařilo zařídit lepší zaznamenání na grafu.

Předpokládám, že **odchylky_licho**, **odchylky_obdelnik** a **odchylky_numpy** jsou seznamy obsahující hodnoty odchylek. Cílem tohoto kódu je nastavit horní hranici y-ové osy tak, aby zahrnovala maximální hodnoty odchylek v těchto seznamech. Konkrétně se využívá **max(max(odchylky_licho), max(odchylky_obdelnik), max(odchylky_numpy))**, abychom získali nejvyšší hodnotu odchylky mezi všemi třemi seznamy, a poté se tato hodnota násobí faktorem 0.1.


Časová náročnost metod
Lichobeznikova metoda
Obdelnikova metoda
Analytic method
Numpy Trapz method
Čas (ms)
0
200
400
600
800
Polynomicka
Harmonicka
Logaritmicka
Funkce


Odchylky metod od analytického řešení
Lichobeznikova metoda
Obdelnikova metoda
Numpy Trapz method
Odchylka od analytického řešení
0.00
0.01
0.02
0.03
0.04
0.05
Polynomicka
Harmonicka
Logaritmicka
Funkce

ODKAZ NA GITHUB:

https://github.com/Marty808s/MSW_9_Integrace