Multidiszciplináris tudományok, 10. kötet. (2020) 2 sz. pp. 338-349 **https://doi.org/10.35925/j.multi.2020.2.38**

NUMERIKUS SZÁMÍTÁSOK HATÉKONYSÁGÁNAK VIZSGÁLATA PYTHON, MATLAB ÉS OCTAVE PROGRAMCSOMAGOKKAL

Gyulai Márk
programtervező informatikus hallgató, Miskolci Egyetem, Informatikai Intézet
3515 Miskolc, Miskolc-Egyetemváros

Absztrakt

A numerikus analízis a matematikai problémák közelítő, számítógéppel is hatékonyan elvégezhető megoldásával foglalkozik. A cikkben bemutatott kutatás célja, hogy összehasonlítja a Python NumPy és SciPy, a Matlab és GNU Octave programcsomagok alapvető numerikus algoritmusainak sebességét. A bemutatásra kerülő sebességtesztek olyan alapvető numerikus műveletekre koncentrálnak, mint a mátrix műveletek, interpolációk, lineáris egyenletrendszer megoldása, valamint a numerikus integrálás és deriválás. Különböző méretű tesztfeladatok esetén, az egyes műveletek elvégzéséhez szükséges időket fogjuk összevetni, valamint a mérés során figyelembe vesszük az egyes implementációk közötti különbségeket is.

Kulcsszavak: *Python, Matlab, Octave, numerikus analízis, mátrix műveletek, numerikus integrálás és deriválás, sajátérték, sajátvektor*

Abstract

Numerical analysis deals with the approximate solution of mathematical problems that can be performed efficiently with computers. The goal of this article is that, to find out, which programming tool is the fastest in various numerical problems. The software tools that I test are the Python with NumPy and SciPy packages, the Matlab and the GNU Octave. Our tests are focusing at the basic numerical operations, like matrix operations, interpolations or solving linear equation systems or compute derivates and integrates and trying to find out which tool is the fastest in the given test and which is the fastest overall.

Keywords: *Python, Matlab, Octave, numerical Analysis, matrix operations, numerical integration and derivation, **e**igenvalue, eigenvector*

1. Bevezetés

A cikk témája különböző széles körben használt matematikai eszközök hatékonyságának vizsgálata alapvető numerikus számítások elvégzése esetén. Ilyen numerikus problémák, például a mátrix transzponálás, mátrix felbontások (LU, Cholesky), lineáris egyenletrendszerek megoldása, interpolációk vagy a numerikus deriválás és integrálás. A numerikus eljárások mindig az egzakt megoldást, vagy annak egy közelítését fogják adni. Numerikus számításokhoz sokféle szoftvert fejlesztettek ki, jelen cikkben 3 technológiára fogunk koncentrálni a **Matlab**, a **Python** és a **GNU Octave** programcsomagokra. Az utóbbi két rendszer nyílt forráskódú, ingyenesen elérhető. Más szerzők [11] is foglalkoztak hasonló összehasonlításokkal dinamikus rendszerek esetén. Cikkemben arra a kérdésre keresem a választ, hogy van-e lényeges különbség teljesítményben ezen széles körben használt alkalmazások teljesítményei között.

1.1. Numerikus számításokhoz használható programcsomagokról és nyelvekről röviden

1.1.1. Fortran

Az IBM által 1950-ben fejlesztett *Fortran* egy általános célú programozási nyelv, melyet napjainkban is használnak és fejlesztik is. Több programozási paradigmát is támogat, köztük az objektum orientáltságot, a procedurális és a generikus programozást is.

1.1.2. Maple

A *Maple* a *Maplesoft* által fejlesztett matematikai szoftvercsomag, mely egy hatékony matematikai motort kombinál egy könnyen használható felhasználói felülettel, ezáltal a használata egyszerű és gyors. A felhasználói felület mellett parancssoros módban is képes működni, a saját nyelvét használva, melyhez írhatunk scripteket is.

