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.

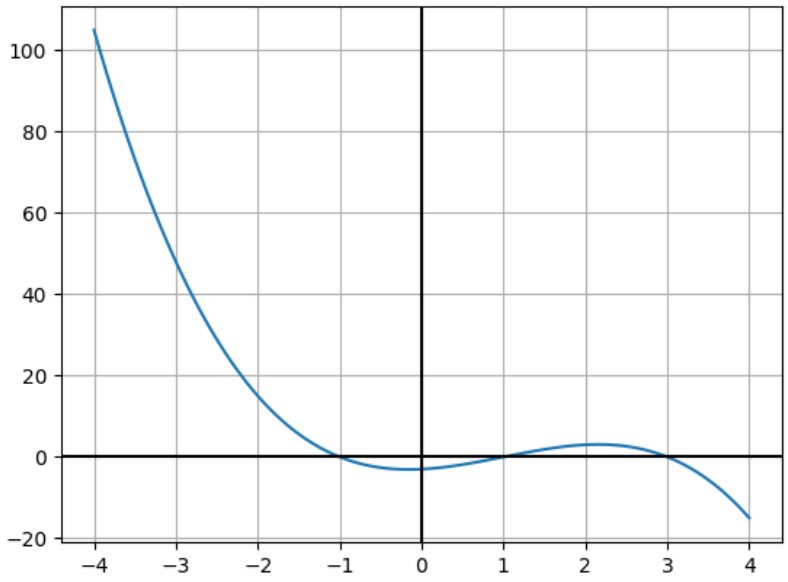
Řešení:

Pro tento úkol jsem si vybrala tři různé funkce (polynomickou, exponenciální a harmonickou). U všech těchto funkcí jsem pak hledala kořeny pomocí metody bisekce a Newtonovy metody, které jsem pak porovnala mezi sebou.

Polynomická funkce:

Jako první jsem si definovala svojí polynomickou funkci. Poté jsem si vytvořila graf, abych věděla, kolik kořenů budu muset najít a v jakých intervalech je budu hledat.

Graf jsem vytvořila pomocí knihoven numpy a matplotlib. Np.linspace mi vytvoří pole x se 100 rovnoměrně rozmístěnými hodnotami mezi -4 a 4. Tyto hodnoty pak vložím na osu x a na osu y vložím znovu tyto hodnoty ale v mé definované funkci. Pak už jenom zvýrazním osu x v 0, abych viděla, kolik kořenů budu hledat. U mé funkce už bude z grafu vidět, že kořeny jsou v bodech -1, 1 a 3. Ovšem pro účely ověření funkčnosti metod si stejně zadám větší intervaly.



Bisekce:

U metody bisekce potřebuji zadat interval, v kterém budu kořen hledat a toleranci, tedy přijatelnou úroveň chyby, s kterou chci kořen najít. Do while cyklu dám podmínku, že bude běžet, dokud absolutní hodnota z (a – b), což je naše počáteční a konečná hodnota intervalu, bude větší nebo rovna toleranci. Pokud tomu tak bude, vytvoří proměnnou c. C nám tento interval rozpůlí a novou hodnotu, kterou z tohoto rozpůlení dostanu, dosadím do definovaného vzorce. Poté můžou nastat dvě možnosti:

1. Jestliže výsledek po dosazení bude větší nebo roven toleranci, tak ze svého počátečního intervalu udělám c.
2. Pokud nastane opak, udělám ze svého konečného intervalu c.

Toto opakuji tak dlouho, dokud právě absolutní hodnota počátku – konce intervalu bude menší než tolerance, až toto nastane, tak vrátím c, které je kořenem funkce.

```
def bisekce(a, b, tolerance):
    while abs(a - b) >= tolerance:
        c = (a + b) / 2 #Rozdělení intervalu
        vzorec = f(a) * f(c) #Dosazení do vzorce
        if vzorec >= tolerance:
            a = c #Posunutí začátečního intervalu
        else:
            b = c #Posunutí konečného intervalu
    return c
```

