

SZAKDOLGOZAT



MISKOLCI EGYETEM

Numerikus számítások hatékonyságának vizsgálata Python környezetben

Készítette:

Gyulai Márk

Programtervező informatikus

Témavezető:

Dr. Nagy Noémi

Konzulens:

Piller Imre

MISKOLC, 2020

MISKOLCI EGYETEM

Gépészszmérnöki és Informatikai Kar

Alkalmazott Matematikai Intézet Tanszék

Szám:

SZAKDOLGOZAT FELADAT

Gyulai Márk (NV4AO2) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Numerikus analízis, Python

A szakdolgozat címe: Numerikus számítások hatékonyságának vizsgálata Python környezetben

A feladat részletezése:

A mérnöki gyakorlatban előforduló numerikus számítások jelentős része közvetlenül elérhető a NumPy nevű Python függvénykönyvtárból. A dolgozat célja, hogy bemutassa ezek használatát, elemesse azok megvalósítási módját, pontosságukat és hatékonyságukat. Mivel a függvénykönyvtárból elérhető számítások jellemzően többféle paraméteressel rendelkeznek, így részletesen meg kell vizsgálni azok működését, össze kell hasonlítani az alternatív számítási módokat.

A dolgozat tematikája célszerűen illeszkedik a Numerikus módszerek című tárgyban előforduló témákhoz, így a későbbiekben az elkészült leírásokat és példákat fel lehet használni a tananyag kiegészítéseként. A számítások áttekinthetősége érdekében a Python nyelvű forráskódokat érdemes Jupyter munkafüzetekbe rendezni. Az eredmények vizualizálásához a Matplotlib függvénykönyvtár kerül felhasználásra.

Témavezető: Dr. Nagy Noémi (egyetemi adjunktus)

Konzulens: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje: 2019. október 10.

.....
szakfelelő

EREDETISÉGI NYILATKOZAT

Alulírott Gyulai Márk; Neptun-kód: NV4A02 a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezenelbüntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Numerikus számítások hatékonyságának vizsgálata Python környezetben* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezclés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, 2020.... év 06.... hó 01.... nap

Gyulai Márk

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

.....

dátum

témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):

.....
.....
.....

konzulens (dátum, aláírás):

.....
.....
.....

3. A szakdolgozat beadható:

.....

dátum

témavezető(k)

4. A szakdolgozat szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....
..... egyéb mellékletet (részletezve)

.....

tartalmaz.

.....

dátum

témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíró neve:

.....

dátum

szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata:

a bíró javaslata:

a szakdolgozat végleges eredménye:

Miskolc,

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. Bevezetés	1
2. Python	2
2.1. A Python programozási nyelv	2
2.2. Felhasználása különböző területeken	2
2.3. Hardveres és szoftveres követelmények	3
2.4. A dolgozatban vizsgált problémák	3
3. Szoftvereszközök numerikus számításokhoz	4
3.1. Alapvető elvárások	4
3.2. Elterjedt matematikai szoftverek	5
3.2.1. Fortran	5
3.2.2. Maple	6
3.2.3. MATLAB	7
3.2.4. Python, NumPy és Matplotlib	9
3.2.5. GNU R	10
3.2.6. Wolfram: Mathematica, Wolfram Alpha	11
3.3. Python eszközkészlet	12
4. A Python környezet összeállítása	14
4.1. Python telepítése	14
4.1.1. pip	15
4.1.2. Anaconda	15
4.1.3. Jupyter munkafüzet	15
5. A Python eszközkészlet hatékonyságának vizsgálata	17
5.1. A hatékonyság típusai	17
5.2. Hibaszámítás	17
5.2.1. Számábrázolás	18
5.2.2. Klasszikus hibaanalízis	19
5.3. Vektor és mátrix műveletek	20
5.3.1. Vektorok	20
5.3.2. Mátrixok	21
5.4. Mátrix felbontások	23
5.4.1. LU felbontás	23
5.4.2. Cholesky felbontás	23
5.5. Lineáris egyenletrendszer	24
5.5.1. Általánosan a lineáris egyenletrendszerkről	24
5.5.2. Lineáris egyenletrendszer megoldása Python-ban	25

5.6.	Interpolációk	30
5.6.1.	Lagrange interpoláció	33
5.6.2.	Spline interpoláció	35
5.6.3.	Hermite interpoláció	36
5.7.	Sajátérték, sajátvektor számítása	37
5.8.	Legkisebb négyzetek módszere	38
5.9.	Numerikus deriválás	39
5.10.	Numerikus integrálás	40
6.	Megjelenítési módok	45
6.1.	Egyszerű grafikonok rajzolása	45
6.2.	Térbeli pontok szemléltetése	48
6.3.	Interaktív widgetek Jupyter-ben	51
7.	Összefoglalás	54
	Irodalomjegyzék	55

1. fejezet

Bevezetés

A szakdolgozat témája a tanult numerikus számítási ejárások Python környezetben való bemutatása. Ilyen numerikus problémák például a mátrix transzponálás mátrix felbontások (LU, Cholesky), lineáris egyenetrendszerek megoldása, interpolációk vagy a numerikus deriválás és integrálás. A numerikus eljárások minden az egzakt megoldást vagy annak egy közelítését fogják adni. Ilyen problémákra már sokféle szoftvert fejlesztettek ki. Például a Matlab, de akár az R környezetet is használhatjuk ilyen célokra. De akkor miért Python? – teheti fel a kérdést a kedves Olvasó. A válaszom pedig az, hogy a dolgozatomban azt fogom bemutatni, hogy, mint sok minden máshoz ehhez is tökéletesen megfelel és könnyen használható ez a nyelv.

Az első fejezetben magáról a Python nyelvről fogok írni általánosan. Bemutatom, hogy miről is szól a szakdolgozat, mi a témája, mit használtam fel benne, mire is van szükség ha ki szeretné próbálni a benne található kódokat.

2. fejezet

Python

2.1. A Python programozási nyelv

A Python egy nagyon magas szintű, platform független, általános célú programozási nyelv, szóval itt nem a kígyófélék egy családjára vagy Monty Pythonra (bár a nevét a Monty Python csapatról kapta) kell gondolni [7]. Egy könnyen elsajátítható programozási nyelv mely jelen pillanatban a 3 legnépszerűbb között van a világon (2020 Február, [3]).

A nyelv interpretált és támogatja a objektum orientált, a funkcionális, az imperatív és a procedurális programozási paradigmákat, valamint a dinamikus típusokat és dinamikus memóriakezelést, szemétgyűjtő algoritmust (használ *garbage collector*). A nyelvet Guido van Rossum holland programozó kezdte el fejleszteni 1989-ben, majd nyilvánosságra hozta 1991-ben. Ez volt a 0.9-es verziószámú. 1994-ben megjelent az 1.0-ás majd 2000-ben a 2.0-ás verzió és csak 2008-ban követte a dolgozatban általam is használt Python 3. A 3-as és a 2-es nem minden esetben kompatibilis egymással és a 2-es utolsó támogatott verzióinak (2.7.x) a támogatása is megszűnt 2020 januárban. A nyelvet napjainkban már a PSF (*Python Software Foundation*) fejleszti és kezeli.

2.2. Felhasználása különböző területeken

A Python nyelvet sok területen felhasználják, többek között

- Webfejlesztésben,
- Tudományos és numerikus számításoknál,
- Oktatásban,
- Asztali grafikus felhasználói felületek (GUI) fejlesztésében,
- Szoftverfejlesztésben,
- Üzleti szoftverek fejlesztésében.

Látható tehát hogy tényleg sok helyen jól használható ez a nyelv.

2.3. Hardveres és szoftveres követelmények

Tulajdonképpen csak egy számítógép, a Python interpreter és egy támogatott operációs rendszer, amit nem lesz nehéz találni hiszen a nyelv minden ma használt és elterjedt operációs rendszert támogat. Az interpreter pedig letölthető a python weboldaláról (www.python.org), illetve egyes operációs rendszerekben alapértelmezés szerint telepítve van (különböző linux disztribúciók).

2.4. A dolgozatban vizsgált problémák

Ahogy már fent említettem, a dolgozat Python-ban fogja bemutatni a tanult numerikus problémákat és módszereket [1], melyekkel egy közelítő megoldást adhatunk egy-egy problémára. Ilyen problémák:

- Hibaszámítás: Abszolút és relatív hiba,
- Vektor és mátrix műveletek,
- Lineáris egyenletrendszer megoldása,
- Interpolációk,
- Sajátérték és sajátvektor.
- Numerikus deriválás,
- Numerikus Integrálás,

Ezeken belül is többek között az alábbi módszerekkel foglalkozni:

- Gauss módszer,
- Gauss-Jordan módszer,
- Legkisebb négyzetek módszere,
- Lagrange interpolációk,
- Spline interpolációk,
- Téglalap módszer,
- Trapéz módszer.

3. fejezet

Szoftvereszközök numerikus számításokhoz

A numerikus számítások lényege, hogy valamilyen algoritmust végrehajtva eljutunk a probléma megoldásához (vagy annak egy közelítéséhez), és ezt egy számítógépes program segítségével tesszük. Ez a program lehet egy előre megírt program csomag része, de akár írhatunk magunk is specifikus problémákra saját programokat. Ezt megtehetjük akár egy olyan eltejedt általános célú programozási nyelven, mint például a **C** vagy a **Java**, de akár használhatunk egy ilyen feladatokra készített nyelvet vagy programcsomagot is. A különbség a probléma megoldásának nehézségében és a hozzá szükséges időben rejlik, hiszen az egyiket erre találták ki, előre definiálva vannak benne szükséges függvények, konstansok, míg a másikban alapjaitól fel kell építeni szinte minden. Szenencsére jótár, numerikus számításokhoz használható szoftvercsomag és nyelv létezik. Ebben a fejezetben ezekről, ezek előnyeiről és hátrányairól fogok írni, illetve a fejezet végén szó esik majd a **Python** eszközkészletéről is.

3.1. Alapvető elvárások

Először is beszéljünk az alapvető elvárásokról, amiket egy ilyen nyelvvel vagy szoftvercsomaggal szemben megkövetelhetünk. Kezdjük a programozhatósággal! Fontos, hogy tudjunk benne programokat írni, hogy speciális feladatokat tudjunk vele megoldani, esetlegesen új módszereket tudjunk implementálni vagy csak egy módszer optimális implementációját szeretnénk elkészíteni, és ne csak olyan problémákra adjon a programcsomag megoldást amit előzetesen annak fejlesztői implementáltak. A programozhatóság ebben a kontextusban elég, ha lekorlátozódik arra, hogy itt matematikai problémák megoldására tudjunk programokat írni, ezért nem feltétlenül kell általános nyelvnek lennie. Erre jó példa a **Matlab**, amit mátrix műveletekhez optimalizáltak vagy az **R**, ami egy statisztikai programcsomag. Ellenpélda a **Python**, mely egy általános célú nyelv de a **NumPy** [4] és **SciPy** [5] csomag segítségével használhatunk numerikus problémák megoldására.

A programozhatóság mellett fontos még a programozási nyelv milyensége. Általában matematikusok, kutatók, mérnökök használják ezeket tudományos vagy mérnöki célokra, illetve diákoknak tanítják a numerikus módszereket ezekkel a programcsomagokkal. Ők itt csak felhasználók, és nem mindig várható el a felhasználóktól, hogy a programozás minden szempontjával és fortélyával tisztában legyenek, így egy viszonylag könnyen tanulható és használható nyelv az, amire szükség van. Ilyen célokra, lehetőleg valami-

lyen egyszerűen megérthető praradigmára építsen, mint az objektum orientáltság vagy a procedurális programozás, vagy néha elég lehet az is, ha csak matematikai formában adjuk meg a megoldandó feladatot. A programozási nyelvek, melyekről itt majd részletesebb is lesz szó, elég könnyen tanulhatóak (kivétel talán a **Fortran** ami egy régi, nehezebben kezelhető nyelv, illetve a **Wolfram Mathematica** eddig említettektől teljesen eltérő szimbolikus nyelve). Ellenben a **WolframAlpha** képes matematikai formában megadott problémákat is megoldani.

Fontos még ezeken felül, hogy a programokat fájlokba tudjuk menteni és ezáltal hordozhatóvá tenni őket, és ez nem csak a programok forráskódjaira legyen igaz, hanem az eredményekre is. Szukséges az, hogy el lehessen menteni a kapott eredményeket, adatokat későbbi felhasználás, publikálás illetve áttanulmányozás érdekében. Az érem másik oldala legalább annyira hasznos és fontos, be is kell tudni olvasni az adatokat különböző adastruktúrákból, különböző formátumú fájlokból, ezért szükséges a népszerűbb adatfájl formátumok támogatása, mint például a CSV, JSON, XML vagy csak egy sima szöveges (`.txt`) fájl.

Az adatok mentése mellett a megjelenítésük is fontos lehet, hiszen az emberek nagy része a grafikus ábrákból egyszerűbben le tudja olvasni az adatokat, ezért szükséges, hogy a program csomag valamilyen módon tudjon ábrákat készíteni. A programokat és módszereket nem csak megírni, menteni és reprezentálni kell, hanem dokumentálni is azokat, ezért szükséges hogy az adott nyelvnek és csomagnak ezt támogatnia kell. Egy nagyon jó példa erre a **Python**, amivel használhatjuk a a **Jupyter** munkafüzetet melyben a dokumentum cellákból épül fel, és a cellákban egyszerre tudunk programozni és dokumentálni is.

A teljesítmény egy nagyon triviális követelmény, de egy nagyon fontos tényező is, bár nagyon nagy adatokhoz szükséges egy erős hardver, de szükséges az is, hogy gyengébb hardveren is futtatható legyen a kód viszonylag jó futási idővel. Erre a problémára egy megoldás még a cloud (felhő) amihez egy viszonylag gyors internet kapcsolat elég. Egyszerűen működik, elküldjük a bemenő adatokat és a programot majd visszakapjuk az eredményt, bár fontos megjegyezni, hogy ez érzékeny adatok esetén ez nem lehet megoldás, hiszen a szerver üzemeltetői láthatják ezeket.

Egy utolsó, de nem elhanyagolható tényező, a programcsomag és az általa használt nyelv elterjedtsége is, hiszen használhatjuk a lehető leggyorsabb nyelvet ha a világban rajtunk kívül csak még egy páran használják, de a világ többi része egy másik elfogadott nyelvet használ, és így nehéz az általunk használt nyelvvel elhelyezkedni, vagy az adatainkat és programjainkat megosztani másokkal, hiszen nem fogják érteni és nem tudják majd futtatni.

3.2. Elterjedt matematikai szoftverek

A következő pontokban áttekintem, hogy melyek az elterjedtebb, matematikai számítások végrehajtásához használt szoftverek.

3.2.1. Fortran

A Fortran egy általános célú programozási nyelv melyet az IBM fejlesztett ki még az 1950-es években [2]. Napjainkban is használatos és még napjainkban is fejlesztik. Aránylag nehezen tanulható nyelv ami több programozási paradigmát is támogat közöttük az objektum orientáltságot, procedurális és generikus programozást is. A Fortran

támogatja a fájlba írást de kimondott különböző fájlformátumú fájlok létrehozását nem, így ha olyan fájlba szeretnénk az adatainkat menteni, nekünk kell gondoskodni a formai követelményekről (például: CSV esetén nekünk kell az elválasztó jelet kiteni az adatok közé).

```

*
C      Factorization of symmetric band matrix with
C      diagonal elements in the first row of the matrix
*
      DO 300 N = 1, NUM_DOF - 1
      M = N - 1
      PIV = 1.0/SYSTEM_FLEXI(1, N)
      DO 150 L = 2, NUM_BANDWIDTH
         IF (SYSTEM_FLEXI(L, N) .EQ. 0.0) GO TO 150
         I = M + L
         IF (I .GT. NUM_DOF) GO TO 150
         C = SYSTEM_FLEXI(L, N)*PIV
         J = 0
         DO 100 K = L, NUM_BANDWIDTH
            J = J + 1
            SYSTEM_FLEXI(J, I) = SYSTEM_FLEXI(J, I) -
               C*SYSTEM_FLEXI(K, N)
   100    CONTINUE
   150    SYSTEM_FLEXI(L, N) = C
   300    CONTINUE

```

3.1. ábra. Fortran forráskód részlet

A 3.1. ábrán egy Fortran forráskód részletet láthatunk. A kód a kötelező jellegű formázás miatt ugyan jól strukturált, viszont az alacsony szintű műveletek (mint például a GO TO) nehezítik a program működésének megértését.

3.2.2. Maple

A **Maple** a Maplesoft által fejlesztett matematikai szoftvercsomag, mely kombinál egy hatékony matematikai motort, mely a számításokat végzi és egy könnyen használható felhasználói felületet, ezáltal a használata egyszerű és gyors is tud lenni [9]. A felhasználói felület mellett parancssoros módban is képes működni, a saját nyelvét használva. Scripteket is írhatunk melyet később megtudunk hívni.

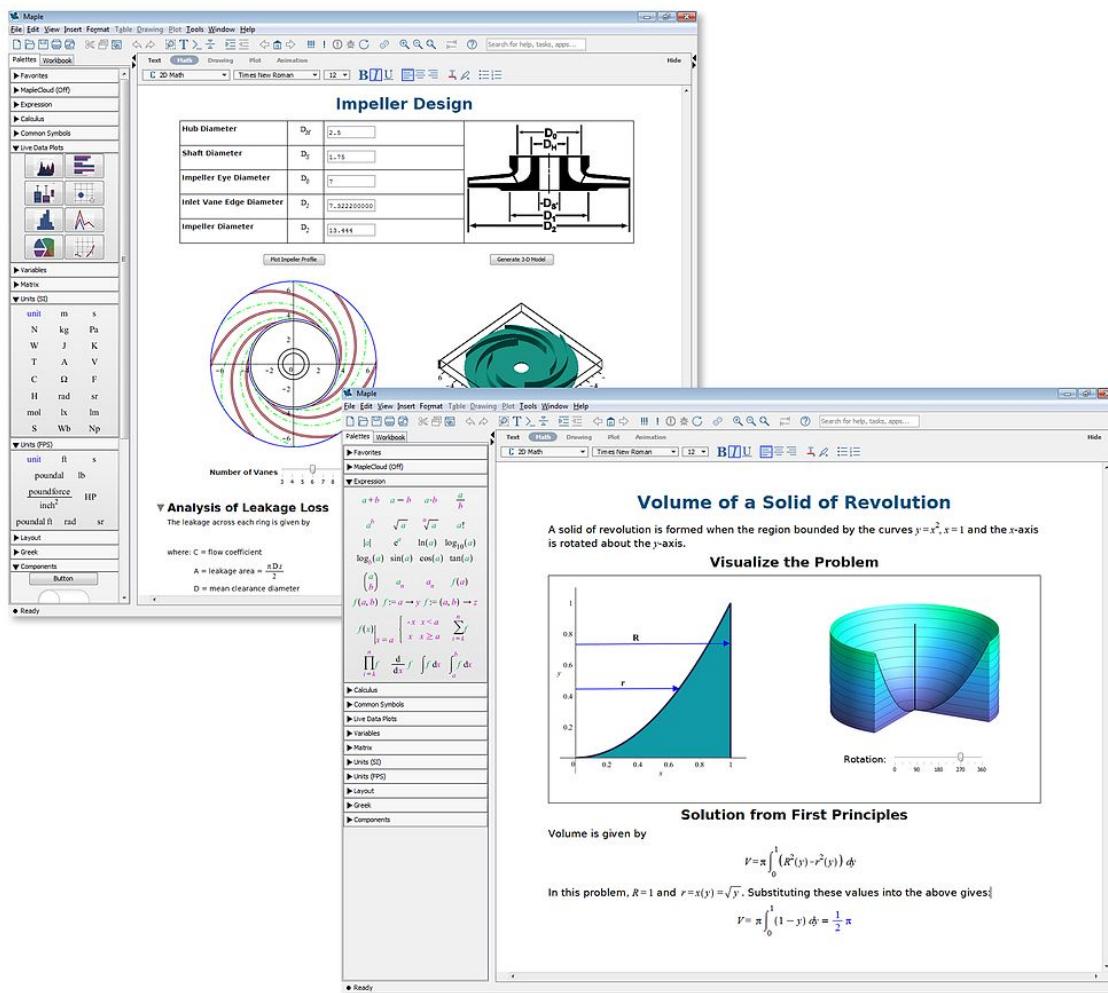


3.2. ábra. A Maple szoftver logója

A 3.2. ábrán láthatjuk a szoftver logóját, a 3.3. ábrán pedig néhány képernyőképet a grafikus felületéről.