1.1.3. Matlab

A *Matlab* a *MathWorks* programcsomagja és programozási nyelve is egyben, egy nagyon elterjedt megoldás és számos helyen használják: numerikus számítások, robotika, adat elemzés, gépi tanulás, jel feldolgozás, kockázat elemzés, és irányító rendszerek fejlesztéséhez. Elsődlegesen mátrix műveletekre és numerikus számításokra készült, ezáltal nagyon jól kezeli a mátrixokat és vektorokat. Képes több szálon dolgozni és támogatja a procedurális és objektum orientált programozási paradigmákat is.

1.1.4. Python

A Python egy nagyon magas szintű, platform független, általános programozási nyelv. Nagyon nagy előnye a könnyű tanulhatósága és olvasható szintaktikája, sokoldalúsága és bővíthetősége. Ezen tulajdonságai miatt rengeteg területen alkalmazzák. A *NumPy, SimPy, SciPy* és *Matplotlib* csomagokat használva hatékony alternatív eszközt nyújt a numerikus számításokhoz.

1.1.5. GNU Octave

A GNU Octave egy magas szintű nyelv, mely elsődlegesen numerikus számításokhoz lett kitalálva, egy kényelmes parancssoros felületet biztosítva, lineáris és nem lineáris problémák numerikus megoldásához és más numerikus kísérletekhez, egy olyan nyelvet használva, amely nagymértékben kompatibilis a Matlab-bal. Az Octave kiterjedt eszközkészletet nyújt a leggyakoribb numerikus problémák megoldására, könnyedén testre szabható és kiterjeszthető a felhasználó által írt metódusokkal, melyek íródhatnak az Octave saját nyelvén, vagy dinamikusan betölthető modulokban, melyek más nyelven íródtak, mint C vagy C++ esetleg Fortran, vagy valami egyéb nyelven.

2. Használt környezet és a mérések menete

A méréseket az alábbi környezetben végeztem el:

2.1. Szoftveres környezet

Használt csomagok:

- GNU Octave 5.2.0
- Matlab R2020a (Trial)
- Python 3.7.4
  - Numpy 1.16.5
  - SciPy 1.3.1

- használt fejlesztő környezetek:
  - PyCharm Professional 2019.2
  - Matlab saját felhasználói felülete (Matlab GUI)
  - Octave saját felhasználói felülete (Octave GUI)

Operációs rendszer: Windows 10 Pro v.2004 64 bit

2.2. Hardveres környezet

Használt hardver:

- Processzor: Intel i7 7700HQ
- Memória: 8GB

2.3. Mérések menete

A méréseket az alábbi módon végeztem el:

- Minden teszt előtt általánosan leírom az adott problémát.
- Csak az adott művelet elvégzésének idejét mértem le.
- Az adott programot 200 alkalommal futtattam le.
- A kapott eredmények átlagát vizsgálom és tüntetem fel a cikkben.
- Az adatok melyeken a műveleteket végeztem véletlenszerűen generáltak, ahol nem ott fel van tüntetve, a generálás ideje nincs benne a művelet elvégzéséhez szükséges mért időben.
- A generált véletlenszerű adatok lebegőpontos számok, ahol nem ott feltüntetem azt, hogy nem.
- Ahol egyéb kitételek is vannak az adatok ezeknek megfelelően lettek le generálva, szintén véletlenszerűen *(pl.: mátrix pozitív definitása)*.
- A mérések esetén igyekeztem a nyelv vagy programcsomag által nyújtott lehetőségeket használni, tehát a nyelv által nyújtott metódusokat mértem le, ahol egy algoritmus általam elkészített implementációja szerepel, azt fel fogom tüntetni.