Poté už stačí jen metodu použít. Intervaly si zadám podle grafu funkce a vzhledem k tomu, že hledám kořeny 3, tak funkci budu muset zavolat 3x vždy s jiným intervalem. Z grafu už jsou sice kořeny vidět, ale intervaly jsem si přeci jen pro vyzkoušení zvětšila. Jako poslední věc,

jsem si ještě před zavoláním funkce přidala čas, abych mohla spočíst délku trvání, k tomu je využita knihovna time.

```
zacatek_bisekce = time.time()
koren1 = bisekce(-5, 0, 1e-8)
koren2 = bisekce(0, 2, 1e-8)
koren3 = bisekce(2, 5, 1e-8)
cas_bisekce = (time.time() - zacatek_bisekce) * 1000  #Převede na milisekundy

print(f"Metoda Bisekce - Kořen 1: {koren1}")
print(f"Metoda Bisekce - Kořen 2: {koren2}")
print(f"Metoda Bisekce - Kořen 3: {koren3}")
print(f"Čas nalezení kořenů (Metoda Bisekce): {cas_bisekce:.6f} milisekundy")
```

Newton:

U metody Newton potřebuji zadat stejně jako u bisekce interval a toleranci. Ty si zadám úplně stejné. V definici metody si hned ze začátku vytvořím proměnnou x_sym, která nám pomocí sp.Symbol(„x“), umožní pracovat se symbolickým výrazem „x“ jako s matematickou proměnnou. Dále si s tou proměnnou ze své původní funkce vytvořím symbolický výraz a z tohoto symbolického výrazu si pak můžu vytvořit výraz numerický, což nám umožní vyhodnotit funkci pro číselné hodnoty x a také funkci zderivovat. Dále definuji funkci df, která numericky vypočítá derivaci funkce pomocí aproximace. Parametr h slouží k určení velikosti kroku aproximace. Stejně jako u bisekce si teď interval rozdělím do proměnné c a stejně jako u bisekce vytvořím while cyklus, který poběží, dokud absolutní hodnota z (c – a) bude větší nebo rovna toleranci. Pokud tomu tak bude, počáteční interval se posune, tudíž z a se stane c. Toto pak dosadím do vzorce a cyklus se opakuje s novým intervalem, dokud právě absolutní hodnota počátku – konce intervalu bude menší než tolerance, až toto nastane, tak vrátím c, které je kořenem funkce.

```
def newton(a, b, tolerance):
    x_sym = sp.Symbol("x") #Převod "x" na matematickou proměnou
    f_sym = -x_sym**3 + 3*x_sym**2 + x_sym - 3 #Převod funkce na symbolickou funkci
    f_num = sp.lambdify(x_sym, f_sym, "numpy") #Převod funkce na numerickou funcki

    def df(x, h = 1E-3): #Derivace funkce
        return (f_num(x + h) - f_num(x - h)) / (2 * h)

    c = (a + b) / 2 #Rozdělení intervalu
    while abs(c - a) >= tolerance:
        a = c
        c = a - f_num(a) / df(a) #Dosazení do vzorce
    return c
```

Na konec metodu použiji, se stejnými hodnotami jako u bisekce, 3x zavolám a spočítám čas pro nalezení kořenů.

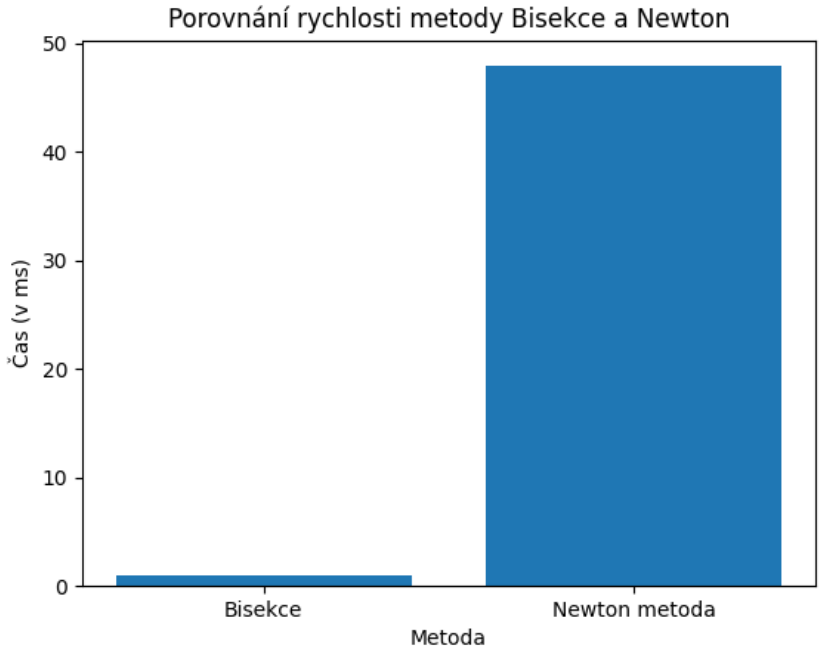
```python
zacatek_newton = time.time()
koren1 = newton(-5, 0, 1e-8)
koren2 = newton(0, 2, 1e-8)
koren3 = newton(2, 5, 1e-8)
cas_newton = (time.time() - zacatek_newton) * 1000  #Převede na milisekundy

print(f"Newton metoda - Kořen 1: {koren1}")
print(f"Newton metoda - Kořen 2: {koren2}")
print(f"Newton metoda - Kořen 3: {koren3}")
print(f"Čas nalezení kořenů (Newton metoda): {cas_newton:.6f} milisekundy")
```