A Maple-t mérnöki munkára és emellett rengeteg matematikai területen be lehet venni, például numerikus számításoknál, lineáris algebrában, differenciál egyenleteknél. Nagyszerű az adatok vizualizálásában is. A Maple nem ingyenes, licenszet kell hozzá vásárolni melynek ára elég borsos, de itt is elérhető diákok kedvezmény, mint a legtöbb ilyen szoftvercsomag esetében, de 15 napig ingyenesen kipróbálhatja bárki.

3.2. Elterjedt matematikai szoftverek



3.3. ábra. Maple képernyőképek¹

Egyik előnye, hogy képes kommunikálni bonyos szoftverekkel, mint a Microsoft Excel, néhány CAD szoftver, SQLite adatbázisokkal, és az újabb Matlab verziókkal is, ezzel könnyítve a munkát. Képes az ismertebb adattárolásra használt fájlokba menteni és betölteni is, mint CSV, XLS. Elérhető Microsoft Windowsra, Linuxra és Apple MacOS-re.

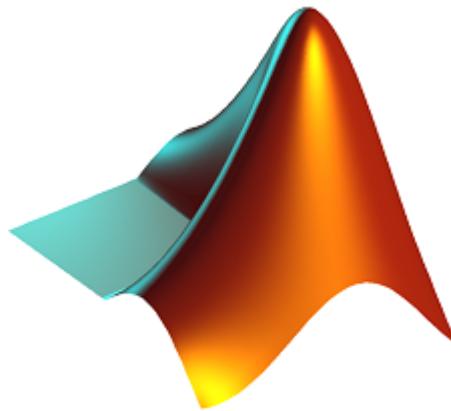
3.2.3. MATLAB

A **Matlab** a MathWorks programcsomagja és programozási nyelve is egyben, egy nagyon elterjedt nyelv a világban és számos helyen használják különböző célokra mint: numerikus számítások, robotika, adat elemzés, gépi tanulás, jel feldolgozás, kockázat elemzés, és irányító rendszerek [13].

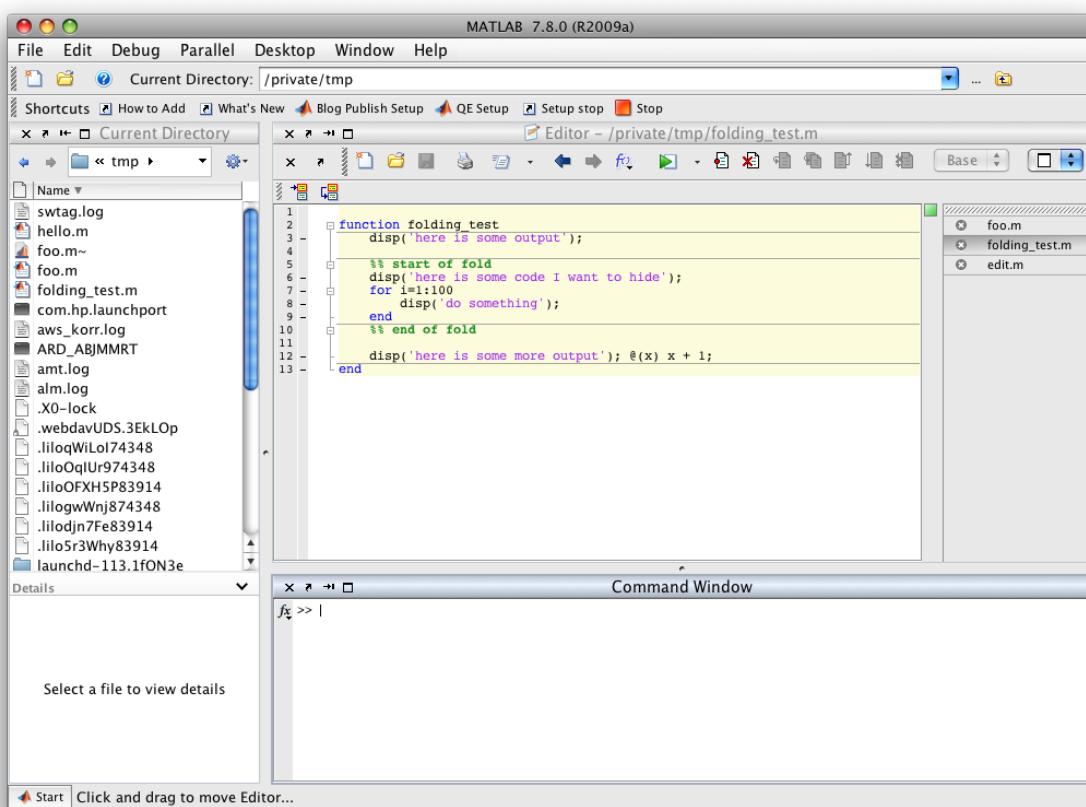
A 3.4. és a 3.5. ábrákon láthatjuk a szoftver logóját és néhány képernyőképet.

Elsődlegesen mátrix műveletekre és numerikus számításokra készült, ezáltal nagyon jól kezelei a mátrixokat és vektorokat. Képes több szálon dolgozni és támogatja a procedurális és objektum orientált programozási paradigmákat is. A Matlab kódokat is

¹forrás: [https://en.wikipedia.org/wiki/Maple_\(software\)](https://en.wikipedia.org/wiki/Maple_(software))



3.4. ábra. A MATLAB szoftver logója



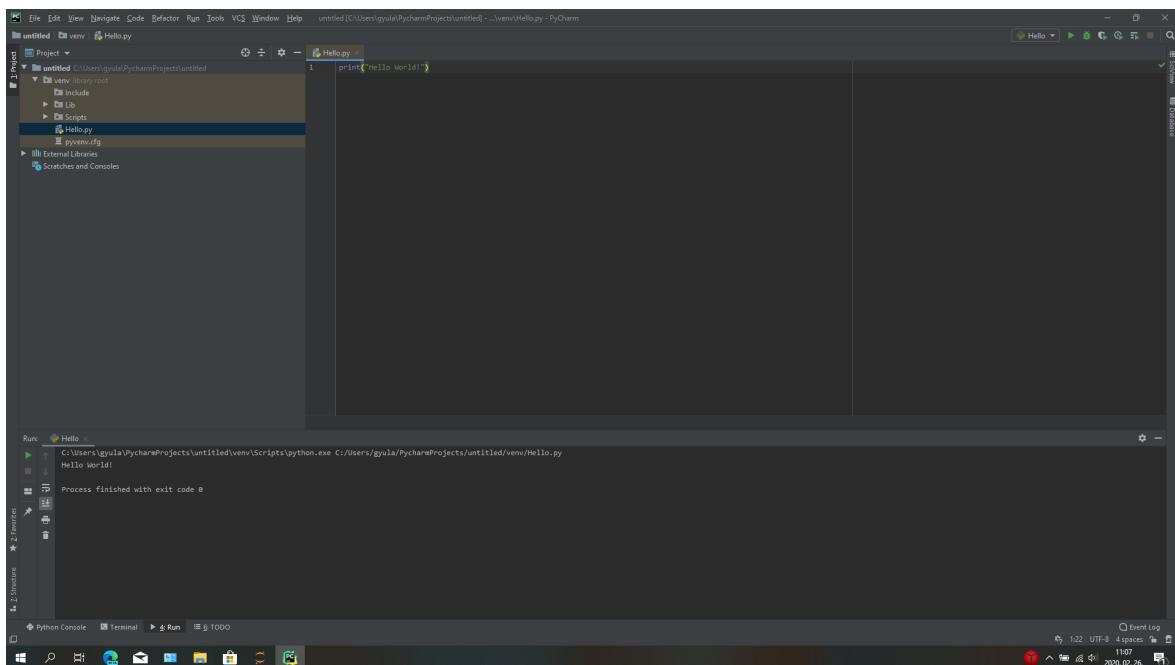
3.5. ábra. Képernyőkép a MATLAB grafikus felületéről²

menthetjük fájlba, ezáltal tudjuk tárolni, publikálni (kiterjesztése: `.m`). Elérhető Microsoft Windows, Linux és Apple MacOS operációsrendszerű gépekre. A Matlab nem egy ingyenes szoftver meg kell vásárolni, de van lehetőségünk ingyenesen kipróbálni illetve van kedvezmény diákok számára. A Matlab is képes importálni és exportálni az adatokat a leginkább használt fájltípusokból.

²forrás: <https://blogs.mathworks.com/community/2009/06/29>



3.6. ábra. A Python programozási nyelv logója [7]



3.7. ábra. Képernyőkép a PyCharm fejlesztőkörnyezetről

3.2.4. Python, NumPy és Matplotlib

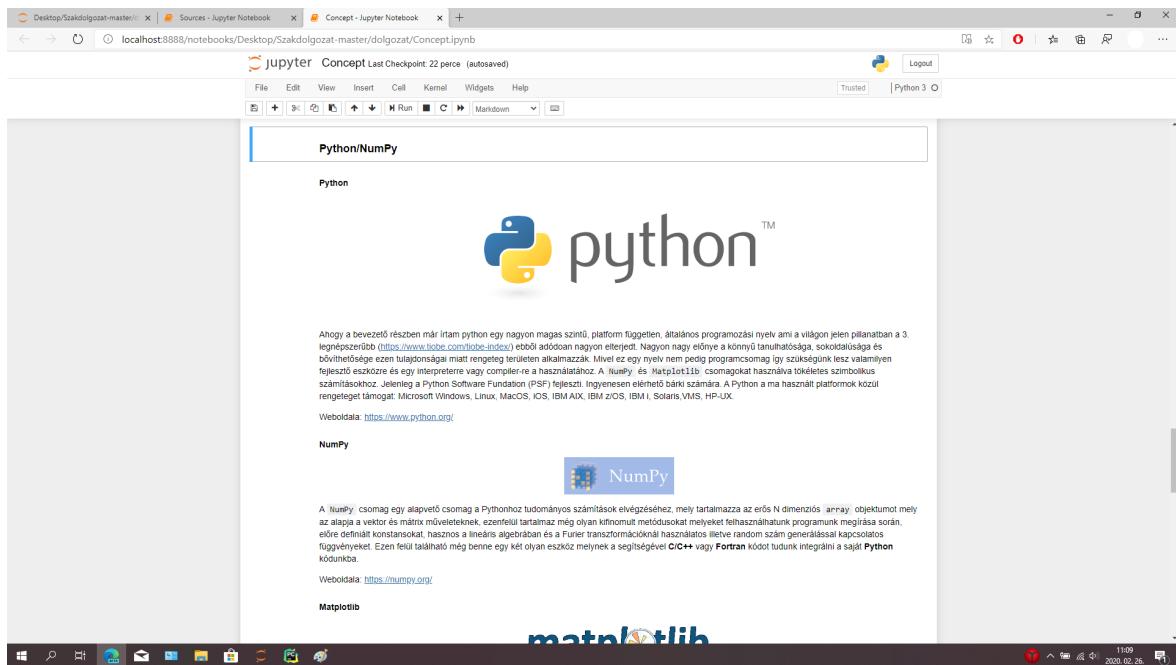
Ahogy a bevezető részben már említettem, a Python egy nagyon magas szintű, platform független, általános programozási nyelv ami a világban jelen pillanatban a 3. legnépszerűbb [3] ebből adódóan nagyon elterjedt. (A programozási nyelv logója a 3.6. ábrán látható.)

Nagyon nagy előnye a könnyű tanulhatósága és olvasható szintaktikája, sokoldalúsága és bővíthetősége. Ezen tulajdonságai miatt rengeteg területen alkalmazzák.

Mivel ez egy nyelv, nem pedig programcsomag így szükségünk lesz valamelyen fejlesztő eszközre és egy interpreterre (vagy compiler-re) a használatához. Fejlesztőeszközök közül a PyCharm [10] (3.7. ábra) és a Jupyter [11] (3.8. ábra) az elterjedtebb eszközök.

A NumPy, SymPy, SciPy és Matplotlib csomagokat használva tökéletes numerikus és szimbólkus számításokhoz. Jelenleg a Python Software Fundation (PSF) fejleszti. Ingyenesen elérhető bárki számára. A Python a ma használt platformok közül renge-

3.2. Elterjedt matematikai szoftverek



3.8. ábra. Képernyőkép egy Jupyter munkafüzetéről

teget támogat: Microsoft Windows, Linux, MacOS, iOS, IBM AIX, IBM z/OS, IBM i, Solaris, VMS, HP-UX, mint ahogy rengeteg fájltípus is támogat, adat export és adat importálás szempontjából is.

A **NumPy** csomag egy alapvető csomag a Python-hoz tudományos számítások elvégzéséhez, mely tartalmazza az erős *N*-dimenziós **array** objektumot, mely az alapja a vektor és mátrix műveleteknek, ezenfelül tartalmaz még olyan kifinomult metódusokat, melyeket felhasználhatunk programunk megírása során, előre definiált konstansokat, hasznos a lineáris algebrában és a Fourier transzformációknál használhatók illetve random szám generálással kapcsolatos függvények. Ezen felül található még benne egy két olyan eszköz melynek a segítségével C/C++ vagy Fortran kódot tudunk integrálni a saját **Python** kódunkba.

A **Matplotlib** csomag segítségével tudjuk vizualizálni az általunk használt adatokat vagy általunk kiszámított adatokat [?]. 2D-s grafikonokat és ábrákat készíthetünk a segítségével. Az ábrákat tudjuk menteni is külön fájlba, a Cairo segítségével akár vektorgrafikusan is.

3.2.5. GNU R

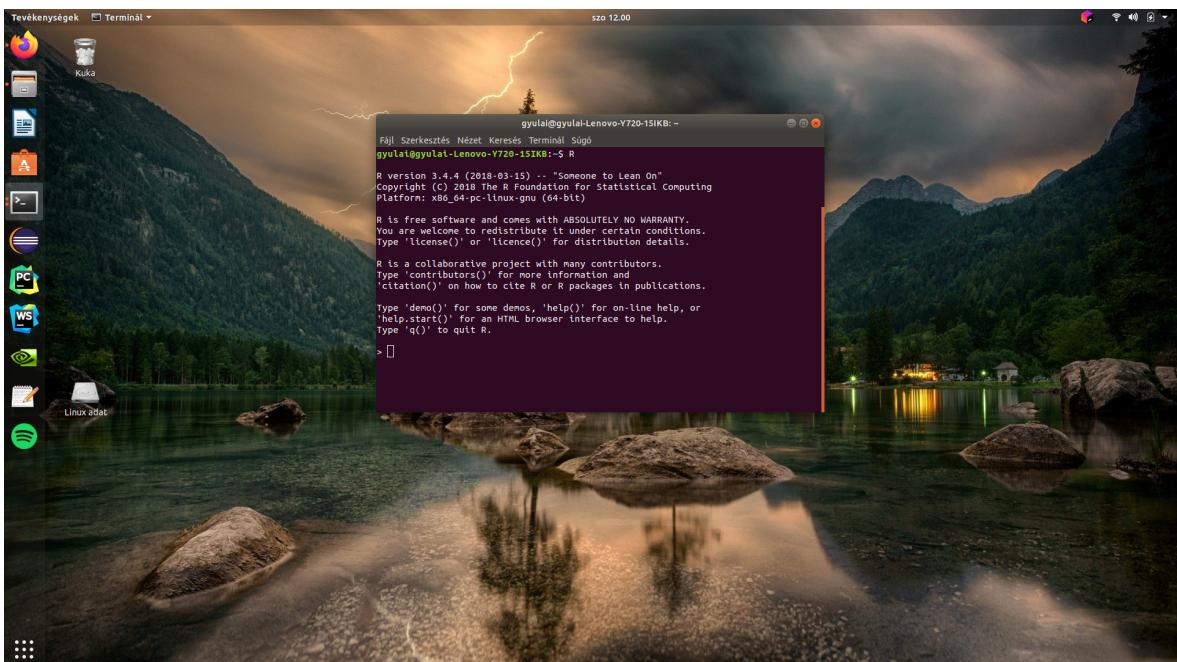
A **GNU R** egy ingyenes szoftvercsomag, melyet elsődlegesen a statisztikai számítások során alkalmaznak [14]. Az **R** nem csak egy programcsomag hanem egy programozási nyelv is egyben.

A programcsomag logóját és egy képernyőképet a program használatáról a 3.9 és a 3.10. ábrákon tekinthetünk meg.

Az R-t, ahogy már említettem statisztikai problémák megoldásánál érdemes használni, de a nyelv egy nagy előnye, hogy nyílt forráskódú és ezáltal nagyon sok csomag készült hozzá és ez kibővíti a képességeit több irányba is. Az R elérhető Microsoft Windowsra, Linuxra, és Apple MacOS-re is. Az R is képes importálni és exportálni az adatokat a leginkább használt fájltípusokból, mint a CSV vagy XLS.