3. Egyszerű mátrix műveletek

Először is nézzük meg, mi az a mátrix. A mátrix a matematikában valamilyen mennyiségek téglalap alakú elrendezésben. Ezek a mennyiségek lehetnek egyszerű számok, de akár függvények is, vagy esetleg más mátrixok is. Mátrixokkal elvégezhető műveletek: transzponálás, mátrixok összeadása, mátrix szorzás, skalárral való szorzás. Az első két tesztben a mátrixszorzást és -összeadást fogom megvizsgálni. Az (1) képletben egy példa mátrix látható.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \tag{1}$$

3.1. Mátrix összeadás

Mátrixok összeadása során (2) a két mátrix megfelelő indexű elemeit adjuk össze. Ezáltal a csak is két azonos dimenziójú mátrix adható össze.

$$A_{nm} + B_{nm} = C_{nm} = \begin{bmatrix} a_{11} & + & b_{11} & \cdots & \cdots & a_{1m} & + & b_{1m} \\ & \vdots & & & & & \vdots & \\ & \vdots & & \ddots & & & \vdots & \\ & \vdots & & & & & \vdots & \\ a_{n1} & + & b_{n1} & \cdots & \cdots & a_{nm} & + & b_{nm} \end{bmatrix} \quad (2)$$

Az első tesztben nagy méretű mátrixok összeadásának a sebességét mértem. A mátrixok méretei az alábbiak voltak 1024 × 1024, 2048 × 2048, 4096 × 4096, 8192 × 8192. Az eredményeket az alábbi táblázat foglalja össze:

1. táblázat. *Mátrixok összeadásának ideje (másodperc)*

Mátrix mérete	**1024 × 1024**	**2048 × 2048**	**4096 × 4096**	**8192 × 8192**
Python	0,003018	0,01045	0,03987	0,17591
Matlab	0,003108	0,01064	0,04156	0,15829
Octave	0,003651	0,01357	0,05235	0,21306

A táblázatból látható, hogy mind a három nyelv közel azonos idővel végzi el a műveleteket, viszont ahogy növeljük a méretet, a Matlab és a Python kezd gyorsabb lenni, mint az Octave. A Python és az Matlab nagyon hasonló időket produkálnak.

3.2. Mátrix szorzás

Két mátrixok akkor szorozhatunk össze, ha a bal oldali mátrix oszlopainak a száma megegyezik a jobb oldali mátrix sorainak a számával, és a kapott mátrixunknak annyi sora lesz, mint a szorzás bal oldalán álló mátrixnak, és annyi oszlopa, amennyi a szorzás jobb oldalán lévő mátrixnak. A (3) képletben a mátrix szorzás általános alakja látható.

$$A_{mn} * B_{nk} = C_{mk} \quad (3)$$

Az egyes indexeken lévő értékeket pedig a következő módon számolhatjuk ki, az eredmény mátrix egy pontjában található érték számolása:

$$C_{ij} = \sum_{p=1}^{n} A_{ip} * B_{pj} \quad (i \in 1,2,3 \ \ldots \ m, \qquad j \in 1,2,3 \ \ldots \ k, \qquad A \in M_{mn} \quad B \in M_{nk} \quad C \in M_{mk}) \quad (4)$$

A következő mérésben a feladat hasonló volt az előzőhöz: a mátrixok méretei megegyeznek, az eltérés csupán annyi, hogy nem összeadjuk, hanem összeszorozzuk a mátrixokat, mivel a mátrixok négyzetes alakúak így a szorzás feltételének eleget tesznek.

A Python lett ebben a tesztben a leggyorsabb, de csak egy hajszálnyival maradt le a Matlab tőle. Az Octave most is lassabb volt, mint a másik két társa. A Python a végén már jelentősnek nevezhető, majdnem 1 másodperces előnye a Matlab-hoz és több, mint 2 másodperces előnye az Octave-hoz viszonyítva, valamint a tendenciát figyelembe véve, azt jelentheti, hogy a méretet tovább növelve a Python előnye tovább nőhet és egyértelmű előnyt jelenthet a másik két vizsgált nyelvhez képest.

