# 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:

Úkolem bylo porovnat hodnoty dynamického kroku se statickým krokem a analytickým řešením pro vybranou funkci. Jako funkci jsem si vybral logaritmickou funkci a porovnával jsem odchylku derivace od analytického řešení, které jsem si zvolil jako referenční. Dále jsem porovnával časovou náročnost, protože v průběhu testování programu jsem narazil na vysokou časovou náročnost s narůstajícím počtem vzorků.

Na začátek jsem si nastavil parametry, se kterými jsem pracoval:

```
# Nastavení programu
pocet_vzorku = range(1, 2000, 500)
epsilon = 0.001  # Max přípustná chyba
int_start = 0.5  # Začátek vybraného intervalu
int_stop = 2*math.pi  # Konec vybraného intervalu
dyn_krok = 1e-3  # Nastavení dynamického kroku
stat_krok = 1e-3  # Nastavení statického kroku
```

Program zavolá funkci "vysledek()" kde jako argument využije námi nastavenou funkci. Pro každý zvolený počet vzorků provede derivace. Na začátek každého běhu cyklu si nastavíme aktuální vektor "x_vec", který obsahuje námi definovaný prostor rozdělený podle počtu vzorků.

```
def vysledek(f):
    for pocet in pocet_vzorku:
        print(f"Počítáme s {pocet} vzorky z intervalu <{int_start}, {int_stop}> funkce {f}.")
        x_vec = np.linspace(int_start, int_stop, pocet)
```

První provedeme analytickou derivaci a změříme čas jejího provedení. Poté se pustíme do změření času a získání hodnot pro derivaci se statickým krokem a derivaci s dynamickým krokem.

```
        start_time = time.time()
        analyticka_derivace = 1 / x_vec
        end_time = time.time()
        analytic_time = end_time - start_time
        print(f"Analytická derivace: {sum(analyticka_derivace)}")

        stat_values, stat_time = measure_time(static_step, f, x_vec, 0.001)
        dyn_values, dyn_time = measure_time(dynamic_step, f, x_vec, dyn_krok)

```

## Měření časů:

Pro měření časové náročnosti jsem si vytvořil funkci, která využívá *args pro odeslání libovolného seznamu parametrů funkci tak, aby bylo její využití variabilnější. Výsledek měřené funkce uložíme do proměnné “result” a vrátíme spolu s údajem o časové náročnosti.

```
# Měření časové náročnosti
def measure_time(func, *args):
    start_time = time.time()
    result = func(*args)
    end_time = time.time()
    return result, end_time - start_time
```

## Derivace se statickým krokem:

Pro derivaci se statickým krokem jsem si vytvořil seznam hodnot, do kterého přidávám hodnotu pro každý vzorek z vektoru prostoru “x_vec". Poté pomocí funkce “subs()” z knihovny sympy uložíme do proměnné “f_x0” výchozí bod vektoru. Do proměnné “f_x1” uložíme posunutý prvek vektoru podle zvoleného kroku. Nakonec spočítáme hodnotu pro aktuální iteraci a přidáme ji do seznamu hodnot a vrátíme sumu tohoto seznamu jako výsledek derivace.

```
# Statická metoda derivace
def static_step(funkce, x_vec, krok):
    hodnoty_stat = []

    for i in range(len(x_vec)):
        f_x0 = funkce.subs(x, x_vec[i])
        f_x1 = funkce.subs(x, x_vec[i] + krok)
        iterace = (f_x1 - f_x0) / krok
        hodnoty_stat.append(iterace)

    print(f"Statický krok: {sum(hodnoty_stat)}")
    return sum(hodnoty_stat)