3.9. ábra. A GNU R logója



3.10. ábra. Képernyőkép a GNU R használatáról

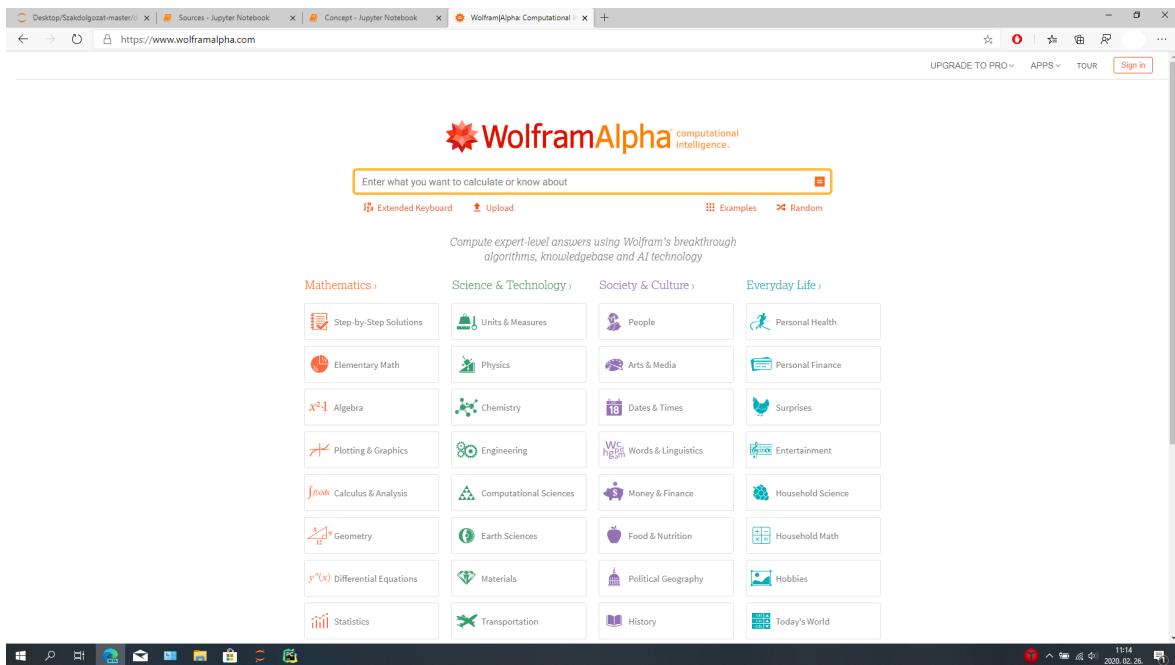
3.2.6. Wolfram: Mathematica, Wolfram Alpha

A Wolfram Alpha nem is igazán egy szoftvercsomag inkább egy probléma megoldó szoftver, melyet a Wolfram fejleszt és mely ötvözi a természetes nyelv feldolgozást, az adatbányászatot és a dinamikus programozást, ezáltal talál a megadott problémára megoldást. (A 3.11. ábrán láthatjuk, hogy hogy néz ki a szoftver használatát biztosító weboldal.)

Az Alpha mögött óriási adatbázis van. Létezik ingyenes megvalósítása mely elérhető a weboldalán, illetve van egy Wolfram Alpha Pro elnevezésű változata is, mely többet nyújt de fizetni kell érte. Az Alpha a fejlesztőknek rengeteg API-t (*Application programming Interface*, magyarul alkalmazásprogramozási interfész) nyújt melynek egy része ingyenes, egy részéért viszont szintén fizetni kell.

A Mathematica szintén a Wolfram által fejlesztett matematikai programcsomag, a Matlab-hoz illetve a Maple-höz hasonlóan rengeteg területet lefed [6]. A Wolfram nevezetű szimbólikus programozási nyelvet használja, mely az eddigiek től nagyon eltérő. A programcsomag szintén nem ingyenes. Elérhető Microsoft Windowsra, Linuxra és Apple MacOS-re, és itt is találunk diákok kedvezményt. Ha nem szeretnénk megvenni a

3.3. Python eszközketeszlet



3.11. ábra. Egy képernyőkép a Wolfram Alpha weboldaláról

teljes csomagot, vagy nincs elég erős számítógépünk hozzá, akkor elérhető a Mathematica Cloud amire elő lehet fizetni egy évre, és csak egy böngésző és internet kapcsolat szükséges hozzá. Az Alpha és a Mathematica is támogatja az adatok importálását és exportálását népszerű formátumú fájlokba.

3.3. Python eszközketeszlet

A szakdolgozatban a Python nyelvet fogom használni és azon belül is nagyrészt a Jupyter munkafüzetben dolgozom. Ahhoz, hogy Python-ban tudjunk dolgozni szükségünk van egy Python értelmezőre. Ez értelmezi a kódot és hajtja azt végre. Több fajta Python interpreter létezik. A legelterjedtebb a CPython mely C-ben és Python-ban íródott, ez a referencia interpreter és ebből indul ki a többi interpreter is. Van interpreter mely engedi hogy Java nyelvű kódot használjunk, és olyan is mely a Python kódot lefordítja C vagy C++ nyelvű kóddá, vagy éppen olyan is ami JavaScript-et vagy Java kódot varázsol a mi kis Python kódunkból.

Mivel Jupyter munkafüzetet használok, így használom az IPython nevezetű shellt is [15]. Az IPython tartalmazza a Jupyter kernelét és támogatja különböző interpreterek használatát. Itt most a referencia CPython interpretert használom majd. Ezen felül támogatja az IPython az interaktív adat vizualizációt, interaktív elemek beépítését a munkafüzetbe, valamint a párhuzamosítást is ha szükség lenne rá, illetve támogat különböző GUI toolkitet is. Az Adatok vizualizációjáért a fent már említett Matplotlib csomag lesz felelős.

Ahhoz, hogy programozzunk valamilyen nyelven, kell valamilyen szöveg szerkeztő vagy IDE és ez a Pythonnal sincs másképp. Pythonhoz elérhető több nagytudású IDE is, mint a Spyder, Jupyter, Pycharm.

Kezdjük a Jupyterrel! Ez egy open-source web-alkalmazás, mely olyan dokumentumok létrehozását, szerkesztését, megosztását teszi lehetővé melyben a programkódok

lefuttathatóak, ezáltal olyan interaktív dokumentumot készíthetünk melyben a programok bemutathatóak és a kapott adatok könnyen vizualizálhatóak. A Jupyter több mint 40 programozási nyelvet támogat.

Használhatjuk akár a Spyder-t is ami eg IDE kifejezetten Pythonhoz. Főleg tudományos célú használatra készült, és mint a Jupyter az IPythont használja. mint minden IDE-t könnyű telepíteni és elkezdeni vele dolgozni. Az Anaconda platformon belül ingyenesen érhető el.

Harmadik fejlesztő eszközünk a Pycharm, amit a JetBrains fejleszt. Létezik egy ingyenes Community Edition és egy megvásárolható Professional Edition. A Professional Edition sokkal több funkciót támogat a Community Edition-nel szemben, például a Jupyter notebookok szerkezetét és megtekintését. A PyCharm azoknak kimondottan előnyös, akik használtak már valamilyen másik JetBrains által fejlesztett IDE-t, hiszen a megszokott kezelő felülettel fognak találkozni.

4. fejezet

A Python környezet összeállítása

Ha szeretnénk Python-ban programozni, ahhoz szükségünk van egy szövegszerkesztőre vagy integrált fejlesztő környezetre a Python értelmezőn túl. A python értelmező ingyenesen elérhető a Python weboldaláról. A népszerűbb GNU/Linux disztribúciók (mint például a Debian, Ubuntu, Linux Mint) már beépítve tartalmazzák az értelmezőt, ezért azt telepíteni sem kell. A csomagok kezeléséhez szükségünk van a `pip` nevű csomagkezelőre is. A `pip` segítségével tudjuk telepíteni lokálisan vagy globálisan a felhasználni kívánt csomagokat is, melyek előre megírt program könyvtárak tulajdonképpen, melyeket fel használhatunk ahhoz, hogy ne mindeneknek kelljen implementálni. Ilyen csomagok a Numpy, Matplotlib, Scipy, vagy a Jupyter notebook is melyeket használunk a későbbiekben.

4.1. Python telepítése

Első lépésként le kell töltenünk a Python értelmezőt a megfelelő operációs rendszerünkre. Windows alatt a telepítőbe konfigurálhatjuk mit telepítünk, és azt is, hogy csak a saját felhasználóknak vagy pedig mindenkinél szeretnénk telepíteni, útóbbitől rendszergazdai jogosultság szükséges.

Unix-szerű rendszereken sem sokkal nehezebb a dolgunk, bár operációs rendszerenként és disztribúciónként eltérések mutatkoznak.

A forráskódiból való telepítéshez először is le kell tölteni a forrást, szintén a Python weboldaláról, majd kicsomagolni, ezek után elnavigálunk egy terminálban a mapába melybe kicsomagoltunk és futtatnunk kell a `configure` nevezetű bash scriptet (`./configure` parancsal tudjuk megtenni). Miután végzett, kiadjuk a `make` parancsot mellyel elkezdődik a fordítási folyamat és felépül az alkalmazás, esetünkben a CPython értelmező. A következő lépés a `make test` mely ellenőrzi, hogy minden rendben ment, majd jöhét a `sudo make install` mellyel telepítjük az értelmezőt (`make install` előtt ott van a `sudo` így láthatjuk hogy ehez szükségünk lesz root jogra (superuser)). (Itt ha a fordítás vagy a telepítés során hibába ütközünk valószínűleg hiányzik valamelyen csomag a számítógépünkről amit telepítenünk kell, ha ez megtörtént, újra le kell futtatni a parancsot amiben a hibát kaptuk és folytatódhat a folyamat tovább.) Néhány további észrevétel az értelmezővel és a verziójával kapcsolatban az alábbi pontokban olvasható.

- Egyes Linux disztribúciók rendelkeznek előre telepített értelmezővel, így felesleges telepíteni azt, ellenőrizzük le először, hogy telepítve van-e. Illetve némely

Linux disztribúció (főleg a Debian alapúak, mint az Ubuntu vagy a Linux mint) különbséget tesz a `python2` és `python3` között. Előbbi terminálból `python` még utóbbi `python3` parancssal érjük el.

- Ez igaz a `pip` csomagkezelőre is, tehát, ha `python3`-al dolgozunk akkor a `pip3`-al tudunk hozzá csomagokat telepíteni.
- A `pip` az új Python változatokban már a Python részeként telepítésre kerül.

4.1.1. pip

A `pip`-et [8] egyszerűen lehet használni terminálban vagy ha a Windows parancssorban begépeljük hogy `pip` (vagy `pip3`) és megadjuk a csomag nevét amit szeretnénk telepíteni:

```
pip somePackage  
python pip -m somePackage
```

Esetleg csinálhatunk olyat is, hogy összeírjuk a csomagokat egy szövegfájlba (soronként egy csomagot) és telepítjük a -

```
pip -r csomagnevek.txt
```

parancs segítségével [8].

4.1.2. Anaconda

A telepítés egy egyszerűsített változata, ha telepítjük az Anaconda környezetet. Az Anaconda egy tudományos, gépi tanulásos platform, melyben összegyűjtöttek több, mint 7500 Python és R csomagot, melyeket egyszerűen, grafikus kezelő felület mellett telepíthetünk, ezen kívül biztosít még IDE-ket (Spyder, Visual Studio Code) illetve a Jupyter notebookot, melyben a dolgozathoz készített példák is íródtak. Az Anaconda telepítése egyszerű. Felmegyünk a weboldalára, letölthjük és telepítjük. Windows alatt mind a 32, mind pedig a 64 bites verziót támogatott még Mac OS alatt választhatunk grafikus vagy parancssoros telepítő között. Linux alatt támogatott az IBM Power 8 és 9 architektúrája is, az x86-64 mellett.

4.1.3. Jupyter munkafüzet

A Jupyter notebookról még csak említés szintjén volt szó az eddigiekben. Most nézzünk bele kicsit mélyebben mi is ez. A Jupyter notebook tulajdonképpen egy nyílt forráskódú web alkalmazás mely lehetőséget nyújt arra, hogy olyan interaktív dokumentumokat készítsünk és osszunk meg, melyek egyszerre tartalmaznak futtatható kódokat és magyarázó szöveget, vizualizációkat [11]. A legelterjedtebben Python nyelvvel használják de támogat több, mint 40 programozási nyelvet köztük a C++, R, Ruby, julia és Calysto scheme programozási nyelveket. A notebook ideális adat vizualizációra, numerikus számításokhoz, gépi tanuláshoz és statisztikai modellekhez.

Maga a munkafüzet egy JSON szintaktikájú `.ipynb` kiterjesztésű fájl, mely úgynevezett cellákból áll. A celláknak az alábbi típusai vannak.

- **markdown**: a markdown cellákban lehet a szöveget írni különböző formázási lehetőségekkel. A markdown cellák támogatják a HTML elemeket és a L^AT_EX matematikai módját, illetve más leíró nyelvekből is vett át elemeket.
- **code**: ezekben a cellákban helyezhetjük el a kódunkat, melyet le szeretnénk futtatni.
- **heading**: Tulajdonképpen egy MarkDown cella, csak rak a Jupyter egy #-ot az elejére, ami a legnagyobb headinget jelöli.
- **raw**: egy cella mely nem kerül formázásra.

A munkafüzet az IPython shellt használja, amely lehetővé teszi nekünk ezt, és az interaktív widget-ek használatát is a munkafüzetben, melynek a segítségével így akár a kódunk paramétereit is megváltoztathatjuk.

5. fejezet

A Python eszközkészlet hatékonyságának vizsgálata

5.1. A hatékonyság típusai

A hatékonyság alatt általában a számítási hatékonyságot szokták érteni. Ez különösen igaz a numerikus számítások esetében. Nem szabad azonban figyelmen kívül hagyni a fejlesztéshez kapcsolódó további fontos jellemzőket sem. A NumPy a számítási műveletek jelentős részét alacsonyabb szinten (főként C nyelven) implementált egységek formájában tartalmazza. Emiatt annak számítási ideje jelentősen nem térhet el a natív implementációtól. Ezért a hatákonyság az alábbi szempontok kell, hogy fontosabb szerepet játszanak.

- A Python nyelv a számítási hatékonysággal szemben a program megírására, az elkészült programkód olvashatóságára helyezi a hangsúlyt. Annál hatékonyabbnak tekintjük tehát az eszközt, minél könnyebben értelmezhető és használható a benne készült program. Becsülhető például a programkód megírásához vagy megértéséhez, használatba vételéhez szükséges idővel.
- A hatékonyságot segíti, hogy ha az elkészült algoritmusok minél egyszerűbben átvihetők más rendszerekre, tehát annál jobb, minél portábilisabb a megoldás. (A Python esetében ez az értelmező átvihetősége és elterjedtsége miatt adott.)
- A hatékony használati módhoz hozzátarozik még az is, hogy a számítások be- menetét, a kapott eredményeket milyen könnyen lehet kezelni.
- Fontos továbbá a megjelenítés módja, amelyet a Python a Matplotlib könyvtár segítségével kifejezetten támogat.

A következőkben különböző témaületeket a numerikus módszerek tárgykörén belül, numerikus számítási módokat, a gyakorlatban és az oktatásban előforduló eseteket láthatunk, amely példaként szolgál arra, hogy a Python és az eszközkészlete milyen módon segíti a számítások elkészítésének, elvégzésének és ellenőrzésének a módját.

5.2. Hibaszámítás

Amikor hibáról beszélünk akkor sok mindenre gondolhatunk, például egy program írása közben kaphatunk futásidéjű vagy fordítási idejű hibákat. Nincs ez másképp a való

életben sem, ott is lehet hiba ha megteszünk valamit vagy ha éppen nem. Matematikai feladatok esetében az alábbi hiba típusokról beszélhetünk.

- Modellhiba: amikor a valóságnak csak egy közelítését használjuk egy feladat matematikai alakjának a felírásához.
- Mérési vagy öröklött hiba: amikor a modell adatai a valós pontos értékeknek csak valamilyen közelítő értékei, ezek általában a mérés pontosságától függnek
- Műveleti és input hiba: ezek a hibák a számítógépen tárolt adatok számítógépen való ábrázolásánból adódó hibák. Ezek a számábrázolási hibák azért léphetnek fel, mivel a számítógépen a racionális számoknak is csak egy részhalmaza ábrázolható lebegő pontos aritmetikában. A műveletek elvégzésénél fel léphet kerekítés, túlilletve alulcsordulás.
- Képlethiba: amikor a végtelen eljárást véges számú lépés után leállítunk, közelítő eljárásokat alkalmazunk.

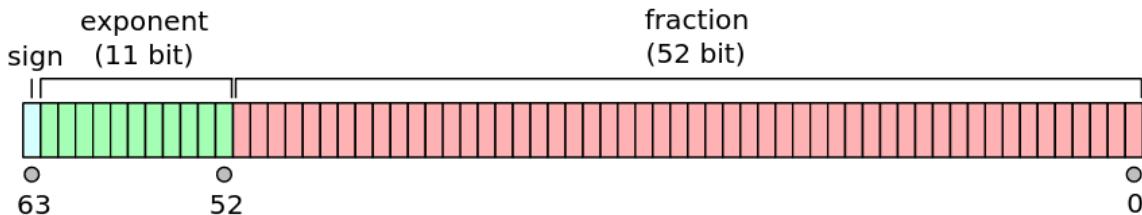
5.2.1. Számábrázolás

Egész számok

Számítógépen az egész számokat előjeles vagy előjel nélküli bináris számként lehet ábrázolni, ezáltal jellemzőek a számjegyek számával. Az egész számokat általában 2 vagy 4 byte-on tároljuk el 2-es, 10-es vagy 16-os számrendszerben. Ez az adott programozási nyelvtől függ. Az egészekkel végzett aritmetikai műveletek általában gyorsabbak a lebegő pontos számokkal végrehajtott aritmetikai műveleteknél, és ezáltal egy algoritmus lefutását fel tudja gyorsítani a használatuk. Használatuknál viszont vigyázni kell arra, hogy tulajdonképpen maradékosztályokban dolgozunk.

Lebegőpontos számok

Lebegő pontos számok esetében a számítógépek egy véges számhalmazt ábrázolnak és a számításokat is ezekkel a számokkal végzik el [12]. Általában a lebegőpontos aritmetikát használják. Ennek nézzük meg a modelljét az 5.1. ábrán.



5.1. ábra. Az egyszeres lebegőpontos számábrázolásnál a szám részeinek elrendezése IEEE 754-es szabvány szerint.

A nem nulla lebegőpontos számok általános alakja a következő

$$\pm a^k \left(\frac{m_1}{a} + \frac{m_2}{a^2} + \cdots + \frac{m_t}{a^t} \right),$$

ahol $a > 1$ a számábrázolás alapja, \pm az előjel, $t > 1$ a számjegyek száma és $k \in \mathbb{Z}$ a kitevő.

Az m_1 számjegy normalizált, $(1 \leq m_1 \leq a - 1)$ ez garantálja a az ábrázolás egyértelműségét. A többi számjegy: $1 \leq m_i \leq a - 1$ ($i = 2, 3, \dots, t$). A nulla az nem normalizált tehát $k = 0$, $m_1 = m_2 = \dots = m_t = 0$ és az előjele általában $+$. A számábrázolás alapja itt is lehet 2, 10, 16 vagy akár más is, átalában a programozási nyelven műlik melyiket használja. Pontosság szempontjából a $t = 8$ az egyszeres pontosság, $t = 16$ a dupla pontosság.