2. táblázat. *Mátrixok összeszorzásának ideje (másodperc)*

Mátrix mérete	**1024 × 1024**	**2048 × 2048**	**4096 × 4096**	**8192 × 8192**
Python	0,0271955	0,138602	1,276361	9,0625243
Matlab	0,0244138	0,147037	1,312633	10,535636
Octave	0,0238087	0,154890	1,553110	11,695740

3.3. LU felbontás

Egy mátrix LU felbontása a következőt jelenti: a mátrixot felbontjuk, úgy egy alsó (**L**ower) és felső (**U**pper) háromszögmátrixra, úgy, hogy ezek szorzatának eredménye az eredeti mátrixunk legyen. (5) Az LU felbontás általános alakja és 3×3-as példa az LU felbontásra.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \text{ x } \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{5}$$

$$A = LU$$

3. táblázat. *LU felbontás ideje (másodperc)*

Mátrix mérete	**1024 × 1024**	**2048 × 2048**	**4096 × 4096**	**8192 × 8192**
Python	0,032241	0,167806	0,817745	5,221102
Matlab	0,020194	0,100511	0,507782	4,512361
Octave	0,050461	0,244357	1,313597	8,335829

Ez a teszt már kicsit látványosabb különbségeket produkált. Az LU-felbontásos tesztet a Matlab nyerte 0,7 másodperces előnnyel a legnagyobb mátrix esetében, a Python előtt. Ezúttal az Octave jelentősen lassabb volt mind a két társánál.

3.4. Cholesky-felbontás

Cholesky-felbontás esetén a mátrixot felbontjuk egy olyan alsó (vagy felső) háromszög mátrixra, melyet, ha összeszorzunk az adjungáltjával, akkor annak a szorzásnak az eredménye maga a mátrix, amelyből kiindultunk. A Cholesky-felbontás csakis szimmetrikus, pozitív-definit mátrixok esetén használható. (6) A lineáris egyenletrendszer általános alakja.

$$A = LL^* \tag{6}$$

A mátrixok a tesztnél szimmetrikus és pozitív definit mátrixok voltak. A Matlab és az Octave a felső míg a Python által használt NumPy csomagban található metódus az alsó háromszög mátrixot számolta ki, de azt feltételeztem, hogy ez a sebességre nem lehet számottevő hatással. Ebben a tesztben is a Matlab

volt a leggyorsabb, amit szintén a Python követett, hasonlóan az előző LU felbontásos tesztesetnél. Az Octave itt is a leglassabb volt.

4. táblázat. *Cholesky felbontás ideje (másodperc)*

Mátrix mérete	**1024 × 1024**	**2048 × 2048**	**4096 × 4096**	**8192 × 8192**
Python	0,028166	0,190326	0,725186	3,444141
Matlab	0,005782	0,036046	0,283129	1,851955
Octave	0,019151	0,086200	0,572526	5,492876

3.5. Lineáris egyenletrendszerek megoldása

A lineáris egyenletrendszer olyan többismeretlenes egyenletrendszer, ahol minden ismeretlen első hatványon szerepel. Általános alakja:

$$Ax = b, \tag{7}$$

ahol A az együttható mátrix b az eredmény vektor és x az ismeretlenek vektora. A feladatunk, hogy az x_i ($i \in 1,2,3 \ldots$ n) értékeket meghatározzuk.

A következő teszt eltérő a többitől, mivel ezt a tesztet lebegőpontos számok helyett egész számokkal végeztem. Különbség még a mátrixok méretében is van, ugyanis óriási futási időket tapasztaltam a Python esetében, amire a következő tesztben bővebben is ki fogok térni. A Python egész számokat lassan generál, ezért lecsökkentettem az együttható mátrixok méretét rendre 512 × 512, 768 × 768, 1024 × 1024, 1280 × 1280 eleműre.

5. táblázat. *Lineáris egyenletrendszerek megoldása (másodperc)*

Mátrix mérete	**512 × 512**	**768 × 768**	**1024 × 1024**	**1280 x 1280**
Python	0,002711	0,006479	0,019265	0,029832
Matlab	0,005896	0,016951	0,049608	0,081372
Octave	0,017313	0,043528	0,090767	0,148583

Maga az egyenletrendszerek megoldása nagyon gyorsan végbe ment minden esetben, a leggyorsabban a Python oldotta meg őt a Matlab követi végül jön az Octave. A képet viszont árnyalja, hogy mégis a Python dolgozott a legtovább a lassú integer generálásának köszönhetően.

