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:

Mým úkolem bylo porovnat iterační metody s build-in funkcí a následně zjistit, od jaké velikosti matice je vhodné využít daný typ řešení. Jako iterační metodu jsem použil Jacobiho metodu a jako built-in metodu jsem využil funkci liang.solve(), kterou obsahuje knihovna numpy.

Po importu knihoven jsem vytvořil tři prázdné listy (list_matrix,list_liang,list_jacobi):

- **list_matrix** - slouží k evidování velikosti matic,
- **list_liang** - slouží k evidenci času v built-in funkci,
- **list_jacob**i - slouží k evidenci času Jacobiho metody.

Generátor matic

Můj první úkol byl vytvořit generátor čtvercových matic, které bude následně program řešit a porovnávat jejich časovou náročnost. Při testování programu jsem dlouho uvažoval nad formou tvorby matic (jakými čísly jí mám plnit). Nakonec jsem se rozhodl matici A naplnit čísly v rozmezí 0 až 10 v celém rozsahu. Pro matici b jsem zvolil matici plnou jedniček pomocí funkce ones(), která se také nachází v knihovně numpy. V generátoru je také implementovaná podmínka pro eliminaci

singulárních matic pomocí podmínky. Pokud byl determinant matice A roven nule, tak došlo ke generaci nové matice.

```
def matice(matice_range,n):
    for w in range(n,matice_range+1):
        n_sloupcu = n
        n_radku = n
        log = True

        while log == True:
            A = np.random.randint(10, size=(n_radku,n_sloupcu))
            b = np.full((n_radku,1),5)


            if np.linalg.det(A) != 0:
                log = False
                print("----------------------------Matice:----------------------------\n", n, "x", n)
                print(A)
                print("Vektor b:\n", b)
                x0 = np.ones(len(A))
                x_jacobi = jacobi(A,b,n_iteraci,x0,dtype=int)


                linalg_starting_time = timeit.default_timer() #zacatek casomiry

                x_linalg = np.linalg.solve(A, b) #build in metoda

                liang_time = 1000*(timeit.default_timer() -  linalg_starting_time) #vypocet casu
                list_liang.append(liang_time) #append casu do listu
```

```
                print("LIANG: Time difference :", liang_time, "ms")
                print("Výsledky:")
                print("Jacobi:", x_jacobi)
                print("linalg.solve:", x_linalg)
                print("-----------------------------------------------------------------")

                #appending do seznamu
                list_matrix.append(n_radku) #append informaci ohledne velikosti matic

                print("list_liang", list_liang)
                print("list_matrix", list_matrix)

                n+=1
                matice(matice_range,n)
                return

            else:
                log = False
                matice(matice_range,n)
                return
```

Pomocí knihovny time jsem vždy aktivoval timer, po dokončení metody jsem čas začátku odečetl od aktuálního času. Tyto časy jsem následně přidal do seznamů pro následný output.

Následně došlo k zavolání funkce jacobi(), která provedla iterační metodu. Zde rozložíme matici A na tři sub matice:

- D - diagonála matice A.
- L - matice, hodnoty v trojúhelníku pod diagonálou.
- U - matice, hodnoty v trojúhelníku nad diagonálou.

```
def jacobi(A,b,n_iteraci,x0,dtype = int):
    jacobi_starting_time = timeit.default_timer() #casomira jacobi start

    x = x0
    D = np.diag(A)
    L = np.tril(A, k=-1)
    U = np.triu(A, k=1)

    for i in range(n_iteraci):
        x = (b - np.matmul((L + U),x))/D
        print("iterace:",i, "x=",x)

    jacobi_time = 1000*(timeit.default_timer() - jacobi_starting_time)
    list_jacobi.append(jacobi_time) #casomira jacobi end

    print("JACOBI: Time difference :", jacobi_time, "ms")
    print("list_jacobi", list_jacobi)
    return x
```

Následně tyto komponenty vložíme do vzorce, který je ve for cyklu v rozsahu počtu iterací. Po dokončení zase zaznamenáme čas a vložíme ho do listu.

Po úspěšném zaznamenání času daných metod a velikosti matic přichází čas na zaznamenání výsledků do grafu.