```

## Derivace s dynamickým krokem:

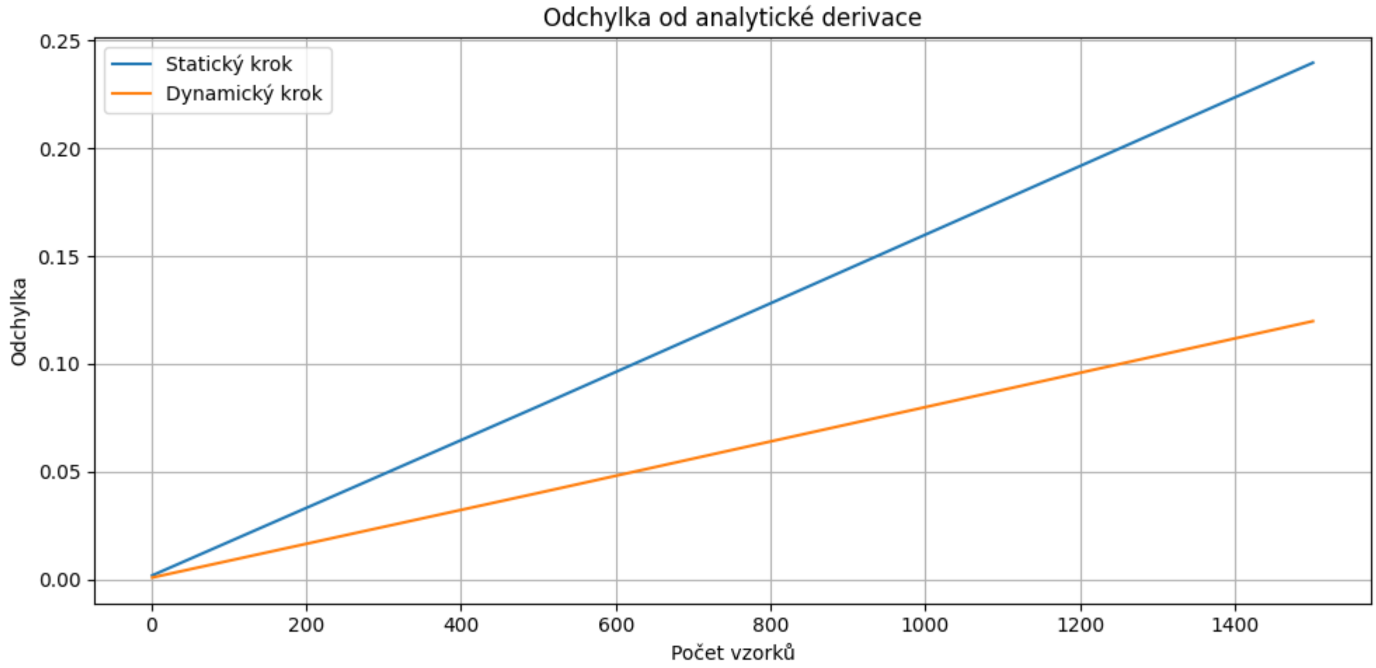
Stejně jako u derivace se statickým krokem jsem si vytvořil seznam, do kterého uložím hodnoty jednotlivých iterací a provedu derivaci. Následně se porovnám, zda chyba je větší, než nastavená maximální přípustná chyba. Pokud ano, tak změním krok a provedu další iteraci. Spočítám aktuální chybu, a přidám hodnotu nové iterace do seznamu hodnot. To opakuji pro všechny vzorky ve vektoru.