3.6. Randomszám-generálás

Az előző tesztben tapasztaltak miatt, gondoltam mégis jó lenne ebből a szempontból is összevetni a nyelveket, itt külön táblázatba fogom szedni a Pythont, hiszen a másik két nyelv, majdhogynem külön ligában játszik egész számok esetén. A tesztben a lineáris egyenletrendszereknél bevezetett méreteket fogom használni. Kezdjük is el a Python eredményeivel. A tesztnél nem csak az egyszerű random

számokból álló mátrixot generáltam, hanem pozitív definit mátrixokat, úgyhogy a mért időben még benne van még 4 másik művelet is.

6. táblázat. *Randomszám-generálás: Python (másodperc)*

Mátrix mérete	**512 × 512**	**768 × 768**	**1024 × 1024**	**1280 × 1280**
Python (egész)	0,218987	0,723971	1,737466	3,37302
Python (lebegőpontos)	0,002108	0,005148	0,008674	0,01289

Látható, hogy a lebegőpontos szám generálási idejének többszöröse az egész generálási ideje és a méret kis mértékű növelése is nagyon meg tudja növelni az időket. Ennek az okának meghatározásához bele kell nézni a generáló metódus forrás kódjába és plusz egy kis utána olvasás után azt szűrtem le, hogy a metódus sok ellenőrzést végez annak érdekében, hogy minden esetben hiba nélkül le tudjon futni, de ennek a cikknek nem feladata a pontos okot kideríteni, úgyhogy a továbbiakban nem foglalkozomk a problémával. Menjünk is tovább a Matlab és az Octave eredményeivel.

7. táblázat. *Random szám generálás: Matlab és Octave (másodperc)*

Mátrix mérete	**512 × 512**	**768 × 768**	**1024 × 1024**	**1280 × 1280**
Matlab (egész)	0,005340	0,012314	0,021890	0,032335
Matlab (lebegőpontos)	0,005007	0,010509	0,019010	0,028215
Octave (egész)	0,014099	0,032247	0,057917	0,088373
Octave (lebegőpontos)	0,006984	0,016511	0,030821	0,045790

Mint látható, a Matlabnak és az Octave-nak nem okoz gondot a random egész szám generálás, bár számomra érdekes módon a lebegőpontos számok generálása itt is gyorsabb. Meg kell jegyezni, a Matlab itt is gyorsabb az Octave-nál.

3.7. Sajátérték, sajátvektor

A mátrixok sajátértékéhez és sajátvektorához szükségünk lehet a komplex számok halmazára is. Egy ilyen komplex számokból álló mátrixot a valós számokból állóhoz hasonlóan $\mathbb{C}^{mxn} - el$ jelöljük ($R^{mxn} \subset C^{mxn}$). Nézzük először a sajátvektor és a sajátérték definícióját!

Legyen $A \in \mathbb{C}^{nxn}$ tetszőleges mátrix. $A, \lambda \in \mathbb{C}$ számot az A mátrix sajátértékének és az $x \in \mathbb{C}^n$ $(x \neq 0)$ vektort pedig λ sajátértékhez tartozó sajátvektornak nevezzük, ha

$$Ax = \lambda x \tag{8}$$

Fontos, hogy egy mátrix sajátértékeinek összeségét a mátrix spektrumának nevezzük és a spektrumból a mátrix fontos tulajdonságait lehet kiolvasni például az alábbiakat.

- Egy négyzetes mátrix akkor és csak akkor nemszinguláris, ha egyik sajátértéke sem nulla.
- Egy mátrix akkor és csak akkor pozitív definit, ha minden sajátértéke pozitív.

A sajátértékeket úgy kaphatjuk meg, ha megoldjuk a karakterisztikus egyenletet, amely a következő:

$$\phi(\lambda) = \det(A - \lambda i) = 0 \quad (9)$$

A mátrixok mérete ebben a teszteb 512 x 512, 1024 x 1024, 1536 x 1536, 2048 x 2048.