Po přidání popisků a legend do spojnicového grafu

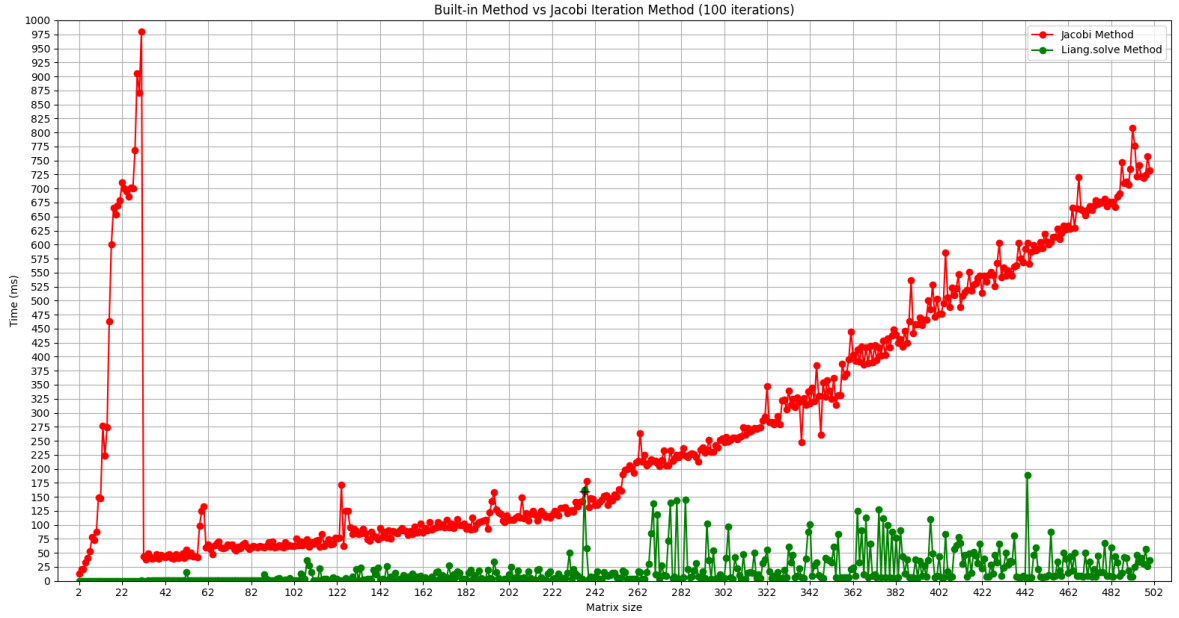
```
#OUTPUT
plt.plot(list_matrix, list_jacobi, 'o-r')
plt.legend(['Jacobi Method'])
plt.plot(list_matrix, list_liang, 'o-g')
plt.title('Built-in Method vs Jacobi Iteration Method (25 iterations)')
plt.ylabel('Time (ms)')
plt.xlabel('Matrix size')
plt.legend(['Jacobi Method','Liang.solve Method'])
plt.grid()


#zjisteni pruseciku
intersection_index = np.argmin(np.abs(np.array(list_jacobi) - np.array(list_liang)))
intersection_x = list_matrix[intersection_index]
intersection_y = list_jacobi[intersection_index]


#zobrazeni krizku v pruseciku
plt.plot(intersection_x, intersection_y, 'k+', markersize=10)


#nastaveni x axis
plt.xlim(min(list_matrix) - 10, max(list_matrix) + 10)
plt.xticks(np.arange(min(list_matrix), max(list_matrix) + 20, 20))

#nasatveni y axis
plt.ylim(0, max(max(list_jacobi), max(list_liang)) + 10)
plt.yticks(np.arange(0, max(max(list_jacobi), max(list_liang)) + 25, 25))

plt.show()
```

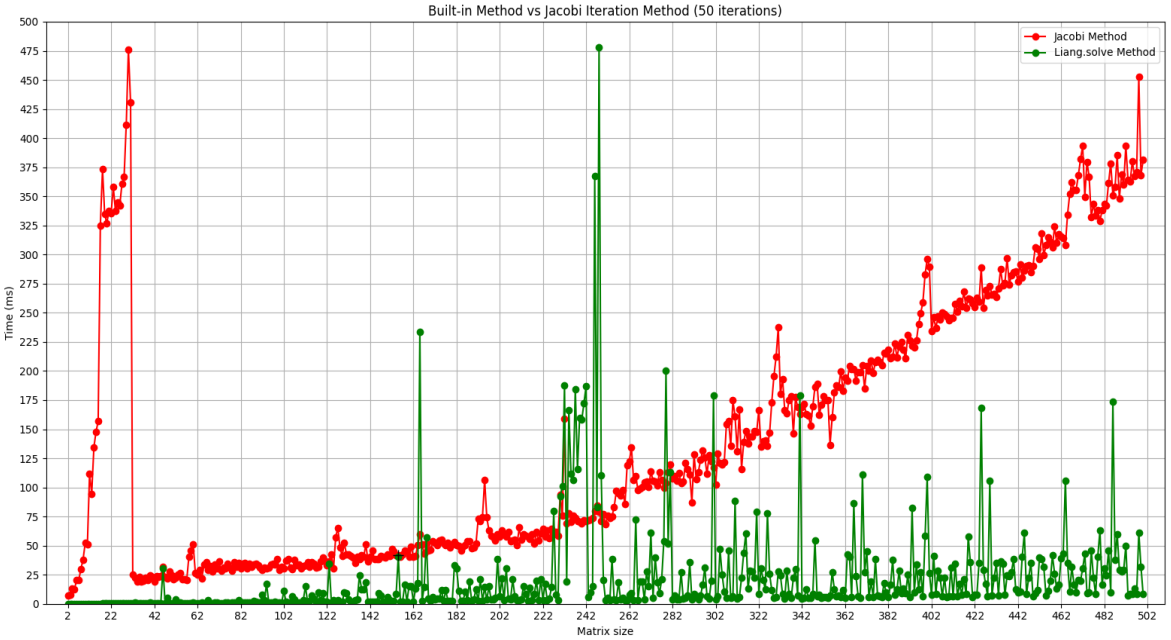
Pro zjištěni průsečíku jsem použil funkci argmin() z knihovny numpy. Tato funkce chybí sloveso nejmenší absolutní hodnoty rozdílu mezi prvky seznamů. Tento index následně použiji na zobrazení časového průsečíku na daném rozměru matice.