A géptől és pontosságtól függően m tárolására 32, 64 vagy 128 bit áll rendelkezésünkre (ez rendre 4, 8, 16 byte). Ezzel párhuzamosan nő a k értékkészlete, és adott pontosság mellett:

$$L \leq k \leq U.$$

A legnagyobb ábrázolható szám:

$$M^\infty = a^U(1 - a^{-t})$$

A legkisebb pedig:

$$-M^\infty$$

A lebegőpontos számok a $[-M^\infty, M^\infty]$ -beli számok diszkrét (racionális) részhalmazát alkotják és ez a részhalmaz a nullára nézve szimmetrikus. A nullához legközelebb eső lebegő pontos szám: $\varepsilon_0 = a^{L-1}$ és az ε_0 -hoz legközelebb eső szám pedig: $\varepsilon_0(1 + a^{1-t})$.

A gép relatív pontossága vagy másnéven a gépi epsilon a $\varepsilon_1 = a^{1-t}$.

5.2.2. Klasszikus hibaanalízis

Legyen A egy pontos érték, és legyen a ennek valamilyen közelítése. A $\Delta a = A - a$ mennyiséget a közelítés hibájának nevezik és a $|\Delta a| = |A - a|$ pedig az abszolút hibájának. Azt a δa értéket pedig abszolút hibakorlának nevezik amelyre fenáll, hogy $|A - a| = |\Delta a| \leq \delta a$.

Az A szám valamilyen közelítő értékének a relatív hibája pedig a $\frac{\delta a}{A}$ mennyisége.

Az additív műveletek abszolút hibákorlátja pedig a következők:

$$\delta(a + b) \leq \delta a + \delta b, \quad \delta(a - b) \leq \delta a + \delta b.$$

Az multiplikatív műveletek abszolút hibákorlátjai:

$$\delta(ab) \approx |a|\delta b + |b|\delta a, \quad \delta(a/b) \approx \frac{|a|\delta b + |b|\delta a}{|b|^2}.$$

Az aritmetikai műveletek relatív hibákorlátjai a következők, feltéve hogy a nevező sehol sem nulla és az additív műveleteknél az operandusok megegyező előjelűek:

$$\begin{aligned} \frac{\delta(a + b)}{|a + b|} &= \max\left(\frac{\delta a}{|a|}, \frac{\delta b}{|b|}\right), & \frac{\delta(a - b)}{|a - b|} &= \frac{\delta a + \delta b}{|a - b|}, \\ \frac{\delta(ab)}{|ab|} &\approx \frac{\delta a}{|a|} + \frac{\delta b}{|b|}, & \frac{\delta(\frac{a}{b})}{|\frac{a}{b}|} &\approx \frac{\delta a}{|a|} + \frac{\delta b}{|b|}. \end{aligned}$$

5.3. Vektor és mátrix műveletek

5.3.1. Vektorok

Pythonban a *NumPy* csomag használatával könnyedén definiálhatunk vektorokat és mátrixokat. Ahhoz, hogy használhassuk előtte telepíteni kell majd be kell a `pip` csomagkezelővel, majd az alábbi módon lehet importálni:

```
import numpy as np
```

Ez a szintaktikája Python-ban egy csomag importálásának. Az `as` operátor után aliaset adhatunk a csomagnak, így megkönyítve a használatát. Alias megadása nem kötelező. Ez esetben `np`-vel tudunk hivatkozni a *NumPy* csomagra. A csomagból az `array` metódus segítségével hozhatunk létre vektorokat, például

```
v1 = np.array([1, 2, 3])
v2 = np.array([3, 2, 1])
v3 = np.array([5, 3, 1, 6])
```

Üres vektort a következőképpen készíthetünk: az `empty` metódusban meg kell adni szögeletes zárójelek között egy dimenziót (például: `[1, 3]` ez azt jelenti 1 sort és 3 oszlopot szeretnénk), és esetlegesen megadhatunk neki egy adattípust, hogy milyen adatokkal szeretnénk feltölteni.

Itt főleg valamilyen numerikus adattípusra kell gondolni, hisz ezekkel tudjuk elvégezni a vektor és mátrix műveleteket. Használhatjuk a Python beépített típusait, mint az `int` vagy a `float`, de használhatjuk a *NumPy*-ban definiált kiegészített változatokat amivel megadhatjuk azt például, hogy hány bájton tároljuk az adott számot. Esetlegesen megadhatunk stringet és `bool` típust is ha szükség van rá.

Az `empty` nem feltétlenül 0-ákkal tölti fel a vektort hanem valamilyen memória szeméttel.

```
v4 = np.empty([1, 3])
```

Ennek az oka nagyon egyszerű: így gyorsabb, mint ha nullázná az elemeket, de ha 0-ákkal szeretnénk feltölteni használhatjuk a `zeros` metódust mely hasonló képpen működik mint az `empty`:

```
v5 = np.zeros([1, 3])
```

1-esekkel is feltölthetjük ehez a `ones` metódust használhatjuk:

```
v6 = np.ones([1, 3])
```

Létrehozhatunk egy bizonyos értékkal vagy pszeudóvéletlen (random) számokkal feltöltött vektorokat is :

```
v7 = np.random.rand(1, 3)
v8 = np.full([1, 3], "aaa")
```

Utóbbiak a következő értékeket tartalmazzák:

```
[[0.22541693 0.47521003 0.90312494]]
[['aaa' 'aaa' 'aaa']]
```

A fent létrehozott `v1`, `v2`, `v3` vektorainkal már egyszerűen elvégezhetőek a vektor műveletek, mint például az összeadás, kivonás, vektoriális, skaláris szorzás és konstanssal való szorzás:

```
v1 + v2
v1 - v2
np.outer(v1, v2)
np.inner(v1, v2)
v1 * 3
```

Amennyiben egyszerűen a összeszorozzuk a két vektort akkor a megfelelő hely lévő tagokat szorozaz össze:

```
v1 * v2
```

Transzponálhatjuk is a vektorainkat a `transpose` metódussal bár vektorok esetén itt nem látványos:

```
v3.transpose()
```

Komolyabb műveletek mint a vektor normák kiszámítása is egyszerű. Vegyük először az 1-es normát:

```
np.linalg.norm(v1, 1)
```

Itt használtuk a numpy `linalg` csomagját melyben előre definiálva vannak a különböző lineáris algebrához tartozó műveletek, módszerek. Most nézzük a 2-es és a végtelen normát:

```
np.linalg.norm(v1, 2)
np.linalg.norm(v1, np.inf)
```

Mind itt a vektorműveleteknél, és majd a mátrix műveleteknél sem árt figyelni, hogy megfelelő dimenziójú vektorokat, mátrixokat adjunk meg a műveletekhez, különben könnyen hibát vagy helytelen eredményt kaphatunk.

5.3.2. Mátrixok

A Mátrixok létrehozása is többféleképpen történhet, hasonlóképpen mint a vektoroknál. Legegyszerűbb módszer az, ha megadjuk mi, hogy milyen mátrixot szeretnénk vagy képezhetjük vektorokból is, estleg a fent megismert `empty`, `zeros`, `ones`, `random.rand`, `full` metódusoknak megadjuk a nekünk megfelelő dimenziókat.

```
m1 = np.matrix([[1, 2, 4], [2, 3, 4]])
m2 = np.matrix([[1, 2, 8], [2, 3, 9]])
m3 = np.matrix([[1, 2], [2, 3], [4, 5]])
m4 = np.matrix([[1, 2], [3, 4]])
```

Tartományok segítségével is létrehozhatunk mátrixokat.

```
m = np.arange(16.0)
m = np.arange(16.0).reshape(4, 4)
```

A `reshape`-nél vigyázni kell, hogy pontosan akkora elemszámú mátrixot adjunk meg mint amekkora a mostani mátrixunké, különben hibát kapunk. Amennyiben mégis kisebb vagy nagyobb mátrixot szeretnénk használni, át kell másolni az elemeket egyik mátrixból a másikba. Az `eye` metódus segítségével képezhetünk $n*n$ -es egységmátrixot, paraméterként a dimenziót kell megadnunk:

```
m = np.eye(3)
```

amely a következő mátrixot hozza létre:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

A vektorokhoz hasonlóan adhatjuk meg a mátrix műveleteket is, mint az összeadást, kivonást, szorzást, normákat:

```
m1 + m2
m1 - m2
m1 * m3
np.linalg.norm(m1, 'fro')
np.linalg.norm(m1, np.inf)
np.linalg.norm(m1, 1)
np.linalg.norm(m1, 2)
```

A transponálás is hasonlóan működik:

```
m3.transpose()
```

Egy négyzetes mátrix determinánsát és inverzét is egyszerűen és gyorsan kiszámíthatjuk a következő formában:

```
np.linalg.det(m4)
np.linalg.inv(m4)
```

Egy elemet az `item` metódussal tudunk kiválasztani, de figyelni kell, mert az indexelés 0-tól kezdődik:

```
print("Az m3 2. sor 1. eleme", m3.item(1, 0))
```

A mátrixokat feldarabolhatjuk a `hsplit` és `vsplit` metódusok segítségével. A `hsplit` oszlopok mentén, míg a `vsplit` sorok mentén vágja el a megadott mátrixot:

```
np.hsplit(m, 2)
np.vsplit(m, 2)
```

Használhatjuk vágásra az `[]` operátorokat is, ha csak egy paramétert adunk meg neki akkor az adott indexű sort kapjuk vissza, ha használjuk a `:` operátort akkor megadhatunk neki intervallumot is, hogy hanyadik sornál kezdje, megadhatunk neki lépésközöt is illetve részeket is kivághatunk egy adott mátrixból, például

```
m[1:3]
m[:, 1:3]
m[1:3, 1:3]
```

Ha adunk meg neki lépésközt, akkor az az utolsó paraméter. Például az egész mátrix minden párátlan számú sorának és oszlopának a közös elemei:

```
m[0:4:2, 0:4:2]
```

5.4. Mátrix felbontások

5.4.1. LU felbontás

Mátrixok LU felbontásával kinyerhetjük a felső- és alsóháromszög mátrixot. Pythonban erre használható a `Scipy.linalg.lu` metódus ami a SciPy tudományos csomagban találhatunk meg ami egy kiegészítése tulajdon képpen a NumPy-nak. Először telepíteni és importálni kell ezt a csomagot és utána használatba lehet venni. Az `lu` metódus végeredményként visszaadja az alsó-, felső- és a permutáció mátrixokat.

```
import scipy as sp
from scipy.linalg import lu
m = np.arange(16.0).reshape(4, 4)
P, L, U = sp.linalg.lu(m)
```

5.4.2. Cholesky felbontás

A Cholesky féle felbontásra is találunk beépített metódust a Numpy-ban `cholesky` néven a `linalg` csomagban. Cholesky felbontásnál vigyázni kell hogy a mátrixunk szimmetrikus legyen és pozitív definit ellenben hibaüzenetet kapunk. Ugyan ez megtalálható a SciPy csomagban is és a különbség az, hogy a NumPy-ban lévő metódus az alsó-, addig a SciPy-ban lévő a felsőháromszög mátrixot számolja:

```
m = np.arange(16.0).reshape(4, 4)
L=np.linalg.cholesky(m)
```

A fenti kód hibát eredményezett, mert a megadott mátrix nem volt pozitív definit. A következő példában már pozitív definit mátrix szerepel:

```
m = np.array([[2, -1, 0], [-1, 2, -1], [0, -1, 2]])
L = np.linalg.cholesky(m)
L = sp.linalg.cholesky(m)
```

Természetesen megírhatjuk a saját felbontó függvényünket is:

```
import numpy as np
import scipy as sp

def cholesky(M, dimensions):
    P, L, U = sp.linalg.lu(M)
    k = 0
    while k < dimensions:
        j = 0
        s = 0
```

```

while j < k - 1:
    s = L[k, j] * L[k, j]
    j += 1
L[k, k] = np.sqrt(M[k, k] - sum)
i = k + 1
while i < dimensions:
    j = 0
    s = 0
    while j < k - 1:
        s = L[i, j] * L[k, j]
        j += 1
    L[i, k] = (M[i, k] - s) / L[k, k]
    i += 1
    k += 1
return L

m = np.array([[2, -1, 0], [-1, 2, -1], [0, -1, 2]])
dimension = 3
l = cholesky(m, dimension)

```

Ahogy láthajuk a metódus megadja egy közelítő megoldását a cholesky felbontásnak.

5.5. Lineáris egyenletrendszer

5.5.1. Általánosan a lineáris egyenletrendszerkről

A lineáris egyenletrendszer egy többismeretlenes egyenletrendszer melyben az ismeretlenek az első hatványon vannak.

Egy n egyenlet és m változó esetén az egyenletrendszer általános alakja a következő:

$$\begin{aligned}
 x_{1,1} + x_{1,2} + \cdots + x_{1,m} &= b_1 \\
 x_{2,1} + x_{2,2} + \cdots + x_{2,m} &= b_2 \\
 &\vdots \\
 x_{i,1} + x_{i,2} + \cdots + x_{i,m} &= b_i \\
 &\vdots \\
 x_{n,1} + x_{n,2} + \cdots + x_{n,m} &= b_n
 \end{aligned}$$

Azt mondjuk, hogy az egyenlet rendszer alulhatárolt, hogy ha $n < m$ és túlhatárolt, hogy ha $n > m$. Négyzetesnek nevezzük, ha $m = n$. Az egyenletrendszer geometriai tartalmát az alábbi módon írhatjuk le: Az R^n euklideszi tér $d \in R^n$ normálvektorú és $x_0 \in R^n$ ponton átmenő hipersíkját az

$$(x - x_0)^T d = 0$$

egyenletet kielégítő $x \in R^n$ pontok határozzák meg.

Az $Ax = b$ egyenletrendszert felírhatjuk ilyen formában is:

$$\begin{aligned} a_1^T x &= b_1, \\ a_2^T x &= b_2, \\ &\vdots \\ a_n^T x &= b_n, \end{aligned}$$

ahol $a_i^T = a_{i1}, \dots, a_{im}$. Innen láthatjuk, hogy az egyenletrendszer megoldása az m hiper sík közös része, ennek megfelelően 3 megoldás lehetséges: nincs megoldás, pontosan egy megoldás van vagy végtelen sok megoldás van. Ha az $Ax = b$ egyenletrendszernek legalább egy megoldása létezik konzisztensnek nevezzük. Ha nem létezik egyetlen megoldása sem, akkor az egyenletrendszer inkonzisztens.

5.5.2. Lineáris egyenletrendszerek megoldása Python-ban

Egy egyenletrendszert megoldani Python-ban nem nehéz. A NumPy tartalmazza a `solve` metódust mellyel könnyedén megoldható egy-egy lineáris egyenletrendszer. A `solve` a NumPy linalg csomagjában található, paramáterként várja az A együtthatómátrixot és a b megoldásvektort. Nézzük is meg! Vegyük először egy egyszerű egyenletrendszeret, mint a következő:

$$\begin{aligned} 2x_1 + 9x_2 + 8x_3 + 7x_4 &= 7, \\ 3x_1 - 2x_2 + 6x_3 + 4x_4 &= 4, \\ 5x_1 + 8x_2 + 4x_3 + 7x_4 &= 2, \\ 6x_1 + 9x_2 + 10x_3 + 2x_4 &= 6. \end{aligned}$$

Importáljuk a `numpy` és a `time` csomagot:

```
import numpy as np
import time
```

Miután ezzel megvagyunk hozzuk létre az együttható métrixunkat és a megoldás vektorunkat:

```
a = np.array([
    [2, 9, 8, 7],
    [3, -2, 6, 4],
    [5, 8, 4, 7],
    [6, 9, 10, 2]])
b = np.array([7, 4, 2, 6])
```

Használjuk a `solve` metódust

```
start = time.time()
import numpy as np
x_solve = np.linalg.solve(a, b)
print("Eredmény:")
print(x_solve)
```

```
time_solve=time.time() - start
print("\nFutási idő:\n")
print(time_solve)
```

A solve az Intel Math Kernel Library-ben megtalálható LAPACK gesv rutint használja. A futás időt is kiírja a program, hogy később össze lehessen vetni a beépített és az általam implementált algoritmust. Az allclose és dot metódusok segítségével pedig le is ellenőrizhetjük az eredményt.

```
np.allclose(np.dot(a, x_solve), b)
```

Ilyen egyszerű az egész, de ha nem szeretnénk az előre megírt metódust alkalmazni akkor definiálhatunk saját magunk is nekünk tetsző megoldó metódust is. Például implementálhatjuk a Gauss módszert is. Ám mielőtt ezt megnéznénk vessünk egy pillantást a ciklusokra Python-ban csak, hogy érthető legyen a kód.

Egy lineáris egyenletrendszer csak akkor megoldható ha az A matrixunk rangja megegyezik az $[A, b]$ mátrix rangjával. Ilyenkor az egyenletrendszerünknek pontosan egy megoldása van. Ezt Python-ban a következőképpen tudjuk ellenőrizni:

```
print("A matrix rangja:")
print(np.linalg.matrix_rank(a))
c = np.column_stack((a, b.transpose()))
print("[A, b] matrix rangja:")
print(np.linalg.matrix_rank(c))
```

Ennek az eredményéből látható lesz, hogy a két rang megegyezik, ezáltal megoldható az itt látható egyenlet rendszer.