8. táblázat. *Sajátérték és sajátvektor számolás (másodperc)*

Mátrix mérete	**512 × 512**	**1024 × 102**	**1536 × 1536**	**2048 × 2048**
Python	0,242369	0,969306	2,606619	5,762008
Matlab	0,232779	0,946277	2,553774	5,725959
Octave	0,554634	1,886882	4,510491	9,013209

A Matlab ekkor egy hangyányival gyorsabb csak, mint a Python, míg az Octave jelentősebb lassabb volt tőlük.

3.8. Interpoláció

Az interpoláció egy közelítő módszer a matematikában, mellyel egy függvény nem ismert értékeit határozzuk meg. A lényege matematikailag az, hogy adott az $f(x)$ függvény és adottak az $x_1 \dots x_n$ pontokban felvett értékei az $[a = x_1, b = x_n]$ intervallumban és magát az $f(x)$ függvényt szeretnénk közelíteni egy könnyen számolható $h(x)$ függvénnyel, melyre fennáll, hogy $f(x_i) = h(x_i)$.

A mostani tesztben az alábbi függvényt interpoláltam a $[0{,}30]$ intervallumban: $(x^3)/4$ és az interpolációkban meghatározandó pontok száma 100 000, 200 000, 300 000 és 400000 darab pont a már fent említett intervallumon. Az adott nyelvek a saját interpolációs eljárásukat használták, 30 darab függvényérték volt megadva az intervallumban (0-tól 30-ig az egész számokon felvett függvény érték).

9. táblázat. *Interpoláció saját eljárással (másodperc)*

Pontok száma	**100 000**	**200 000**	**300 000**	**400 000**
Python	0,001299	0,002884	0,004422	0,006021
Matlab	0,000161	0,000060	0,000044	0,000042
Octave	0,002694	0,002028	0,001994	0,001987

Az eredmény kissé meglepő a Matlab és az Octave gyorsult a pontok számának növelésével, míg a Python ahogy elvárható volt lassult, a Matlab volt magasan a leggyorsabb, de a Python és az Octave is nagyon gyors volt.

3.9. Spline interpoláció

A Spline esetén szintén adott az intervallum, $f(x)$ függvény és az y_i függvény értékeink és a feladat az, hogy megkeressük azt az $S(x)$ függvényt, mely teljesíti az alábbi feltételeket:

$$\begin{aligned} &(\mathbf{1}.)\ S(x) = S_i(x), && x \in [x_i, x_{i+1}], \\ &(\mathbf{2}.)\ S(x_i) = y_i, && (i = 1, \dots, n), \\ &(\mathbf{3}.)\ S_i(x_{i+1}) = S_{i+1}(x_{i+1}), && (i = 1, \dots, n-2) \end{aligned} \tag{10}$$

Az első feltétel megfogalmazza, hogy szakaszokból áll a függvényünk, a második megmondja, hogy valóban interpoláló függvény az $S(x)$ függvényünk, és a harmadikkal a folytonosság van definiálva az az $[a, b]$ intervallumon.

10. táblázat. *Spline interpoláció (másodperc)*

Pontok száma	100 000	200 000	300 000	400 000
Python:	0,004153	0,007814	0,011618	0,015247
Matlab	0,000145	0,000047	0,000039	0,000038
Octave	0,003727	0,002801	0,002788	0,002783

A tesztben a Matlab volt ismét a leggyorsabb, de szintén nagyon gyors volt az összes környezet. A Python és az Octave lassultak a teszt során az előző méréshez képest, míg a Matlab gyorsult.

3.10. Numerikus deriválás

A numerikus deriválás alapfeladata a következő, adott az $f : D(\subseteq \mathbb{R}) \to \mathbb{R}$ függvény, amely analitikusan ismeretlen vagy nehezen számolható, esetleg csak diszkrét pontokban ismert, és az f függvényünk deriváltját szeretnénk közelíteni adott pontokban.

Nekifuthatunk úgy a feladatnak, hogy az f függvényünkhöz készítünk egy interpolációt és az interpolációnk k-adik deriváltja fogja közelíteni az f függvényünk k-adik deriváltjához.

