

# Matematický software

Zápočtový dokument

**Jméno:** Michal Melicher

**Kontaktní email:** melicmic@gmail.com

**Datum odevzdání:** 12.07.2022

**Odkaz na repozitář:** <https://github.com/mmelicher91/kmsw-seme>

# 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:

| n   | Iterační řešení [s]  | Cramerovo řešení [s]   | Symbolické řešení [s] |
|-----|----------------------|------------------------|-----------------------|
| 2   | 0.000427229309082031 | 0.00040721893310546875 | 0.04103398323059082   |
| 5   | x                    | 0.00030350685119628906 | 0.1396484375          |
| 10  | x                    | 0.0003478527069091797  | 0.43741941452026367   |
| 20  | x                    | 0.09691977500915527    | 2.1939613819122314    |
| 50  | x                    | 0.004071235656738281   | x                     |
| 100 | x                    | 0.07618141174316406    | x                     |

Cramerovo řešení je nejrychlejší pro nalezení výsledku matice. Symbolické řešení selhává od velikosti matice 20x20. Iterační nedává relevantní výsledky.

### Iterační řešení

```
def jacobi(A, b, niteraci, x0=None):  
    x = x0 if x0 else np.ones(len(A))  
    D = np.diag(A)  
    R = A - np.diagflat(D)  
    for i in range(niteraci):  
        x = (b - np.dot(R, x)) / D
```

### Cramerovo řešení

```
def cramer(A, b):
    N = A.shape[1]
    mask = np.broadcast_to(np.diag([1 for i in range(N)]), [N, N,
N]).swapaxes(0, 1)
    Ai = np.where(mask, np.repeat(b, N).reshape(N, N), A)
    x = np.linalg.det(Ai) / np.linalg.det(A)
    return x
```

### Symbolické řešení

```
def symbolicke_reseni(A, b):
    N = A.shape[1]
    s = sympy.symbols(','.join(string.ascii_lowercase[0:N]))
    rows = []
    for i in range(N):
        str_expr = "+".join([f"{A[i,j]}*{s[j]}" for j in range(N)])
        sympy_eq = sympy.sympify(str_expr)
        rows.append(sympy.Eq(sympy_eq, b[i]))
    x = sympy.solve(rows, s)
    return x
```

# Interpolace a aproximace funkce jedné proměnné

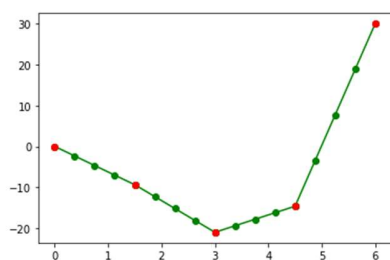
## Zadání:

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 rozkmitajte), 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.

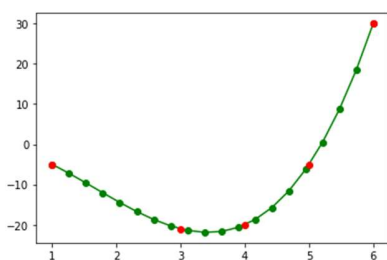
## Řešení:

Polynomičká funkce:  $x^3 - 5 * x^2 - x$

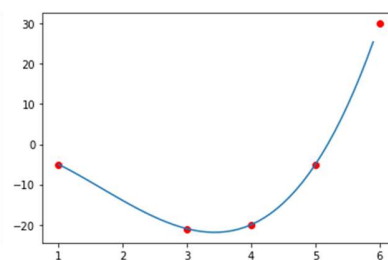
Lineární aproximace



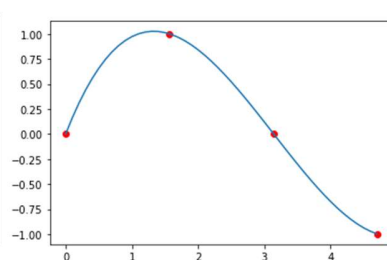
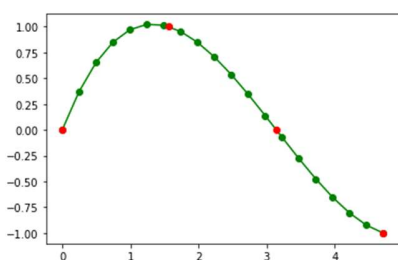
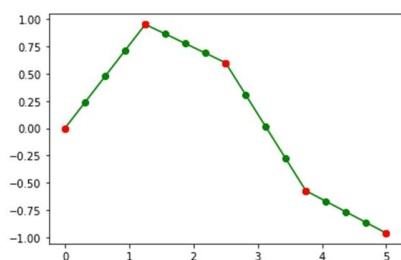
Lagrange aproximace