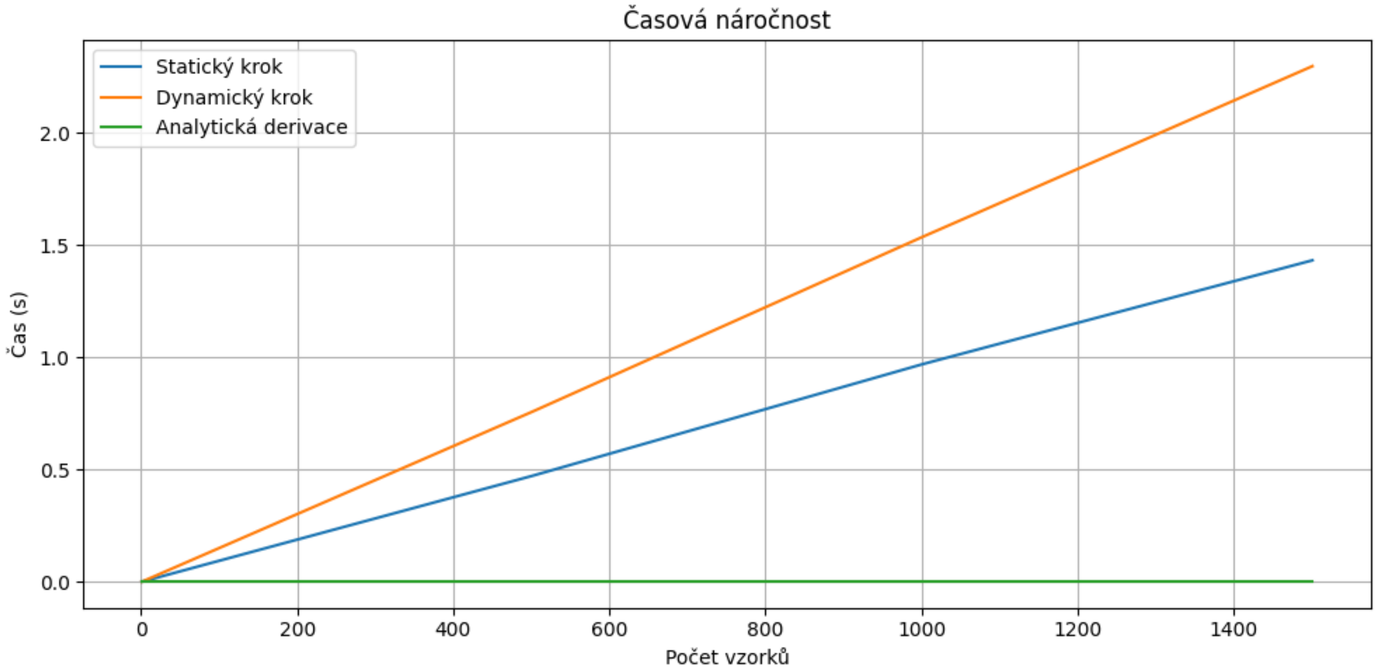
```python
# Dynamická metoda derivace
def dynamic_step(funkce, x_vec, krok):
    hodnoty_dyn = []

    for i in range(len(x_vec)):
        h = krok  # Nastavení dynamického kroku na původní hodnotu pro každou iteraci
        f_x0 = funkce.subs(x, x_vec[i])
        f_x1 = funkce.subs(x, x_vec[i] + h)
        iterace = (f_x1 - f_x0) / h
        chyba = 1

        while chyba > epsilon:
            h /= 2  # Změna kroku
            f_x0 = funkce.subs(x, x_vec[i])
            f_x1 = funkce.subs(x, x_vec[i] + h)
            nova_iterace = (f_x1 - f_x0) / h
            chyba = abs(nova_iterace - iterace)  # Výpočet chyby
            iterace = nova_iterace
            hodnoty_dyn.append(iterace)

    print(f"Dynamický krok: {sum(hodnoty_dyn)}\n")
    return sum(hodnoty_dyn)
```

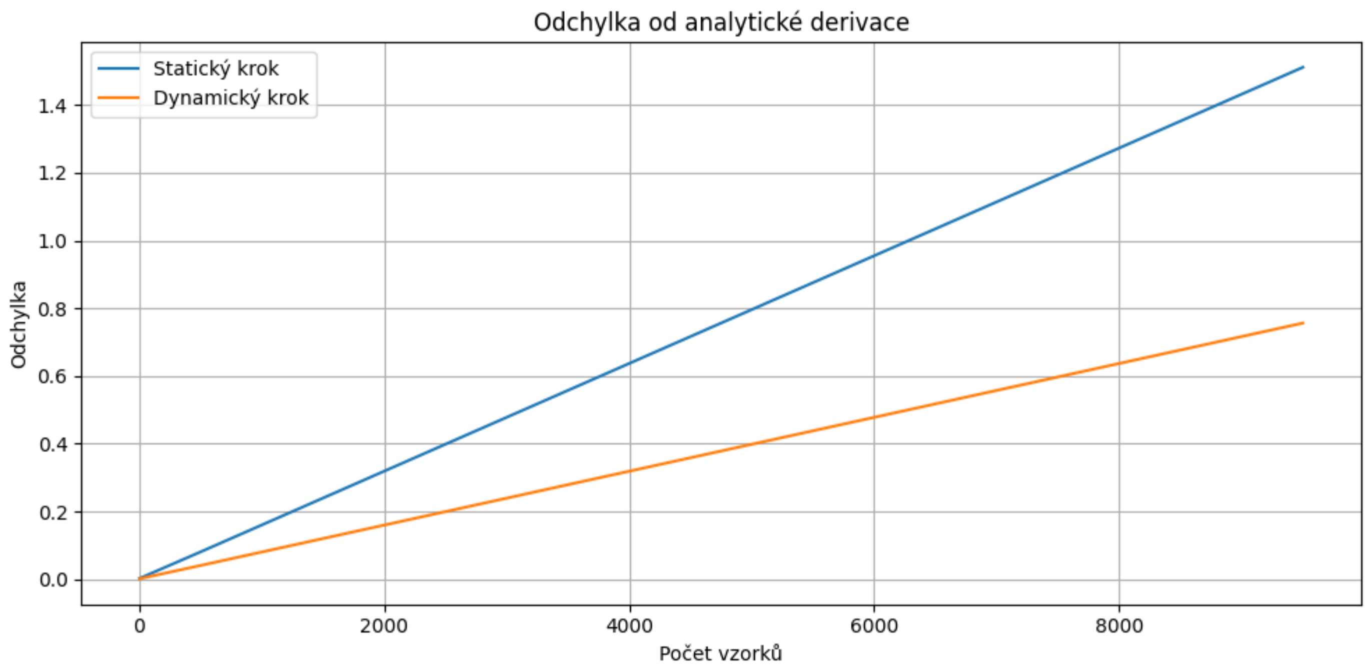
## Závěr:

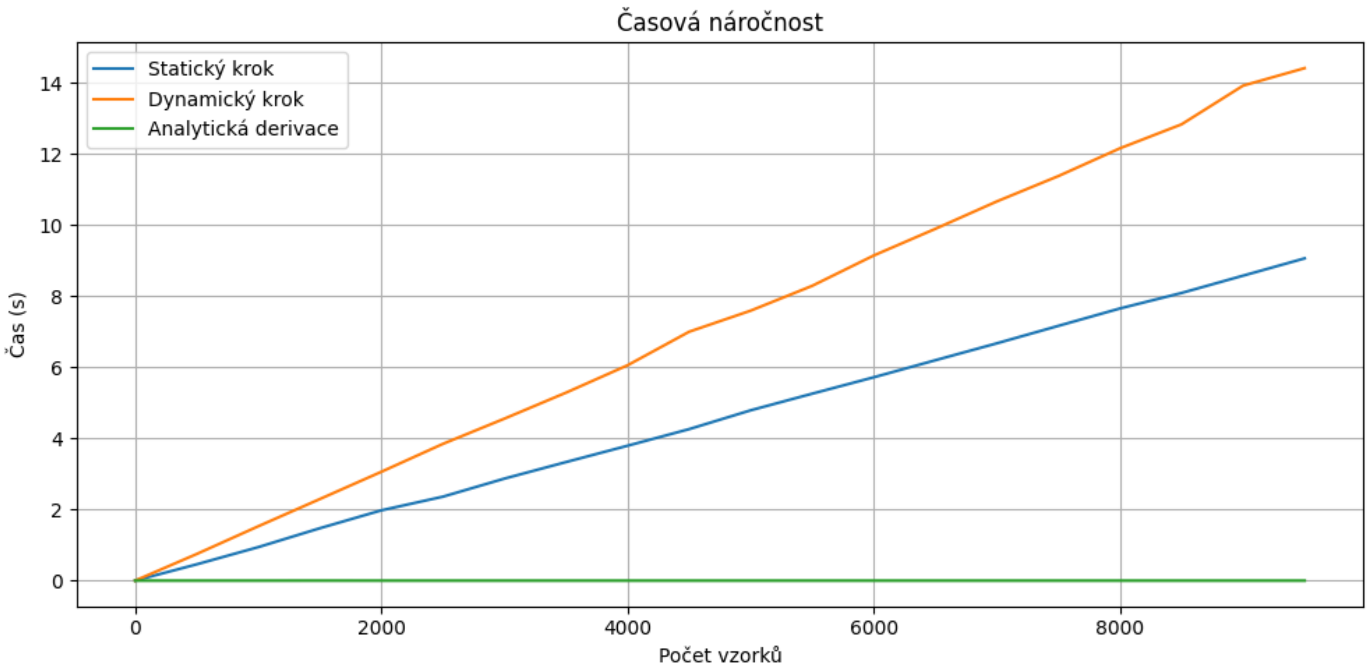
Z grafů vyplývá, že derivace s dynamickým krokem je přesnější, než derivace se statickým krokem. Na druhou stranu derivace se statickým krokem je daleko méně časově náročná. Když se zvýší počet vzorků, derivace s dynamickým krokem může běžet i desítky sekund.

Dále lze říct, že časová náročnost s krokem, ať už dynamickým, nebo statickým, je několikanásobně pomalejší, než analytické řešení derivace.


Odchylka od analytické derivace
Statický krok
Dynamický krok
Odchylka
0.00
0.05
0.10
0.15
0.20
0.25
0
200
400
600
800
1000
1200
1400
Počet vzorků


Časová náročnost
Statický krok
Dynamický krok
Analytická derivace
Čas (s)
0.0
0.5
1.0
1.5
2.0
0
200
400
600
800
1000
1200
1400
Počet vzorků


Odchylka od analytické derivace
Statický krok
Dynamický krok
Odchylka
Počet vzorků
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
2000
4000
6000
8000


Časová náročnost
Statický krok
Dynamický krok
Analytická derivace
Čas (s)
Počet vzorků
0
2
4
6
8
10
12
14
0
2000
4000
6000
8000