Ha $f \in C^2\,[a, b]$, akkor felírhatjuk rá a másodfokú Taylor-polinomot, amelyből kifejezhető az alábbi összefüggés, (11) numerikus első derivált:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \tag{11}$$

a képlet hibája $O(h)$.

A harmadfokú Taylor-polinommal még pontosabb közelítést kaphatunk, (12) numerikus első derivált harmadfokú Taylor-polinomból kifejezve:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{12}$$

A másodfokú deriváltra az alábbi két összefüggést alkalmazhatjuk: (13) numerikus másodfokú derivált, (14) centrális differencia formula:

$$f'^{(x)} \approx \frac{f(x) - 2f(x+h) + f(x+2h)}{h^2}, \tag{13}$$

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \tag{14}$$

Mindkét képlet hibája $O(h^2)$. A tesztben az alábbi függvények deriváltjait számoltattam ki a fenti módszerek általam elkészített implementációjával:

- $f(x) = \cos(x^3) - 1$
- $g(x) = x^2 + x^3 + 3$
- $h(x) = \sin(x)^2 + 3x + 5 + \tan(x + 2)$
- $j(x) = \sin(x^{20} + x^{30} + 3)^4 + 31x + 6x^{12}$

11. táblázat. *Numerikus deriválás (másodperc)*

Függvényt	$f(x)$	$g(x)$	$h(x)$	$j(x)$
Python első derivált	0,001664	0,002764	0,003953	0,005021
Python második derivált	0,002276	0,003721	0,005686	0,007285
Matlab első derivált	0,004235	0,003542	0,003343	0,004005
Matlab második derivált	0,004294	0,003545	0,003374	0,005346
Octave első derivált	0,076524	0,057344	0,097503	0,064113
Octave második derivált	0,071820	0,056799	0,089114	0,079865

A teszt eredménye a következő: a Matlab volt a leggyorsabb mind az első, mind a második derivált kiszámításánál, őt követi a Python és jön az Octave harmadikként.

3.11. Numerikus integrálás

Integrálás során a Newton–Leibnitz-formulából indulhatunk ki. Ha az adott f függvényünk Reimann-integrálható $[a, b]$-n, és itt létezik primitív függvénye, akkor

$$I = \int_a^b f(x)dx = F(b) - F(a). \tag{15}$$

Ez a módszer viszont csak akkor alkalmazható, ha f-nek létezik primitív függvénye, ha nincs, akkor viszont numerikus integrálást kell alkalmaznunk. Numerikus integrálás során az f függvényt közelítjük egy p interpolációs polinommal és ezáltal a határozott integrál értékét közelítjük, általános formában:

$$I = \int_a^b f(x)w(x)dx \approx \int_a^b p\,(x)w(x)dx, \tag{16}$$

ahol $w(x) \geq 0$ pedig egy tetszőleges súlyfüggvény.

Numerikus integrálást rengeteg algoritmussal le lehet írni, vannak egyszerűbbek (például: trapéz-, téglalap formulák) és bonyolultabbak is. Ezek egy része visszavezethető a nyílt vagy zárt Newton–Cotes-formulákra. A teszt során a numerikus deriválásnál megismert függvényeket $(f(x), g(x), h(x), j(x))$ fogom a $[0{,}100]$ – intervallumon integrálni az általam implementált összetett Simpson formula segítségével.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} \frac{x_{i+1} - x_i}{6}\left[f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right] \tag{17}$$

12. táblázat. *Numerikus deriválás (másodperc)*

Függvény	$f(x)$	$g(x)$	$h(x)$	$j(x)$
Python	0,003258	0,004483	0,006469	0,007976
Matlab	0,000710	0,000544	0,000290	0,002493
Octave	0,074990	0,058867	0,093007	0,081972

A Matlab volt a leggyorsabb, a Python nem sokkal követi, az Octave-nak sokszorosa kellett a Matlab vagy a Python idejének, egy-egy integrál kiszámításához.