A Gauss módszer elkészítése kapcsán bemutathatók a Python nyelv vezérlési szerkezetei. Python-ban is megtalálható a for és a while ciklus. A for végig iterál egy iterálható objektumon ami pythonban lehet egy tömb, string vagy az általunk kreált iterálható struktúra.

```
s = "string"
tomb = [1, 3, 5]
collection = ["string", 8, 'a']

for i in s:
    print(i)

for i in tomb:
    print(i)

for i in collection:
    print(i)
```

Ha klasszikus értelemben szeretnénk használni (tehát egy ciklus változót léptetni egy kezdő értéktől egy végértékgig) használnunk kell a range metódust mely generál egy tömböt a megadott paraméterek alapján. Hárrom paramétere van, az első a kezdő érték, a második a végérték és a harmadik a lépésköz. A range esetében figyelni kell, mert a végéértékgig generálja a számokat ezért a végérték nem szerepel az általa vissza adott iterálható példányban:

```

x = range(1, 4)

for i in x:
    print(i)

for i in range(1, 50, 2):
    print(i)

```

A `while` működése nem tér el a többi általános célú programozási nyelvekben megszokottól. Amíg a megadott feltételünk igaz addig maradunk a ciklusban, például

```

x = 0
while x < 10:
    x += 1

```

Nézzük még meg az `if` elágazást is mely megvizsgál egy logikai kifejezést és annak értéke szerint ágaztatja el a programot.

```

if True:
    print("igaz")

if False:
    print("igaz")
else:
    print("hamis")

if 2 + 2 == 4:
    print("two plus two is four")

```

Ezen ismeretek birtokában már belekezdhetünk a Gauss módszer implementációjának vizsgálatába:

```

import numpy as np
import time

def gauss(a, b, dimension):
    l = np.zeros([dimension, dimension])
    x = np.zeros([dimension])

    for k in range(0, dimension - 1):
        for i in range(k + 1, dimension):
            l[i, k] = a[i, k] / a[k, k]
            b[i] = b[i] - (l[i, k] * b[k])
            for j in range(k, dimension):
                a[i, j] = a[i, j] - (l[i, k] * a[k, j])

    x[dimension - 1] = b[dimension - 1] / a[dimension - 1, dimension - 1]

    for i in range(dimension - 1, -1, -1):
        s=0

```

```

        for j in range(i + 1, dimension):
            s = s + a[i, j] * x[j]
            x[i] = (b[i] - s)/a[i, i]
    return(x)

start = time.time()
a = np.array([
    [2, 9, 8, 7],
    [3, -2, 6, 4],
    [5, 8, 4, 7],
    [6, 9, 10, 2]])
b = np.array([7, 4, 2, 6])
dimension = 4
x_g = gauss(a, b, dimension)
print("Eredmeny:\n")
print(x_g)
time_gauss = time.time() - start
print("\nFutasi ido:\n")
print(time_gauss)

```

A futtatás után láthatóvá válik, hogy időben elég közel van a numpy-ban lévő megoldáshoz. De nem kapunk olyan pontos eredményt.

Tegyük most kis kitérőt a számítási munkára. Ezt valahogyan mérni kell, hogy megtudjuk adni egyes algoritmusoknak a műveletigényét, és ennek a mértékegysége a flop. Egy régi flop az a számítási munk ami az $s = s + x * y$ művelet (egy összeadás és egy szorzás) elvégzéséhez kell, 1 új flopp pedig az a számítási munka mely egyetlen művelet (mindegy az hogy additív vagy multiplikatív) elvégzéséhez szükséges. Az új flop bevezetését az indokolta, hogy a mai számítógépeken a multiplikatív és az additív műveletek elvégzéséhez szükséges idő azonosnak tekinthető.

A gauss módszer $\frac{n^3}{3} + O(n^2)$ additív és ugyanennyi multiplikatív műveletet igényel, így a művelet igénye: $\frac{n^3}{3} + O(n^2)$ régi flop és $\frac{2n^3}{3} + O(n^2)$ új flop.

Főelem kiválasztás

A fenti algoritmus csak olyan mátrixokat képes megoldani amikben egyik együtt-ható sem nulla. Hogy képes legyen rá ki kell egészíteni főelem kiválasztással. A főelem kiválasztás azt jelenti, hogy a sorokat úgy cseréljük fel, hogy az aktuális pivot elemünk ne legyen nulla. A főelemkiválasztás lehet részleges vagy teljes. Részleges főelem kiválasztásnál a k -adik lépésben megkeressük a k -adik oszlop elemei közül a maximális abszolút értékű a_{ik} együtthatót és ez az j -edik sorban van akkor megcseréljük a j -edik és a k -adik sort. Ezzel fogunk majd találkozni a gauss-Jordan módszer implementációjánál. A teljes főelem kiválasztás során pedig a k -adik lépésben megkeressük a mátrixban szereplő értékek közül a maximális abszolút értékűt és ha ez a a_{ij} akkor a k -adik sort felcseréljük az i -edikkkel és a k -adik oszlopot is felcseréljük a j -edikkkel.

Gauss-Jordan módszer

Egy másik lehetséges módszer az egyenletrendszerek megoldására a Gauss-Jordan módszer. Ezt implementálhatjuk mi magunk is, de én most egy másik csomagot a

SymPy-t fogom segítségül hívni. Első lépések vegyük az A mátrixunkat és a b vektorunkat, majd használjuk az A mátrixra definiált `gauss_jordan_solve` mmetódust melynek paraméterként a b vektort adjuk meg. A metódus két visszatérési értéke a megoldás és a mátrix.

```
from sympy import Matrix

start = time.time()
A = Matrix([
    [2, 9, 8, 7],
    [3, -2, 6, 4],
    [5, 8, 4, 7],
    [6, 9, 10, 2]])
b = Matrix([7, 4, 2, 6])
sol, params = A.gauss_jordan_solve(b)

n, m = sol.shape
x_gj= np.zeros(n)
for i in range(0, n):
    x_gj[i]= sol[i]

time_gauss_jordan = time.time() - start
print("Eredmeny:")
print(x_gj)
print("Futasi ido:")
print(time_gauss_jordan)
```

Fontos figyelembe venni, hogy a `Matrix` nem kompatibilis az `np.array`-el, így az értékeket nekünk kell áthelyezni.

Van egy 4. módszer is a lineáris egyenlet rendszer megoldására a `scipy` csomagban, ami az LU-faktorizáción alapul. A metódus neve pedig `lu_solve` és az alábbi módon lehet használni:

```
from scipy.linalg import lu_factor, lu_solve

start=time.time()
A = Matrix([
    [2, 9, 8, 7],
    [3, -2, 6, 4],
    [5, 8, 4, 7],
    [6, 9, 10, 2]])
b = Matrix([7, 4, 2, 6])
lu, piv = lu_factor(A)
x_lu = lu_solve((lu, piv), b)

time_lu = time.time() - start
print("Eredmeny:")
print(x_lu)
print("futasi ido:")
print(time_lu)
```

A pandas függvénykönyvtár segítségével összesítő táblázatot is készíthetünk, amely eredményét az 5.2. ábrán láthatjuk.

	x1	x2	x3	x4	time
solve	5.755000	-0.006667	0.10000	-0.750000	0.001000
gauss	-0.382500	0.010000	0.85000	0.125000	0.001003
gauss_jordan	-0.517621	0.035977	0.85279	0.127019	0.003004
Lu	-0.517621	0.035977	0.85279	0.127019	0.001021

5.2. ábra. Az egyenletrendszer megoldó módszerek eredményeinek összesítése és megjelenítése Pandas segítségével.

5.6. Interpolációk

Az interpolációk alapfeladata az, hogy egy $f(x)$ függvény felvett értékeit különböző $x_1, x_2, x_3 \dots x_n$ pontokban ismerjük az $[a, b]$ ($a = x_1, b = x_n$) intervallumban és magát az f függvényt szeretnénk közelíteni egy könnyen számolható $h(x)$ függvénnnyel, amelyre fenáll, hogy $f(x_i) = h(x_i)$. Az $x_i, i = 1, \dots, n$ pontokat interpolációs alappontoknak, a feltételelt interpolációs feltételnek nevezzük. Az interpolációs feltétel teljesülése esetén azt reméljük, hogy az interpoláló $h(x)$ függvény jól közelíti az $f(x)$ függvényt az $[x_i, x_i + 1]$ intervallumokban. Itt a Lagrange, Hermit és Spline interpolációkról fogok elsősorban írni.

Legegyszerűbben interpolációt az `interp1d` függvénytel tudunk végre hajtani. Ez a módszer nem kér csak x, y értékeket és az interpoláció módját. A metódus a `scipy` csomag `interpolate` alcsomagjában található meg. A következő kódrészletben egy ezzel készített példa látható.

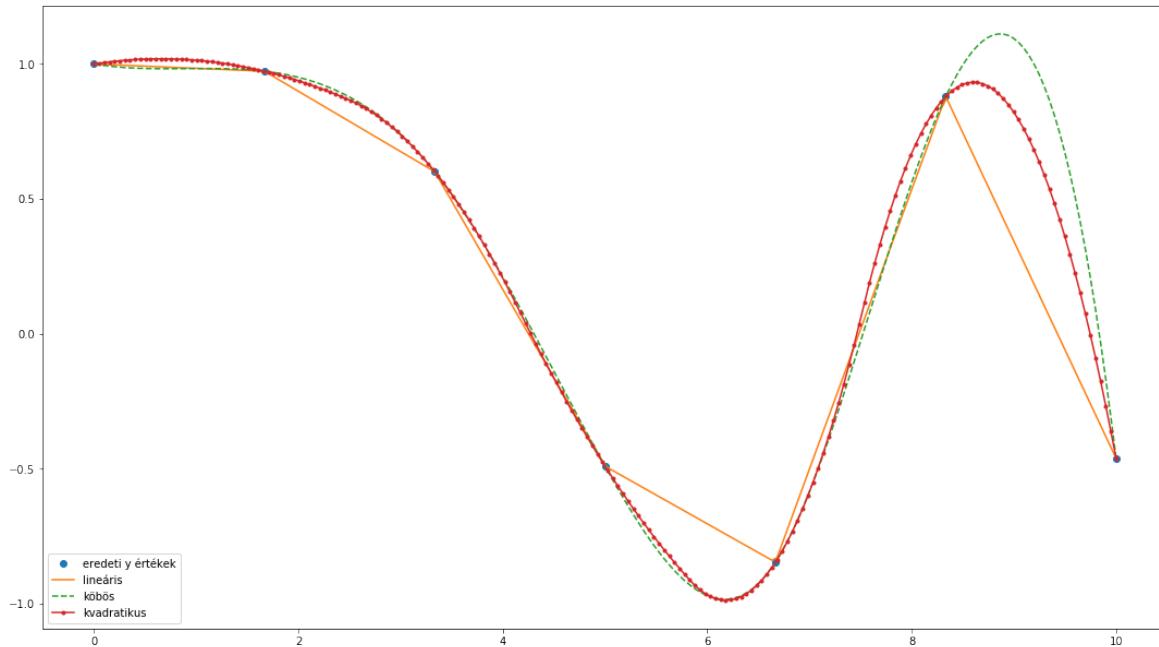
```
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

x = np.linspace(0, 10, num=7, endpoint=True)
y = np.cos(-x**2/12)
f = interp1d(x, y)
f2 = interp1d(x, y, kind='cubic')
f3 = interp1d(x, y, kind='quadratic')
xnew = np.linspace(0, 10, num=200, endpoint=True)

plt.figure(figsize=(18.5, 10.5))
plt.plot(x, 'o',
          xnew, f(xnew), '--', xnew, f2(xnew), '—', xnew, f3(xnew), '-.')
plt.legend(['eredeti y értékek', 'linearis', 'kobos', 'kvadratikus'],
```

```
    loc='best')
plt.show()
```

A programrész kimenetét az 5.3. ábrán tekinthetjük meg.



5.3. ábra. Példa az interpolált pontok számítására és megjelenítésére.

Látható, hogy az `interp1d` egy függvénytelér vissza, aminek csak meg kell adnunk az x értékeinket és vissza adja az eredményt. Ezek használhatóak a legkönnyebben Python-ban. fontos, hogy miután elvégeztük az interpolációt adott pontokra, a függvénynek azonos intervallumon de több alppontot tartalmazó tömböt adjunk át krajzolásnál, hogy szépen és pontosan rajzolja ki a közelítő függvényt.

Míg az `interp1d` $y = f(x)$ típusú függvényeket interpolál, addig ennek a metódusnak a párja az `interp2d` már az $z = f(x, y)$ típusú egyenleteket tudja közelíteni. használata az `interp1d`-hez hasonló, például

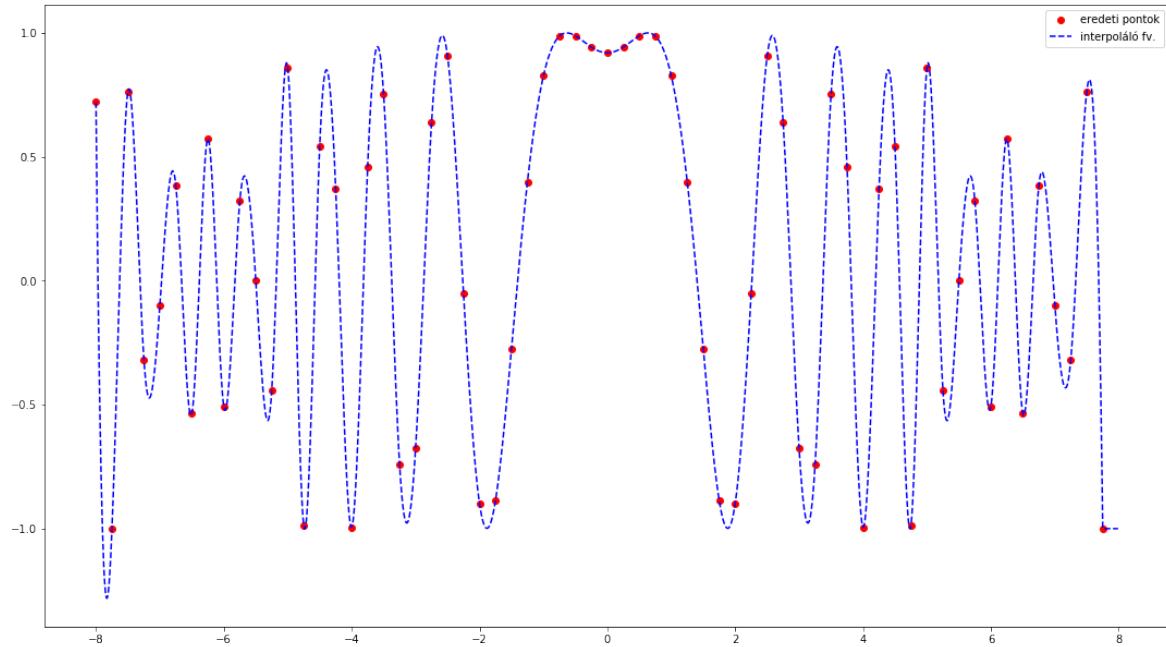
```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt
from matplotlib import figure

x = np.arange(-8, 8, 0.25)
y = np.arange(-8, 8, 0.25)
xx, yy = np.meshgrid(x, y)
z = np.sin(xx**2 + yy**2)
f = interpolate.interp2d(x, y, z, kind='cubic')

xnew = np.arange(-8, 8, 1e-2)
ynew = np.arange(-8, 8, 1e-2)
znew = f(xnew, ynew)
```

```
plt.figure(figsize=(18.5, 10.5))
plt.plot(x, z[0, :], 'ro', xnew, znew[0, :], 'b--')
plt.legend(["eredeti pontok", "interpoláló fv."])
plt.show()
```

A program kimenetét az 5.4. ábrán láthatjuk.



5.4. ábra. Példa az `interp2d` használatára és eredményének megjelenítésére.

További egyszerűen használható függvény a numpy csomagban található `interp1d`. Az `interp1d` esetében meg kell adni az x és y értékeinket illetve egy olyan intervallumot az $[a, b]$ intervallumon ami több alpontot tartalmaz. A megadott alpontok számától függ a pontosság tehát lehetőleg elég sokat adjunk meg. Az `interp1d` szakaszokra ad meg lineáris interpolációt. Ennek használata látható a következő kódpéldában.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

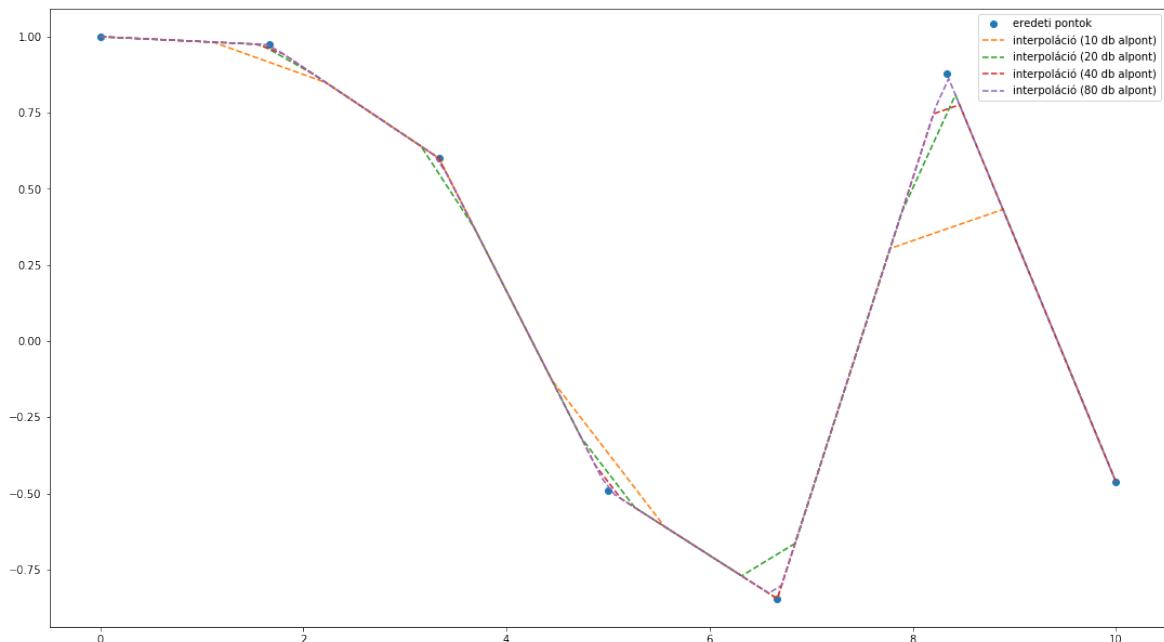
x = np.linspace(0, 10, num=7, endpoint=True)
y = np.cos(-x**2 / 12)
xnew1 = np.linspace(0, 10, num=10, endpoint=True)
xnew2 = np.linspace(0, 10, num=20, endpoint=True)
xnew3 = np.linspace(0, 10, num=40, endpoint=True)
xnew4 = np.linspace(0, 10, num=80, endpoint=True)
yinterp1 = np.interp(xnew1, x, y)
yinterp2 = np.interp(xnew2, x, y)
yinterp3 = np.interp(xnew3, x, y)
yinterp4 = np.interp(xnew4, x, y)

plt.figure(figsize=(18.5, 10.5))
```

```

plt.plot(x, y, 'o',
          xnew1, yinterp1, '—',
          xnew2, yinterp2, '—',
          xnew3, yinterp3, '—',
          xnew4, yinterp4, '—')
plt.legend([
    "eredeti pontok",
    "interpolacio (10 db alpont)",
    "interpolacio (20 db alpont)",
    "interpolacio (40 db alpont)",
    "interpolacio (80 db alpont)"])
plt.show()
    
```

A kimenete az 5.5. ábrán látható.



5.5. ábra. Példa az `interp1d` használatára és eredményének megjelenítésére.

5.6.1. Lagrange interpoláció

Legyenek a ϕ_i bázisfüggvények a következők $\phi_1 = 1, \phi_2 = x, \dots, \phi_n = x^{n-1}$ és legyenek x_1, x_2, \dots, x_n alpontjaink és $y_i = f(x_i)$ az alpontokhoz tartozó függvény értékek. Ekkor a feladatunk az, hogy határozzuk meg a legfeljebb $n-1$ -ed fokú p polinomot amelyre igaz, hogy $p(x_i) = y_i$. Ez tulajdonképpen az alapfeladat a lényegi rész az, hogy ezt a p polinomot hogyan állítjuk elő. A Lagrange féle előállítás a következőképpen néz ki:

$$l_i(x) = \prod_{k=1, k \neq i}^n \frac{x - x_k}{x_i - x_k},$$

majd ezekhez az l_i értékeket megszorozzuk a y_i értékeket és megkapjuk a p polinomot

$$p(x) = \sum_{i=1}^n y_i l_i(x).$$

Ezáltal meg kapjuk az $f(x)$ függvényünk közelítését a $p(x)$ polinom által ($f(x) \approx p(x)$).

Pythonban a `scipy.interpolate` csomag `lagrange` függvényével tudunk a lagrange interpolációt számolni. Figyelni kell arra, hogy egy `poly` nevű változóba tér vissza.