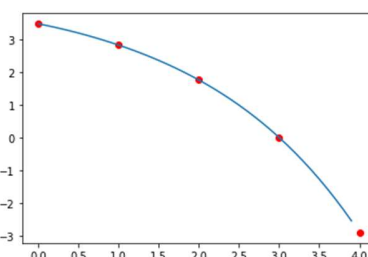
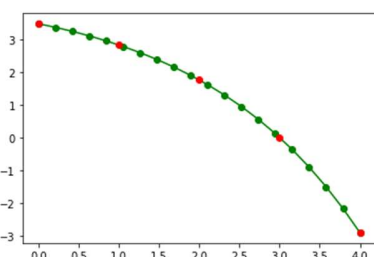
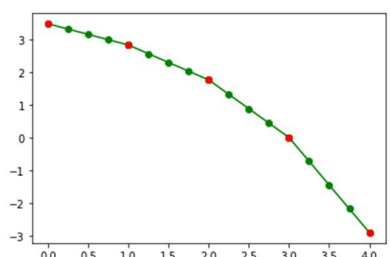
Newtonova aproximace



Harmonická funkce:  $\sin(x)$



Exponenciální funkce:  $e^{\frac{3}{2}} - e^{\frac{x}{2}}$



### Lineární aproximace

```
def linear(x, a, b):  
    return f(a) + (f(b)-f(a))/(b-a)*(x-a)  
  
for i in range(len(x)-1):  
    xinterpol = np.linspace(x[i], x[i+1], ninterpol) #body, které mě  
zajímají uvnitř interpolační funkce  
    g = linear(xinterpol, x[i], x[i+1]) #hodnoty bodů interpolační funkce  
    xphi.extend(xinterpol)  
    phi.extend(g)
```

### Lagrange aproximace

```
from scipy.interpolate import lagrange  
  
phi = lagrange(x, fx)  
print(phi)  
print(phi.coef)  
  
n = 20  
xphi = np.linspace(x[0], x[-1], n)
```

### Newtonova aproximace

```
def divided_diff(x, y):  
    n = len(y)  
    coef = np.zeros([n, n])  
    coef[:,0] = y  
    for j in range(1,n):  
        for i in range(n-j):  
            coef[i][j] = \  
                (coef[i+1][j-1] - coef[i][j-1]) / (x[i+j]-x[i])  
    return coef  
  
def newton_poly(coef, x_data, x):  
    n = len(x_data) - 1  
    p = coef[n]  
    for k in range(1,n+1):  
        p = coef[n-k] + (x -x_data[n-k])*p  
    return p
```

# 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:

Časová náročnost jednotlivých metod:

|                        |                         |
|------------------------|-------------------------|
| Bisekce pro n:         | 0.32941603660583496 s   |
| Bisekce pro odchylku:  | 0.2155601978302002 s    |
| Regula falsi:          | 0.0005152225494384766 s |
| Newtownova metoda:     | 0.0003044605255126953 s |
| Newton. m. z knihovny: | 0.006808280944824219 s  |

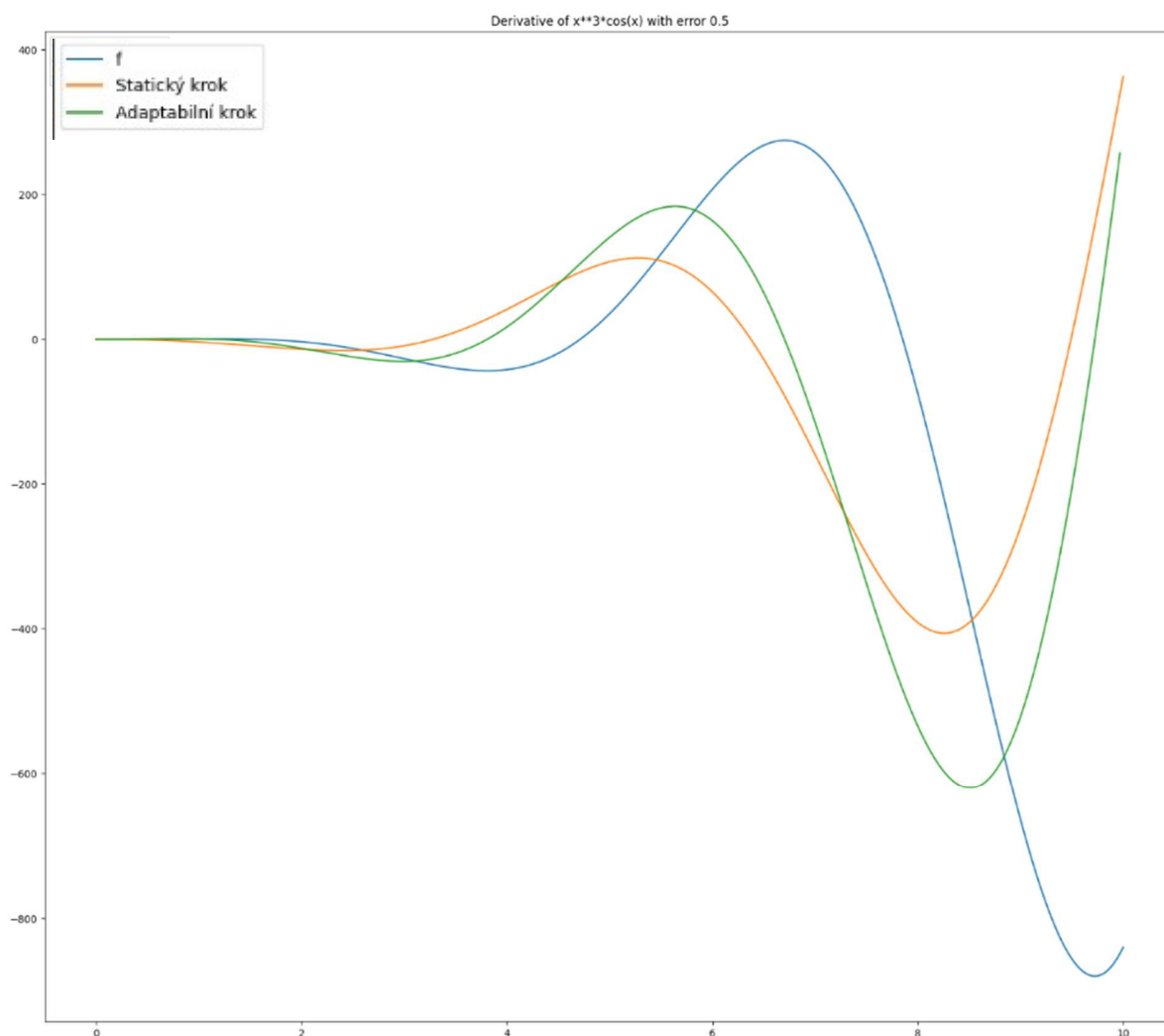
| Bisekce pro odchylku  | Regula falsi   | Newtonova metora   |
|---|--|--|
| <pre>while b**2 - a**2 &gt; 2*eps:     x = (a + b)/2     if f(a)*f(x) &lt; 0:         b = x     else:         a = x</pre> | <pre>xnew = (a+b)/2 xold = a  while abs(xnew - xold) &gt; delta:     xold = xnew     xnew = a - f(a)*(b-a)/(f(b)-f(a))     if f(a)*f(xnew) &lt; 0:         b = xnew     else:         a = xnew</pre> | <pre>def df(x, h = 1E-5):     return (f(x+h)-f(x-h))/(2*h)  xnew = (a+b)/2 xold = a  while abs(xnew - xold) &gt; delta:     xold = xnew     xnew = xold - f(xold)/df(xold)</pre> |