Vyhodnocení:

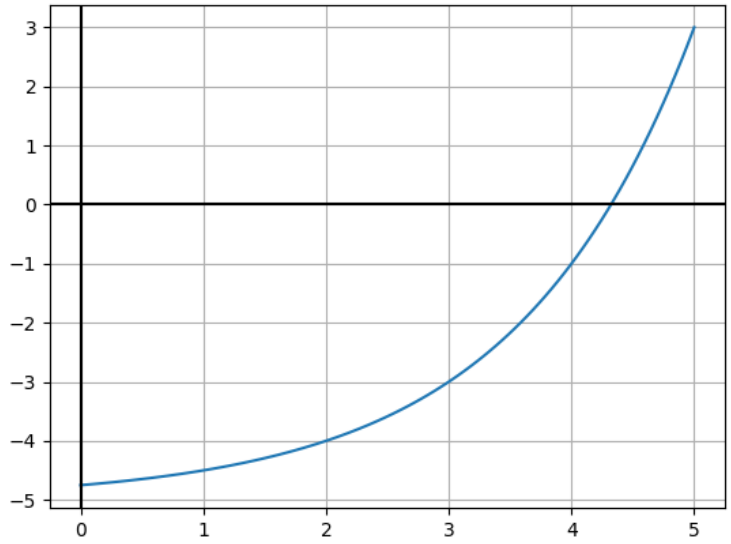
```
Metoda Bisekce - Kořen 1: -1.0000121500343084
Metoda Bisekce - Kořen 2: 0.9999771192669868
Metoda Bisekce - Kořen 3: 2.999987607821822
Čas nalezení kořenů (Metoda Bisekce): 0.997066 milisekundy
Newton metoda - Kořen 1: -1.0
Newton metoda - Kořen 2: 1.0
Newton metoda - Kořen 3: 3.0000000000000004
Čas nalezení kořenů (Newton metoda): 47.870874 milisekundy
```

Přesné kořeny měly být -1, 1 a 3. Tudíž jak můžeme vidět z outputu, metoda bisekce nenachází tak přesné výsledky jako Newtonova metoda, za to je ale zase o něco rychlejší. Newtonova metoda našla 2 ze 3 kořenů na desetinné místo přesně, ale trvalo to o něco déle. Samozřejmě čas je v tomto případě irelevantní, protože dané funkce jsou jednoduché a obě metody zvládly kořeny najít do sekundy.



Exponenciální funkce:

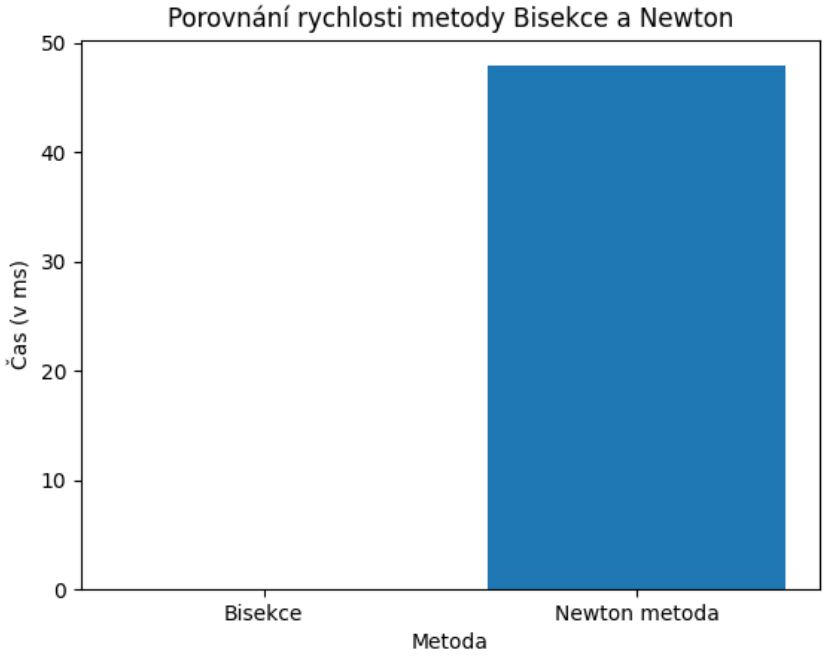
Kód pro bisekci i Newtonovo metodu je naprosto stejný jako u polynomické funkce, tudíž už ho nebudu znovu popisovat a podíváme se pouze na výsledek. Jediným rozdílem bude, že na základě grafu funkce budu hledat pouze jeden kořen, který se bude nacházet mezi 4 a 5.



Vyhodnocení:

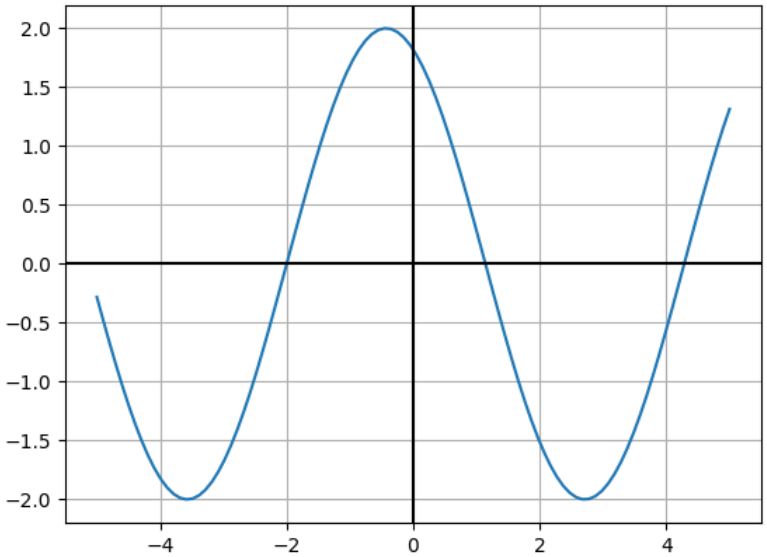
```
Metoda Bisekce - Kořen: 4.321899421513081
Čas nalezení kořenů (Metoda Bisekce): 0.000000 milisekundy
Newton metoda - Kořen: 4.321928094887363
Čas nalezení kořenů (Newton metoda): 44.907808 milisekundy
```

Přesný kořen je 4.3219280935382. V tomto případě jsou obě metody velice přesné, ale stejně jako u minulého příkladu Newton metoda je o něco přesnější, ale zároveň o něco pomalejší.



Harmonická funkce:

Stejně jako u minulé funkce kód pro bisekci i Newtonovo metodu je naprosto stejný. Ovšem teď mám harmonickou funkci se sinusem. Takováto funkce má nekonečné množství kořenů, proto si vyberu pouze určitý interval, v kterém kořeny budu hledat. Vybrala jsem si interval od -5 do 5, z grafu vidím, že kořeny v tomto intervalu budou 3, a kde se cca budou nacházet, podle toho si nastavím intervaly do svých metod.



Vyhodnocení:

```
Metoda Bisekce - Kořen 1: -2.0000457670539618
Metoda Bisekce - Kořen 2: 1.1415443494915962
Metoda Bisekce - Kořen 3: 4.2831420954316854
Čas nalezení kořenů (Metoda Bisekce): 0.997305 milisekundy
Newton metoda - Kořen 1: -2.0
Newton metoda - Kořen 2: 1.1415926535897933
Newton metoda - Kořen 3: 4.283185307179586
Čas nalezení kořenů (Newton metoda): 46.903372 milisekundy
```

Přesné kořeny jsou -2, 1.1415922025004, 4.2831853092398. Vidíme stejný výsledek jako u minulých funkcí, tudíž obecně můžeme říct, že Newton metoda je pomalejší, ale přesnější oproti metodě bisekce.