```
import numpy as np
from scipy.interpolate import lagrange

import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

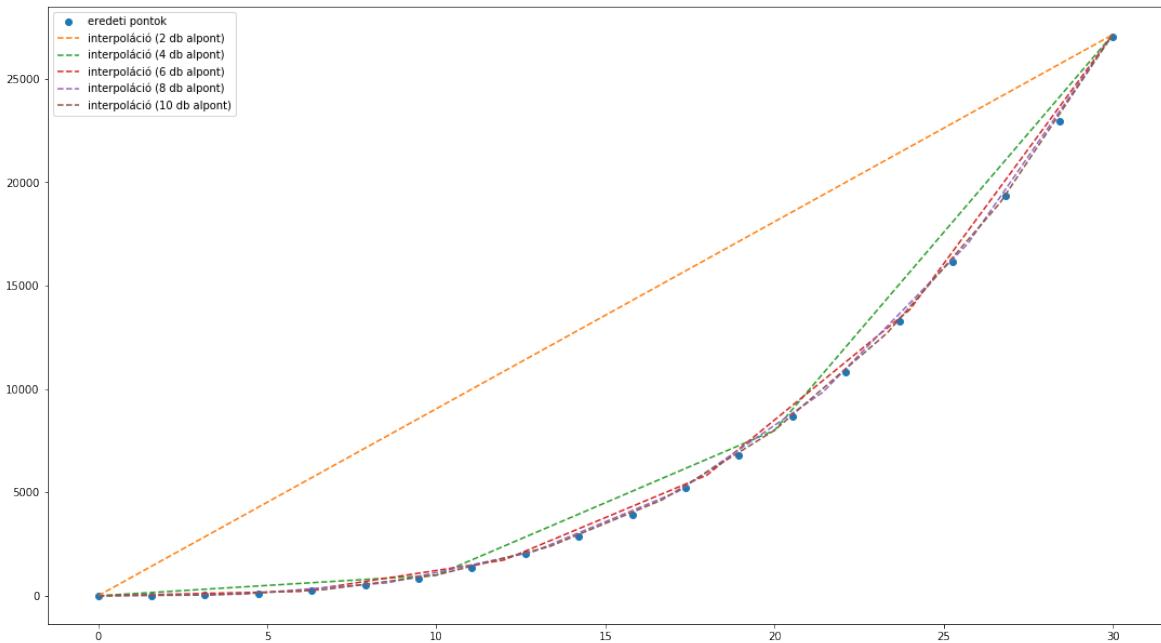
x = np.linspace(0, 30, num=20, endpoint=True)
y = x**3
poly=lagrange(x,y)

xnew1 = np.linspace(0, 30, num=2, endpoint=True)
xnew2 = np.linspace(0, 30, num=4, endpoint=True)
xnew3 = np.linspace(0, 30, num=6, endpoint=True)
xnew4 = np.linspace(0, 30, num=8, endpoint=True)
xnew5 = np.linspace(0, 30, num=10, endpoint=True)

plt.figure(figsize=(18.5, 10.5))
plt.plot(x, y, 'o',
          xnew1, poly(xnew1), '—',
          xnew2, poly(xnew2), '—',
          xnew3, poly(xnew3), '—',
          xnew4, poly(xnew4), '—',
          xnew5, poly(xnew5), '—')
plt.legend([
    "eredeti pontok",
    "interpolacio (2 db alpont)",
    "interpolacio (4 db alpont)",
    "interpolacio (6 db alpont)",
    "interpolacio (8 db alpont)",
    "interpolacio (10 db alpont)"])
plt.show()
```

A program futásának eredménye az 5.6. ábrán látható.

Látszik, hogy a pontosság itt is függ a megadott alpontok számától. Bár a Lagrange interpolációnak létezik Python-os implementációja, numerikusan instabil ezért ha mindenféleképpen ezt szeretnénk használni, érdemesebb a saját implementációt készíteni.



5.6. ábra. Példa a Lagrange interpolációra.

5.6.2. Spline interpoláció

Spline interpoláció esetén is megvannak az x_i pontjaink és y_i függvényértékeink, és ezek mellett keressük azt az $S(x)$ függvényt, mely teljesíti az alábbi feltételeket:

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

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 $[a, b]$ intervallumon.

A spline meghatározásánál felírunk n darab k -ad fokú polinomot. Ebből látszik, hogy az ismeretlenek száma $n(k + 1)$. Az első és a harmadik feltételből következik az, hogy a feltételek száma $(k + 1)n - (k - 1)$, ugyanis $k - 1$ db simasági feltétel a $n - 1$ belső pontban ebből jön az, hogy $(n - 1)(k - 1)$ és $2n$ interpolációs feltétel. Összesen $(n - 1)(k - 1) + 2n = (k + 1)n - (k - 1)$, és ebből látszik, hogy hiányzik a spline egyértelműségéhez még $k - 1$ darab feltétel. Ezeket a végpontokra szokták megadni.

Vegyük először a lineáris spline interpolációt ebben az esetben a $k = 1$, és a három feltétel egyértelműen meghatározza. minden $[x_k, x_{k+1}]$ intervallumon:

$$\begin{aligned} S_k(x_k) &= a_k x_k + b_k = y_k \\ S_k(x_{k+1}) &= a_k x_{k+1} + b_k = y_{k+1}. \end{aligned}$$

Ebből a kétismeretlenes egyenletrendszerből meghatározható a_k és b_k .

Beszélhetünk még kvadratikus splinekról is. Ekkor a $k = 2$ és $k - 1 = 1$ feltétel hiányzik a spline egyértelmű felírásához. Ezt a feltételt általában az intervallum elején vagy végén a derivált megadásával szokás teljesíteni. Az így felvázolt esetben az egymás melletti intervallumokra Hermite interpolációt alkalmazva meghatározható a spline.

Gyakorlatban azonban a harmadfokú spline interpolációt használjuk túlnyomó részt, és csak harmadfokú splineről beszélünk, viszont ezek további feltételek felírását követelik meg, úgy mint:

$$\begin{aligned} S'_i(x_{i+1}) &= S'_{i+1}(x_{i+1}), & (i = 1, \dots, n-2), \\ S''_i(x_{i+1}) &= S''_{i+1}(x_{i+1}), & (i = 1, \dots, n-2), \\ S''(x_1) &= A_n \quad \text{és}, & S''(x_n) = B_n \end{aligned}$$

Pythonban használhatjuk a harmadfokú spline-t is. Ehhez elő kell készíteni az értékeket a `scipy.interpolate` csomagban megtalálható `splprep` függvényel, majd utána tudjuk használni a `splev` függvényt, amely a harmadfokú spline-t adja vissza.

Először vegyük egy függvényt és pár alpontot. Legyen most ez a szinusz függvény és próbáljuk ezt közelíteni spline-al!

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
from matplotlib.pyplot import figure

x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/8)
y = np.sin(x)
tck = interpolate.splrep(x, y, s=0)
xnew = np.arange(0, 2*np.pi, np.pi/50)
ynew = interpolate.splev(xnew, tck, der=0)

plt.figure(figsize=(18.5, 10.5))
plt.plot(x, y, '-x', xnew, ynew, 'g—')
plt.legend(['sin(x) adott pontokra', 'harmadfoku spline'])
plt.axis([-0.05, 6.33, -1.05, 1.05])
plt.show()
```

Az 5.7. ábrán látható, hogy a harmadfokú spline jól közelíti a függvényünket, hiszen nagyjából lefedi, és a megadott pontokban felveszi a megadott értékeket.

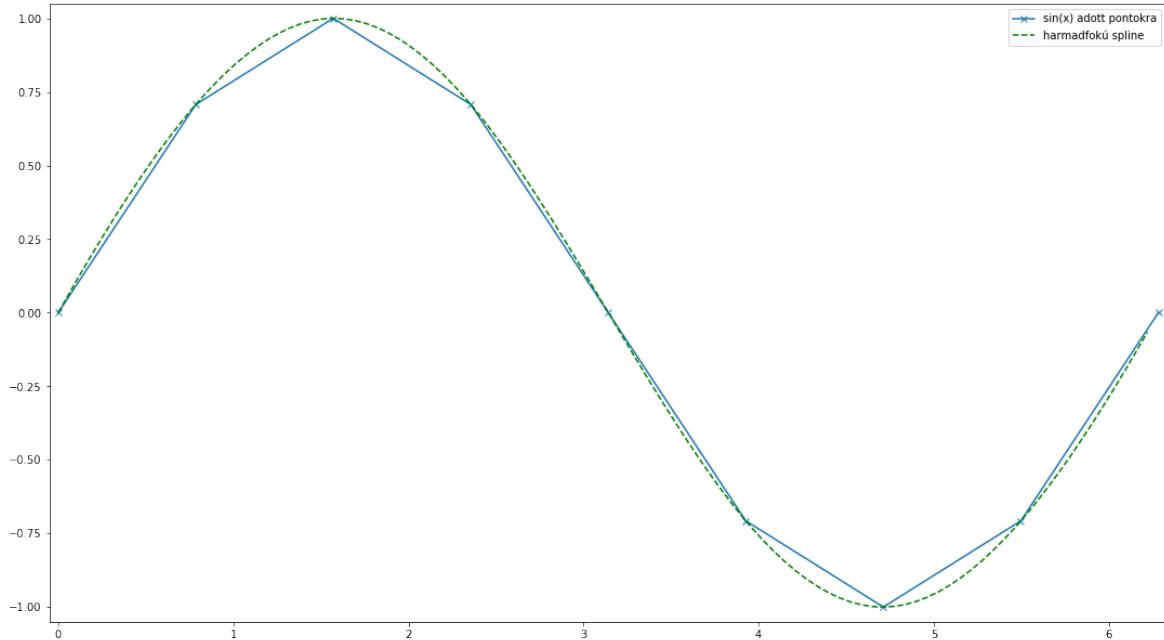
5.6.3. Hermite interpoláció

A harmadik interpoláció amit csak megemlítek, az a Hermite interpoláció. Hermite interpoláció esetén ugyan szintén megvannak az $x_0, x_1, \dots, x_n \in [a, b]$ pontjaink, mint eddig és vannak mellé m_0, m_1, \dots, m_k ($k \leq n$) multiplicitások is úgy, hogy $\sum_{i=0}^k m_i = n + 1$, továbbá adottak az

$$f^{(j)}(xi) = y_{ij}, \quad (i = 0, \dots, k \quad \text{és} \quad j = 0, \dots, m-1)$$

értékek is. Feladatunk az, hogy megkeressük egy n -ed fokú P polinomot melyre igaz hogy:

$$P^{(j)}(xi) = y_{ij}, \quad (i = 0, \dots, k \quad \text{és} \quad j = 0, \dots, m-1).$$



5.7. ábra. Példa spline segítségével történő interpolációra.

5.7. Sajátérték, sajátvektor számítása

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

Legyen $A \in \mathbb{C}^{n \times n}$ 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

$$Ax = \lambda x.$$

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:

$$\phi(\lambda) = \det(A - \lambda I) = 0.$$

Ezt kifejtve megkapjuk a λ változó n -ed fokú polinomját, azaz a:

$$\phi(\lambda) = (-1)^n \lambda^n + p_{n-1} \lambda^{n-1} + \cdots + p_1 \lambda + p_0$$

karakterisztikus polinomot. Komplex számok körében ennek a polinomnak pontosan n darab zérushelye van, tehát egy $A \in \mathbb{C}^{n \times n}$ mátrixnak pontosan n darab sajátértéke van, ha figyelembe vesszük a multiplicitásokat.

Most térdünk rá a Python-ban való megvalósítására! A sajátértékeket és vektorokat a `numpy.linalg` csomagban található `eig` metódussal tudjuk kiszámolni az alábbi módon.

```

from numpy import linalg as LA
import numpy as np

A = np.random.randint(10, size=16)
A = np.reshape(A, (4, 4))

w, v = LA.eig(A)
print("Sajatértékek:\n")
print(w)
print("\nSajatvektorok:")
print(v)

```

Az `eig` egy $n \times n$ -es mátrixot vár és két vektorral tér vissza az első (`w`) tartalmazza a sajátértékeket, a második (`v`) pedig a saját vektorokat normalizált alakban. Az i -edik (`w[i]`) sajátértékhez az i -edik (`v[:, i]`) a oszlop tartalmazza a sajátvektort.

A `numpy.linalg` csomagban találunk még egy `eigvals` függvényt, mely csak a sajátértékeket számolja ki, és ugyanúgy egy $n \times n$ -es mátrixot vár paraméterként.

```

w = LA.eigvals(A)
print("Sajatértékek:\n")
print(w)

```

Ezek a metódusok a `_geev` LAPACK rutint használják.

5.8. Legkisebb négyzetek módszere

Nézzünk meg egy példát a legkisebb négyzetek módszerének szemléltetésére Python segítségével!

Legyen a feladatunk az, hogy kapott mérési eredményekre rálillesszünk egy egyenest! Adottak a mért értékek $y \in R^n$ és a hozzájuk tartozó helyek $x \in R^n$. Ezek alapján keressük a $y = a_0 + a_1x$ egyenest melyre a

$$\sum_{i=0}^N [y_i - (a_0 + a_1x_i)]^2$$

minimális. Szerencsénkre erre is tartalmaz beépített metódust a NumPy. A `linalg` csomagban található `lstsq` metódus megadja az egyenest amire szükségünk van:

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
y = np.array([-1, 0.2, 0.9, 2.1, 3.2, 3.8, 4.0, 4.3, 4.8])
a = np.vstack([x, np.ones(len(x))]).T

m, c = np.linalg.lstsq(a, y, rcond=None)[0]

plt.figure(figsize=(18.5, 10.5), dpi=150)

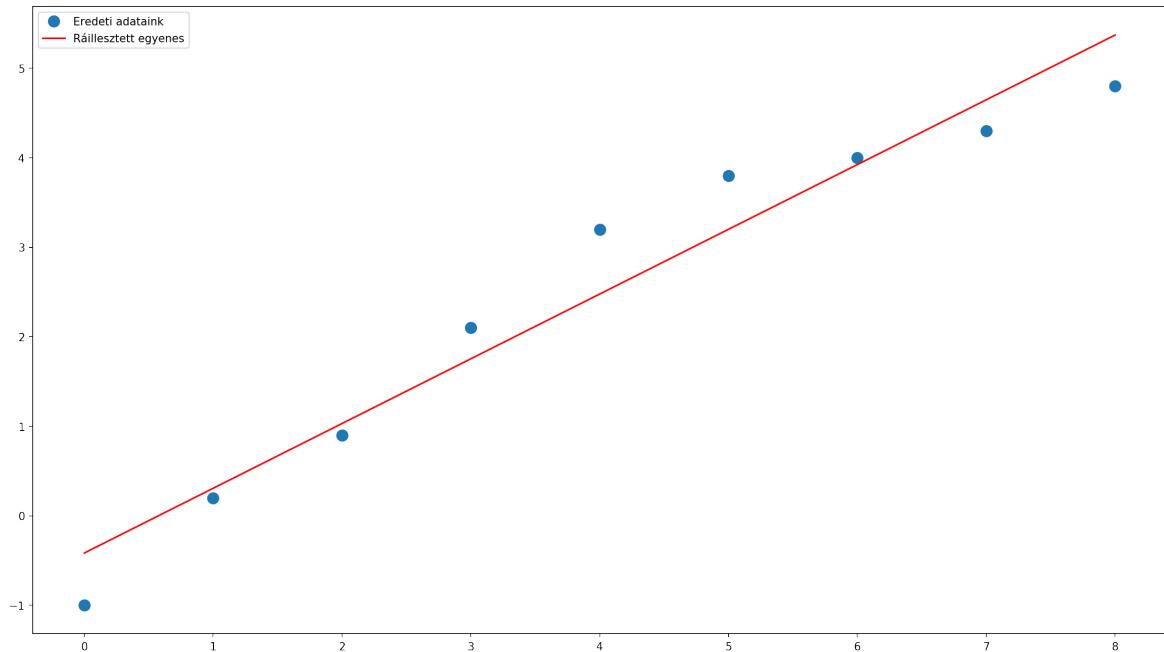
```

```

plot = plt.plot(x, y, 'o', label='Eredeti adataink', markersize=10)
plot = plt.plot(x, m*x + c, 'r', label='Ráillesztett egyenes')
plot = plt.legend()
plt.show(plot)

```

A program futásának eredménye az 5.8. ábrán tekinthető meg.



5.8. ábra. A legkisebb négyzetek módszerének szemléltetése.

5.9. Numerikus deriválás

A numerikus deriválás alapfeladata az, hogy az analitikusan ismeretlen, vagy nehezen számolható, esetleg csak diszkrét pontokban ismert $f : D(\subseteq \mathbb{R}) \rightarrow \mathbb{R}$ függvény deriváltját közelítsük adott pontokban.

Többféleképpen indulhatunk neki. Készíthetünk például egy interpolációt, mely közelít az f függvényünkhez, és akkor az interpolációnk k -adik deriváltja fog közelíteni az f függvényünk k -adik deriváltjához.

Ha $f \in C^2[a, b]$, akkor felírhatjuk rá a másodfokú Taylor polinomot, amiből kifejezhetjük az alábbi összefüggést:

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

melynek hibája $O(h)$. A harmadfokú Taylor polinommal még pontosabb közelítést kaphatunk:

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

Másodfokú deriváltra alkalmazhatjuk az alábbi összefüggéseket:

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

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

A második képlet a centrális differencia formula. Mind a két képlet hibája $O(h^2)$.

Ezek a következő formában implementálhatók Python-ban:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2 + x**3 + 3

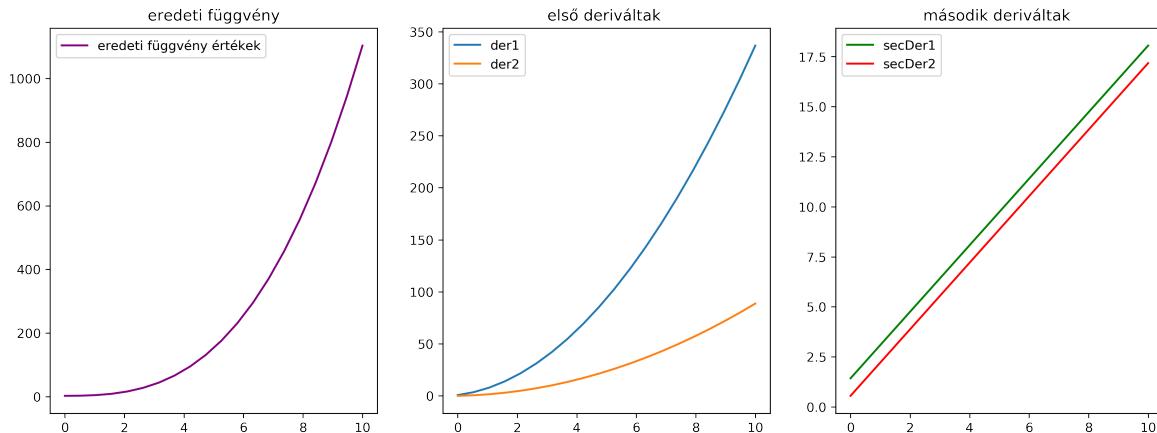
def firstDerivate1(x, h):
    return (f(x + h) - f(x)) / h

def firstDerivate2(x, h):
    return (f(x + h) - f(x - h)) / 2*h

def secondDerivate1(x, h):
    return (f(x) - 2*f(x + h) + f(x + 2*h)) / h*h

def secondDerivate2(x, h):
    return (f(x + h) - 2*f(x) + f(x - h)) / h*h
```

Polinom segítségével megadott pontokkal a módszert az 5.9. ábrán látható formában szemléltethetjük.