4. Konklúzió

Összességében a tesztek eredménye az lett, hogy a legtöbb esetben a Matlab lett a leggyorsabb, amit a Python követ, míg az Octave csak a harmadik. Most szeretném összegezni a nyelvek előnyeit, hátrányait. Látszik, hogy a Matlab volt a leggyorsabb, ez kiegészül azzal, hogy a nyelv könnyen tanulható és sok mindent támogat, mint például a GPU-gyorsítást egyszerűen, amit nem használtam a teszteknél, de a segítségével még gyorsabb tudna lenni. Hátulütője a dolognak, hogy a programcsomag nem ingyenes. A Python és az Octave ingyenes, hiszen a Python egy általános magasszintű nyelv, ami nem pont ilyen célra lett kitalálva, de bizonyította már többször, hogy akár erre is használható. A GPU-gyorsítást ez is támogatja, bár igaz, korlátozásokkal. A Python mint nyelv a könnyebben tanulhatóak közé tartozik, de jelentősen eltér, így is a Matlab vagy akár más specifikusan ilyen célra tervezett nyelvektől. Az Octave volt a leglassabb, miért választaná ezt valaki? A válasz egyszerű, az Octave ingyenes és a Matlab nyelvét használja a kódok, melyek lefutnak a Matlab környezet alatt, nagy eséllyel lefutnak Octave alatt is. A tesztben a Matlab és az Octave kódok nagyrésze ugyanaz volt egy-két esetben volt csak minimális eltérés. Ellenben míg a Matlab-hoz kapunk támogatást az Octave esetén szükség esetén magunknak kell megtalálni a segítséget. A cikk konklúziója a következő, ha szükségünk van egy gyors könnyű nyelvre, amihez kapunk támogatást is és akár fizetnénk is érte, a Matlab egy jó választás. Abban az esetben viszont, ha ismerjük a Python-t, akkor a NumPy és SciPy csomagokkal nagyon jól használható ilyen célra is. Az Octave arra a legjobb, ha nem szeretnénk fizetni, vagy kisebb projekteket szeretnénk végre hajtani, vagy pedig olyan kódot szeretnénk írni, ami később kompatibilis a Matlabbal.

Irodalom

[1] Ford, W.: *Numerical linear algebra with applications: Using MATLAB*, Academic Press, London, 2014.

[2] Walt, Stéfan van der, Colbert, S. C., Varoquaux, G.: *The NumPy array: a structure for efficient numerical computation*, Computing in Science & Engineering 2011, 13(2): 22-30. **https://doi.org/10.1109/MCSE.2011.37**

[3] Hansen, J. S.: *GNU Octave: Beginner's Guide: Become a proficient octave user by learning this high-level scientific numerical tool from the ground up*, Packt Publishing Ltd., Birmingham, 2011.

[4] Shampine, L. F., Allen, R. C., Pruess, S.: *Fundamentals of numerical computing*, New York: Wiley, 1997.

[5] Davis, P. J., Rabinowitz, P.: *Methods of numerical integration*, Courier Corporation, North Chelmsford, 2007.

[6] Kincaid, D., Kincaid, D. R., Cheney, E. W.: *Numerical analysis: mathematics of scientific computing*, American Mathematical Soc., Rhode Island, 2009.

[7] Galántai, A.: *Alkalmazott lineáris algebra*, Miskolc: Miskolci Egyetemi Kiadó, 1996.

[8] Trefethen, L. N., Bau, III. D.: *Numerical linear algebra*, Siam, 1997. **https://doi.org/10.1137/1.9780898719574**

[9] Kahan, W.: *Numerical linear algebra*, Canadian Mathematical Bulletin 1966, 9(5): 757-801. **https://doi.org/10.4153/CMB-1966-083-2**

[10] Forrás kódok a cikkben szereplő mérésekhez: https://github.com/gyulaimark97/Source-code-for-the-article-about-numerical-problems

[11] Guedes, P. F. S., Nepomuceno E. G.: *Some remarks on the performance of Matlab, Python and Octave in simulating dynamical systems*, https://arxiv.org/pdf/1910.06117v1.pdf 2019. **https://doi.org/10.17648/sbai-2019-111188**