# Derivace funkce jedné proměnné

## Zadání:

Numerická derivace je velice krátké téma. V hodinách jste se dozvěděli o nejvyžívanějších 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:



```

def finite_diff(f, x, type="forward", error=0.01, static=True):

    if (type == "backward"):
        g = lambda x, h: (f(x) - f(x-h))/h
    elif (type == "central"):
        g = lambda x, h: (f(x+h) - f(x-h))/(2*h)
    elif (type == "forward"):
        g = lambda x, h: (f(x+h) - f(x))/h

    D = []
    X = []
    x_ = x[0]
    if(not static):
        x_ = x[0]
        while x_ < x[-1]:

            h1 = error
            h2 = error/2

            dx1 = g(x_, h1)
            dx2 = g(x_, h2)

            while(abs(dx1 - dx2)>= error):
                h1 = h2
                h2 = h2/2

                dx1 = g(x_, h1)
                dx2 = g(x_, h2)

            D.append(dx2)
            X.append(x_)
            x_ = x_ + h2
    else:
        h = 2*np.sqrt(error)
        for x_ in x:
            dx = g(x_, h)
            D.append(dx)
            X.append(x_)

    return D, X

```

# 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 vypočítat 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:

### Výsledky integrací:

**Polynomická funkce:**  $\int_1^4 \frac{(x+1)^2}{x} dx$

Lichoběžníková metoda: 2.2483626815960536

Newton-Cotes vzorec: 2,2438626815960536

Rombergův vzorec: 2.2499999999988605

**Harmonická funkce:**  $\int_1^4 \sin(x+1) dx$

Lichoběžníková metoda: -0.744690621947801 + 0.05\*sin(2)

Newton-Cotes vzorec: -0.744690621947801 + 0.05\*sin(2)

Rombergův vzorec: -0.698391520835868 + 0.0285631918311657\*sin(5) + 0.0285631918311657\*sin(2)

**Exponenciální funkce:**  $\int_1^4 e^2 - e^x dx$

Lichoběžníková metoda: 0.05\*E + 163.881446274428

Newton-Cotes vzorec: 0.05\*E + 163.881446274428

Rombergův vzorec: 0.0285631918311657\*E + 0.0285631918311657\*exp(4) + 3.0\*exp(2) + 50.2427279663545

### Lichoběžníková metoda

```
x = a
i = 0

while x < b:
    integral += dx * (f(x) + f(x+dx))/2
    x += dx
```

### Newton-Cotes vzorec

```
n = int((b-a)//dx)+1
integral = f(a) + f(b)

for i in range(1, n):
    integral += 2*f(a+i*dx)
integral *= dx/2
```

### Rombergův vzorec

```
def df(x, h = 1E-5):
    return (f(x+h)-f(x-h))/(2*h)

xnew = (a+b)/2
xold = a

while abs(xnew - xold) > delta:
    xold = xnew
    xnew = xold - f(xold)/df(xold)
```