5.9. ábra. A numerikus deriválás szemléltetése.

Interpolációkkal való közelítéshez használható többek között a Newton, Lagrange és a Spline interpoláció is.

5.10. Numerikus integrálás

Numerikus integrálásnál a Newton-Leibnitz formulából indulunk ki. Ha adott egy f függvény és az Riemann integrálható $[a, b]$ -n, és itt létezik primitív függvénye, akkor

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

Ezt a képletet viszont csak akkor tudjuk alkalmazni, ha létezik f -nek primitív függvénye, egyébként numerikus integrálást kell alkalmaznunk.

Numerikus integrálást úgy tudunk alkalmazni, ha az f függvényünket közelítjük egy p interpolációs polinommal, és ezáltal a határozott integrál értékét általános formában közelíthetjük:

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

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

Integráláshoz találunk előre megírt függvényeket a `scipy.integrate` csomagban a `quad`, `dblquad`, `tplquad` formájában. A `quad` egy általános esetekben használható függvény, a `dblquad` a `quad`-hoz hasonló de kétszeres integrálást végez, míg a `tplquad` pedig háromszorosan integrál. Az alábbi példában szereplő módon használható:

```
import scipy.integrate as integrate

def f(x):
    return x + x**2 + 3

x = np.linspace(0, 10, num=20)
y = x**2 + x**3 + 3
result = integrate.quad(f, 0, 10)
```

Vannak természetesen egyébb algoritmusok is, melyeket lehet használni, ilyen a Lagrange formula vagy a zárt vagy nyitott Newton-Cotes formulák is. Ezek mellett ott vannak még a téglalap és trapéz formulák. A Newton-Cotes formulákra visszavezethető a Simpson és az érintő formula is.

Nézzük meg ezek közül először a két legegyszerűbbet a téglalap formulákat és a trapéz módszert. Kezdjük a téglalap formulákkal!

Az f függvény integrálját szeretnénk az $[a, b]$ intervallumon meghatározni. Legyen $h = (b - a)/n$ a lépésköz ekkor az első téglalap formula a következő:

$$I^{(1)} = h \sum_{j=0}^{n-1} y_j,$$

a második pedig:

$$I^{(2)} = h \sum_{j=1}^n y_j,$$

és van egy harmadik is:

$$I^{(3)} = h \sum_{j=1}^n y_{j-1/2}.$$

Látszik hogy a formulák a téglalapot illesztenek a függvényre és ezek területének az összege adja meg az integrál értékét. Ezek Python implementációja például az alábbi lehet:

```
def teglalap1(x, h, n):
    ossz = 0
    for i in range(0, n-1):
```

```

    ossz = ossz + f(x[i])
    return h * ossz

def teglalap2(x, h, n):
    ossz = 0
    for i in range(1, n):
        ossz = ossz + f(x[i])
    return h * ossz

def teglalap3(x, h, n):
    ossz = 0
    for i in range(1, n):
        x1 = (x[i-1] + x[i]) / 2
        ossz = ossz + f(x1)
    return h * ossz

```

Ezek nem ugyanazt az eredményt adják, de elég sok alpont esetén az egzakt megoldáshoz tartanak. A téglalap formula hibája az alpontok számától függ lineárisan, azaz $O(h)$.

Most nézzük a Trapéz módszert! A trapéz módszernek van egy egyszerű és egy összetett változata is. Először vegyük az egyszerűt. Ebben az esetben 1 alpontunk van tehát $h = 1$. Legyen $a = x_0$ és $b = x_1$ ekkor:

$$\int_a^b f(x)dx \approx \frac{y_0 + y_1}{2}h.$$

A számítás az adott intervallumra az alábbi:

```

def trapez_egyszeru(x, a, b, h):
    return (f(a) + f(b)/2) * h

```

Ez az esetek jelentős részében nem ad elég pontos eredményt. most nézzük meg az összetett trapéz módszert.

Az Összetett trapéz módszer esetén már több alpontuk van tehát $h \geq 1$ és az alábbi képletet alkalmazzuk:

$$\int_a^b f(x)dx \approx \sum_{j=0}^{n-1} \frac{x_{i+1} - x_i}{2}(y_i + y_{i+1}).$$

A hozzá tartozó Python implementáció a következő:

```

def trapez_osszetett(x, n):
    ossz = 0
    for i in range(0, n-1):
        xi = (x[i+1] - x[i]) / 2
        ossz = ossz + xi * (f(x[i]) + f(x[i+1]))
    return ossz

```

Ez már sokkal pontosabb eredményt ad, mint az egyszerű trapéz módszer.

Folytassuk az érintő formulával! Az érintő formula egy nyílt Newton-Cotes formula melyre:

$$\int_a^b f(x)dx \approx (b-a)f\left(\frac{a+b}{2}\right)$$

Ez a formula úgy is értelmezhető, hogy az f függvényünket a középpontjához húzott egyeneskel közelítjük az $[a, b]$ intervallumon, és az egyenes alatti területet vesszük. Ebből következik, hogy maximum első fokú polinomig pontos. Gyakorlatban nem ezt a képletet szokták alkalmazni, hanem az intervallumot felosztják n egyenlő részre és a következő képletet alkalmazzák:

$$\int_a^b f(x)dx \approx \frac{(b-a)}{n} \sum_{i=1}^n f\left(a - \frac{h}{2} + ih\right)$$

Nézzük meg ezek implementációját!

```
def erinto_egyszeru(x, a, b):
    return (b - a) * (f(b + a) / 2)

def erinto_osszetett(x, a, b, n, h):
    d = (b - a) / n
    ossz = 0
    for i in range(1, n):
        xi = a - (h / 2) + (i * h)
        ossz = ossz + f(xi)
    return ossz * d
```

Az egyszerű érintő formula hiba becslése:

$$\left| \int_b^a f(x)dx - (b-a)f\left(\frac{a+b}{2}\right) \right| \leq \frac{(b-a)^3 M_2}{24}.$$

Végül jöjjön a Simpson formula mely egy zárt Newton-Coats formula, melyre $n = 2$. Ennek is van egyszerű és összetett alakja. Az egyszerű alakhoz 3 alpontra van szükséünk $x_1 = a$, $x_1 = \frac{a+b}{2}$, $x_2 = b$ és alakalmazzuk a három pontra támaszkodó Lagrange-féle interpolációs polinomot:

$$\int_a^b f(x)dx \approx \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$

Az összetett Simpson formula esetén is az intervallumot felosztjuk n egyenlő részre, és akkor ilyen módon változik meg a képletünk:

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

Ezeknek az implementációja az alábbi:

```
def simpson_egyszeru(x, a, b):
    return ((b-a)/6) * (f(a)+ 4*f((a+b)/2)+f(b))
```

```
def simpson_osszetett(x, n):
    ossz = 0
    for i in range(0, n-1):
        x1 = (x[i+1]-x[i])/6
        x2 = (f(x[i])+ 4*f((x[i]+x[i+1])/2)+f(x[i+1]))
        ossz = ossz + (x1*x2)
    return ossz
```

A Simpson formula hibája:

$$\left| \int_b^a f(x)dx - S_n(f) \right| \leq \frac{M_4(b-a)}{32 \cdot 90} h^4 = \frac{M_4(b-a)^5}{2880n^4}.$$

6. fejezet

Megjelenítési módok

Az eredmények megjelenítésére már a korábbiakban is sor került. Ez a fejezet külön egységként vizsgálja, hogy a Python, főként a `matplotlib` segítségével milyen segítséget ad az adatok vizualizálásához.

6.1. Egyszerű grafikonok rajzolása

Ahogy a szoftvereszközök bemutatásánál már volt róla szó, az adatok vizualizálása nagyon fontos, hisz egy-egy ábráról gyakran könnyebb adatokat leolvasni, mint nagy táblázatokból vagy egyéb adastruktúrákból. Python-ban a `matplotlib` függvénykönyvtárt használhatjuk ezeknek az ábráknak a létrehozásához [?].

Először vegyük egy egyszerű példát, amelyben bizonyos x értékekhez rendeljünk hozzá $f(x) = y$ értékeket.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.zeros(20)
y = np.zeros(20)

for i in range(-9, 10):
    x[i] = i
    y[i] = pow(x[i], 2)

plot = plt.plot(x, y, "o")
```

A `matplotlib plot` hívásával tudjuk kirajzolatni az ábráinkat, melynek az első paramáttere a értelmezési tartomány, a második az értékkészlet, és további paramátereként meg lehet neki adni a jelölés formáját, illetve feliratokat a label paraméter segítségével.

Az értékeket megadhatjuk függvény segítségével is, például

```
import matplotlib.pyplot as plt
import numpy as np

x = np.zeros(20)
y = np.zeros(20)
```

```

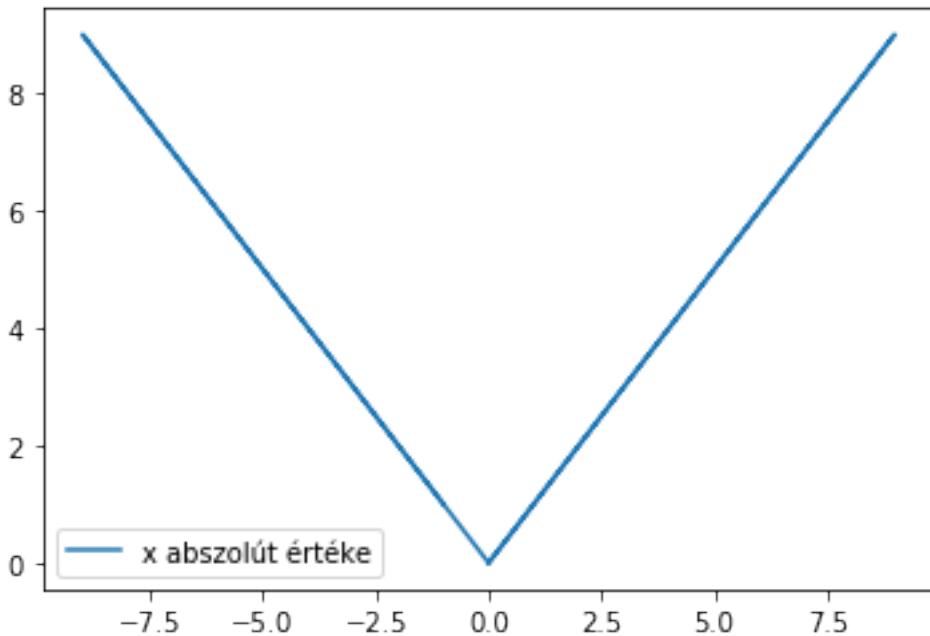
for i in range(-9, 10):
    x[i] = i

plot=plt.plot(x ,abs(x),"—", label='x abszolut értéke', markersize=10)
plot = plt.legend()

plt.show(plot)

```

Ennek az eredménye a 6.1. ábrán látható.



6.1. ábra. Függvény kirajzolása a `matplotlib` segítségével.

Megváltoztathatjuk a színét is az adott ábránknak, illetve több függvényt is ábrázolhatunk egyszerre, például

```

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.pyplot import figure

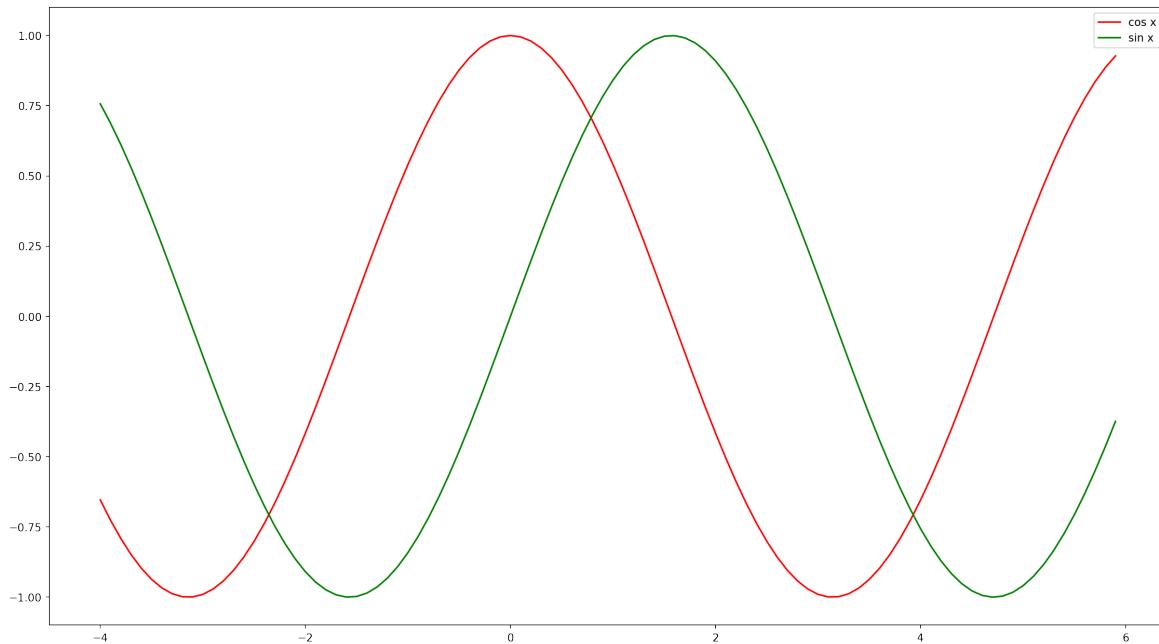
x = np.zeros(100)
d = -4
i = 0
while i < 100:
    x[i] = d;
    d = d + 0.1
    i = i + 1

plt.figure(figsize=(18.5, 10.5), dpi=150)
plot = plt.plot(x,np.cos(x), "-r", label='cos x')
plot = plt.plot(x,np.sin(x), "-g", label='sin x')
plot = plt.legend()

```

```
plt.show(plot)
```

Az eredmény a 6.2. ábrán szerepel.



6.2. ábra. Több függvény egyidejű kirajzolása.

Az előző kódrészben importáltam a `figure`-t a `matplotlib` csomagból. Ennek segítségével képesek vagyunk beállítani az ábra alapvető tulajdonságait, úgy mint például a méretet, de beállíthatjuk a háttér és szélek színeit vagy a felbontást.

A tengelyeket is elnevezhetjük könnyedén a `matplotlib` `ylabel` és `xlabel` metódusaival és adhatunk címet is az ábránknak a `title` metódussal.

Létrehozhatunk hisztogrammokat is, továbbá adatainkat egy ploton belül töbféle-képpen is megjeleníthetjük:

```
test = ['1', '2', '3', '4', '5', '6']
points = [68, 70, 75, 76, 80, 78]

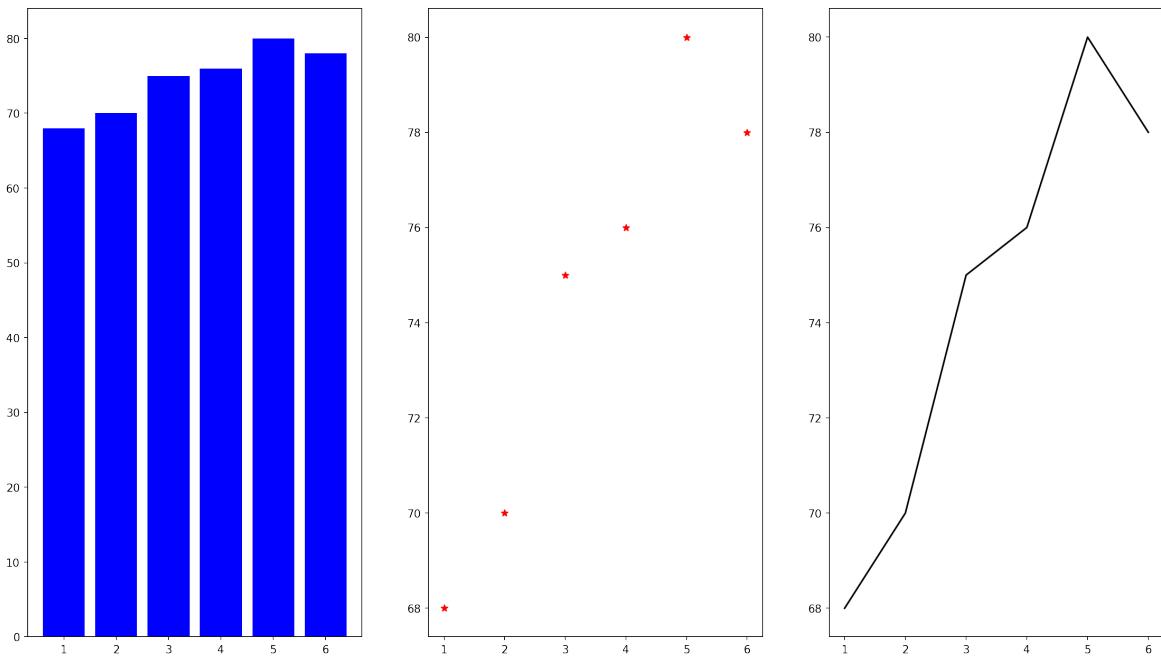
plt.figure(figsize=(18.5, 10.5), dpi=150)

plt.subplot(131)
plt.bar(test, points, color='b')
plt.subplot(132)
plt.scatter(test, points, marker='*', color='r')
plt.subplot(133)
plt.plot(test, points, color='black')

plt.show()
```

Az eredménye a 6.3. ábrán látható.

Ilyen esetben a jelölő pontokat a `marker`, míg a színt a `color` paraméterrel tudjuk megváltoztatni.



6.3. ábra. Több grafikon megjelenítése egymás mellett.

6.2. Térbeli pontok szemléltetése

A Matplotlib megengedi számunkra az is, hogy térbeli pontokat képezzünk le ábráinkon. Ezek lehetnek felületek térben elszórt pontjai vagy egyéb függvények is. Ehhez szükségünk van a `mplot3d` csomagra ami az `mpl_toolkits` része [16]. Először nézzünk meg egy térbeli ívet!

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

fig = plt.figure(figsize=(18.5, 10.5), dpi=150)
ax = fig.add_subplot(111, projection='3d')

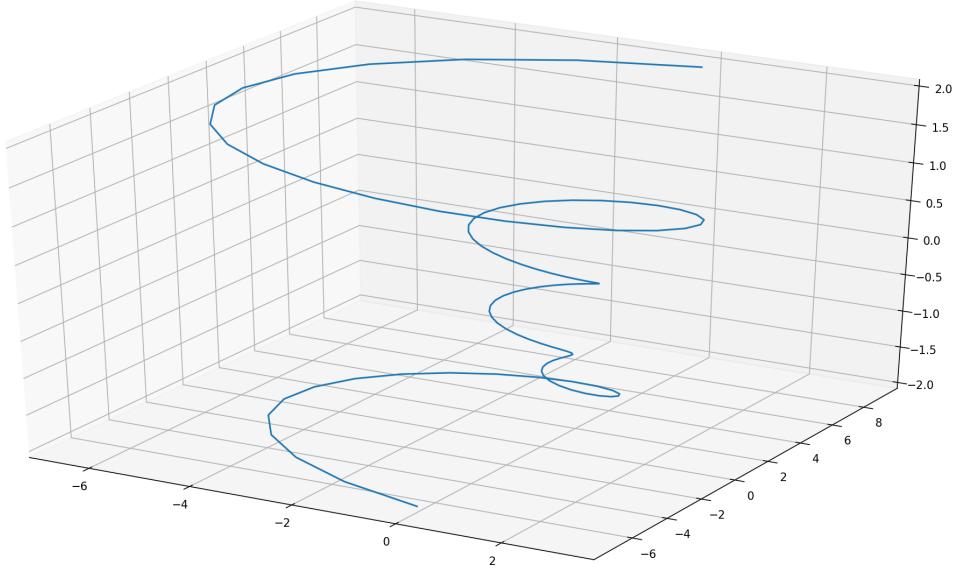
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**3 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z)
plt.show()
```

A kirajzolás eredménye a 6.4. ábrán látható.