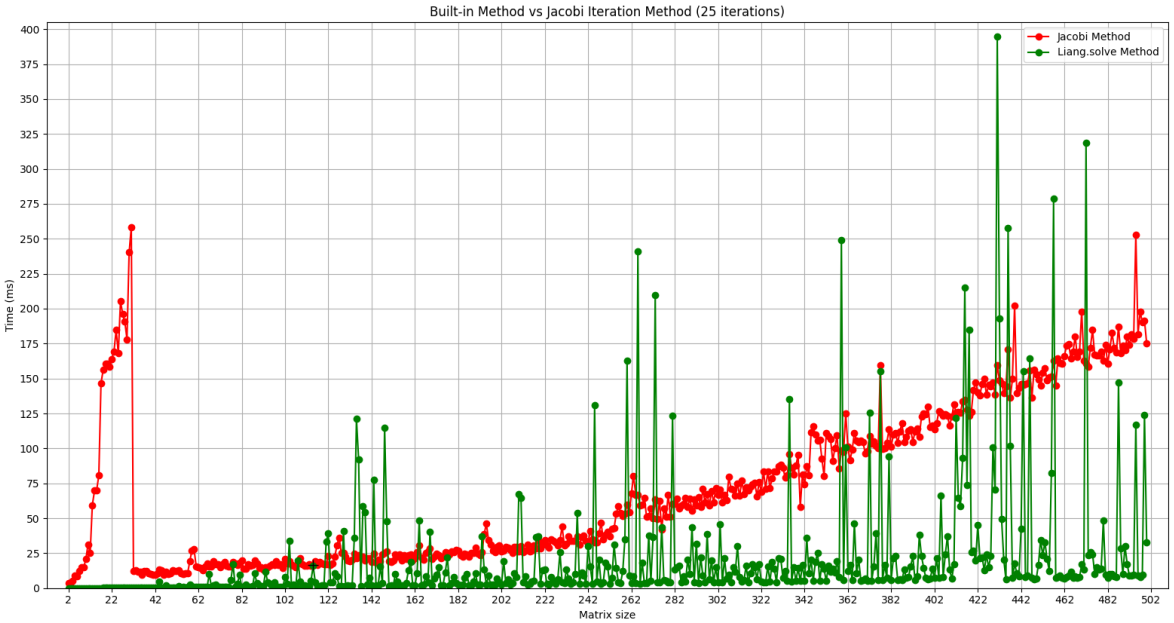
Následně jsem zformátoval obě osy pro lepší čitelnost grafu. U každého axisu zjistím nejmenší a nejvyšší hodnotu a připočítám k ní polštář hodnot. Následně definuji kroky na daném vektoru xticks() a yticks().

Nakonec graf zobrazím pomocí funkce show().

Výsledky


Built-in Method vs Jacobi Iteration Method (100 iterations)
Jacobi Method
Liang.solve Method
Time (ms)
Matrix size


Built-in Method vs Jacobi Iteration Method (50 iterations)
Jacobi Method
Liang.solve Method
Time (ms)
Matrix size



Na závěr bych chtěl říct, že si nejsem rozhodně jist výsledkem celé analýzi. Podle mého názoru mám v programu chybu. Nejspíše se jedná o matematickou chybu. Při generování větších matic mi dochází k podivným výsledným prvkům a hodnoty se daných metod se spíše nepodobají.

ODKAZ NA GITHUB:

https://github.com/Marty808s/ZAKL_LIN_ALG_3

- ve složce GRAPHS jsou i grafy