A pontok által leírt ívet jeleníti meg egyetlen töröttvonalként, de a `matplotlib` segítségével akár elhelyezhetjük rajta az adott diszkrét pontokat is (6.5. ábra):

```
fig = plt.figure(figsize=(18.5, 10.5), dpi=150)
ax = fig.add_subplot(111, projection='3d')

theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
```



6.4. ábra. Térbeli ív szemléltetése.

```

zline = np.linspace(-2, 2, 100)
r = z**3 + 1
xline = r * np.sin(theta)
yline = r * np.cos(theta)
ax.plot(xline, yline, zline)

zpoints = np.linspace(-2, 2, 100)
xpoints = r * np.sin(theta)
ypoints = r * np.cos(theta)
ax.scatter3D(xpoints, ypoints, zpoints, c=z, cmap="Reds")
plt.show()

```

Megtehetjük azt is hogy cak a pontokat helyezzük el a 3D-s térben (6.6. ábra):

```

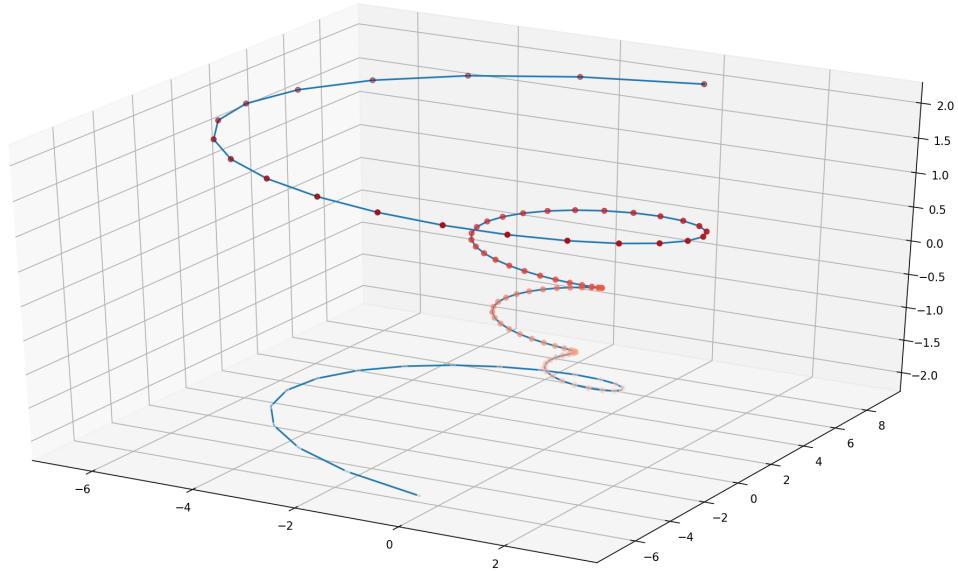
fig = plt.figure(figsize=(18.5, 10.5), dpi=150)
ax = fig.add_subplot(111, projection='3d')

theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
r = z**3 + 1
zpoints = np.linspace(-2, 2, 100)
xpoints = r * np.sin(theta)
ypoints = r * np.cos(theta)
ax.scatter3D(xpoints, ypoints, zpoints, c=z, cmap="hsv")
plt.show()

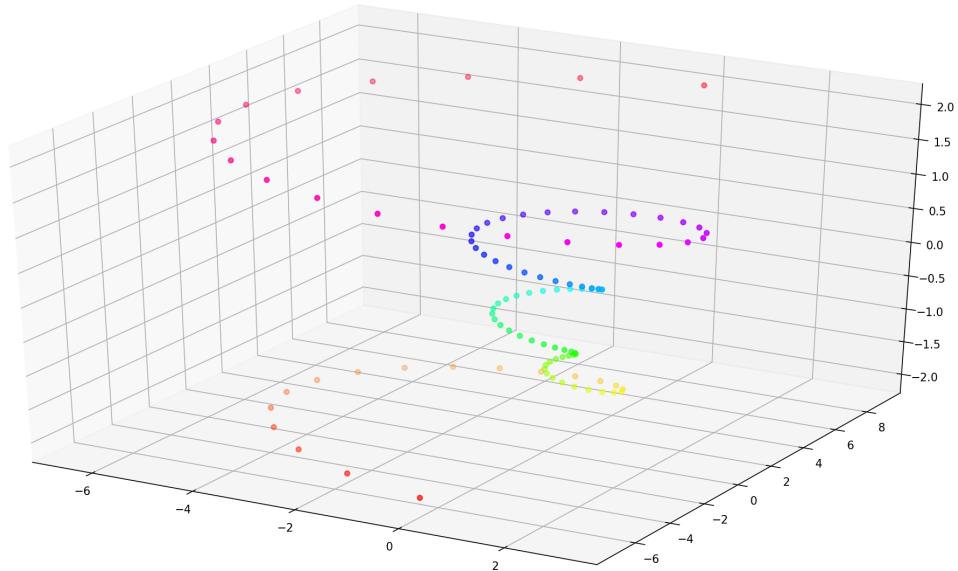
```

Egy felületet megjeleníthetünk többféleképpen is a **wireframe** segítségével egy háló szerűen, míg a **surface** használatával egybefüggő felületként. A következő példában már a kapott térbeli felületi elforgatása is szerepel:

```
def f(x, y):
```



6.5. ábra. Térbeli ív szemléltetése diszkrét pontokkal együtt.



6.6. ábra. Térbeli pontok szemléltetése HSV színezéssel.

```

return np.sin(np.sqrt(x ** 2 + y ** 2))

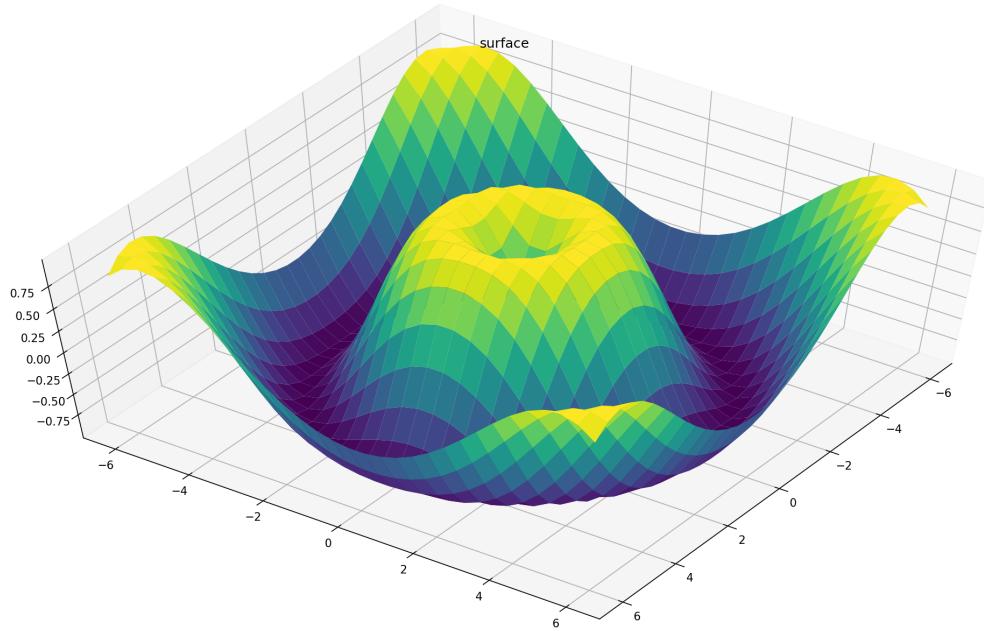
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)

```

```
plt.figure(figsize=(18.5, 10.5), dpi=150)
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax.set_title('surface');
ax.view_init(60, 35)
```

Az így megjelenített felület a 6.7. ábrán látható.



6.7. ábra. Térbeli felület megjelenítése.

Sok más további lehetőséget is kínál a matplotlib, például megjeleníthetünk 3d-ben 2-ds adatokat, vagy készíthetünk 3d-s hisztogrammot több adattal, de a fent említettek talán a legfontosabbak és legtöbbet használtak.

6.3. Interaktív widgetek Jupyter-ben

Az interaktív plotokról csak említés szintjén esik szó a dolgozatban, hiszen ezek készítését nem a Python, hanem a Jupyter notebook és a IPython kernel teszi lehetővé. Az `interact` segítségével tudunk interaktív dolgokat készíteni, például azért, hogy több adatra megnézhessük egy függvény képét. A következő példában különböző hatvány függvényeket lehet megnézni, és a csúszka segítségével lehet beállítani a hatvány kitévőt:

```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np
from ipywidgets import Button, Layout
```

```
def f(y):
    x = np.linspace(-100, 100, 100)
    plt.figure(figsize=(14, 7), dpi=150)
    plot = plt.plot(x, x**y)
    plt.show()

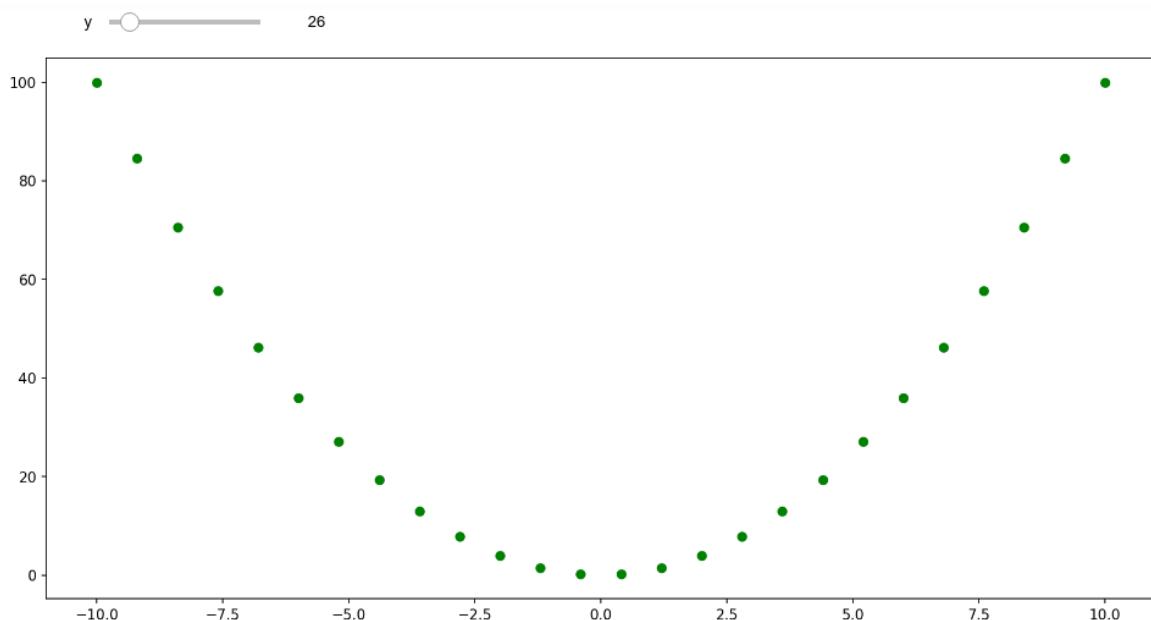
interact(f, y=widgets.IntSlider(min=1, max=30, step=1, value=0));
```

A minta számot is beállíthatjuk egy ilyen csúszkával és ezzel a pontosságot tudjuk szabályozni:

```
def f(y):
    x = np.linspace(-10, 10, y)
    plt.figure(figsize=(14, 7), dpi=150)
    plot = plt.plot(x, x**2, 'o', color="green")
    plt.show()

interact(f, y=widgets.IntSlider(min=1, max=200, step=1, value=0));
```

Egy értékbeállítást láthatunk a 6.8. ábrán.



6.8. ábra. Csúszka használata.

Az interaktivitásnak sok további formája van. Például vannak még igaz/hamis értékekre állítható jelölőnégyzet (Checkbox), szöveges bevitelre alkalmas mező (Text), lenyíló lista (Dropdown). Itt szerepel egy kis példa egy szöveges mezőre mely összead két számot.

```
def f(a, b):
    if a == "":
        x = 0
    else:
        x = float(a)
```

```

if b == "":
    y = 0
else:
    y = float(b)
print(x + y)

interact(f, a="", b="");
```

Fontos megjegyezni, hogy a beírt dolgok szövegként kerülnek átadásra, ezért szükség lehet azok számmá való konvertálására, ahogy az előbbi példában is látható.

Megvalósításából fakadóan ezek az interaktív eszközök használhatóak 3D plotoknál, illetve más helyeken is. Különösen hasznos lehet például térben ábrázolt felület forgatásánál. Ez az alábbi kód részlettel valósítható meg.

```

from mpl_toolkits.mplot3d import axes3d

def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

def g(angle1, angle2):
    x = np.linspace(-6, 6, 30)
    y = np.linspace(-6, 6, 30)

    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)

    plt.figure(figsize=(14, 7), dpi=150)
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                    cmap='viridis', edgecolor='none')
    ax.set_title('surface');
    ax.view_init(angle1, angle2)

interact(g,
         angle1=widgets.IntSlider(min=0, max=360, step=1, value=0,
                                   layout=Layout(width="600px")),
         angle2=widgets.IntSlider(min=0, max=360, step=1, value=0,
                                   layout=Layout(width="600px")));
```

7. fejezet

Összefoglalás

A dolgozat példákkal illusztrálva azt igyekezett bemutatni, hogy a numerikus módszerek témakörében milyen hatékonyan használható a Python programozási nyelv és a hozzá tartozó eszközök készlet. Elsősorban a klasszikus numerikus problémákra koncentrált, amelyek jellemzően a Numerikus módszerek tárgy kurzusainak témakörei között is szerepelnek.

A hatékonyság alatt elsősorban a Python adta olvasható forráskód és a Jupyter munkafüzetek használati módjából következő egyszerűbb kezelési mód kapott szerepet. E mellett érdemes lehet a későbbiekben megvizsgálni még az adatok importálásával és exportálásával kapcsolatos előnyöket is, melyek szintén a Python használatából, komoly támogatottságából adódnak.

A dolgozathoz készített Jupyter munkafüzetek remélhetőleg később hasznosíthatóak lesznek majd a Numerikus módszerek tárgy oktatásában is.

Irodalomjegyzék

- [1] Házay Attila. Numerikus módszerek, egyetemi jegyzet. <https://www.uni-miskolc.hu/~matha>, 2020.
- [2] Ian D Chivers and Jane Sleighholme. *Introduction to Programming with FORTRAN*, volume 2. Springer, 2018.
- [3] TIOBE Software Quality Company. Tiobe index. <https://www.tiobe.com/tiobe-index>, 2020.
- [4] NumPy developers. Numpy documentation. <https://numpy.org/doc>, 2019.
- [5] SciPy developers. Scipy documentation. <https://docs.scipy.org/doc>, 2020.
- [6] Vincent E Dimiceli, Andrew SID Lang, and LeighAnne Locke. Teaching calculus with wolfram| alpha. *International Journal of Mathematical Education in Science and Technology*, 41(8):1061–1071, 2010.
- [7] Python Foundation. Python programozási nyelv. <https://www.python.org>, 2020.
- [8] Python Software Foundation. Python package index. <https://pypi.org>, 2020.
- [9] William P Fox and William C Bauldry. *Advanced Problem Solving with Maple: A First Course*. CRC Press, 2019.
- [10] Quazi Nafiu Islam. *Mastering PyCharm*. Packt Publishing Ltd, 2015.
- [11] Project Jupyter. Jupyter. <https://jupyter.org>, 2020.
- [12] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.
- [13] Holly Moore. *MATLAB for Engineers*. Pearson, 2017.
- [14] Roger D Peng. *R programming for data science*. Leanpub, 2016.
- [15] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.
- [16] Jake VanderPlas. *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.", 2016.

A CD melléklet tartalma

A CD-n a következő mellékletek szerepelnek.

- A dolgozatot egy `dolgozat.pdf` fájl formájában,
- A dolgozat `LATEX` forráskódja a `szakdolgozat` jegyzékben.
- A dolgozathoz készített Jupyter munkafüzetek a `notebooks` jegyzékben.

A cikkben/előadásban/tanulmányban ismertetett kutató munka az EFOP-3.6.1-16-2016-00011 jelű „Fiatalodó és Megújuló Egyetem – Innovatív Tudásváros – a Miskolci Egyetem intelligens szakosodást szolgáló intézményi fejlesztése” projekt részeként – a Széchenyi 2020 keretében – az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg”